Vom Fachbereich für Mathematik und Informatik

der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

Christian Lindig

Algorithmen zur Begriffsanalyse und ihre Anwendung bei Softwarebibliotheken

19. November 1999

1. Referent: Prof. Dr. Gregor Snelting

2. Referent: Prof. Dr. Bernhard Ganter

Eingereicht am: 4. August 1999

Christian Lindig
Institut für Software
Technische Universität Braunschweig
D–38106 Braunschweig
lindig@ips.cs.tu-bs.de

Algorithmen zur Formalen Begriffsanalyse und ihre Anwendung bei Softwarebibliotheken

Zusammenfassung

Formale Begriffsanalyse ist eine algebraische Theorie über binäre Relationen und zu ihnen in enger Verbindung stehende vollständige Verbände. Diese Arbeit präsentiert Algorithmen und Datenstrukturen zur Berechnung von Begriffen und ihrer Verbandsstruktur. Da die Anzahl der Begriffe im ungünstigsten Fall exponentiell mit der Größe der Relation ansteigen kann, wurde der Komplexität der Algorithmen besondere Aufmerksamkeit geschenkt. Sowohl die Laufzeit der Algorithmen, als auch die tatsächliche Größe von Begriffsverbänden wurde durch eine Vielzahl von Experimenten untersucht. Sie zeigen, daß für praktische Anwendungen die Laufzeit der Algorithmen und die Größe der Verbände nur quadratisch von der Größe der Relation abhängt. Als Anwendung der Begriffsanalyse wird die Organisation einer Bibliothek wiederverwendbarer Software-Komponenten vorgeschlagen. Softwarewiederverwendung zielt auf eine Steigerung der Software-Qualität und der Produktivität bei ihrer Erstellung. Die vorgeschlagene Methode vereinigt eine leichte Wartbarkeit der Komponentensammlung mit einer starken Unterstützung für den Anwender bei der Suche nach Komponenten. Das Werkzeug zur Suche verwendet als Datenstruktur den Begriffsverband, der einmalig für eine Sammlung berechnet wird. Der Verband enthält im Wesentlichen alle Entscheidungsmöglichkeiten eines Anwenders bei der Suche nach Komponenten und unterstützt die effiziente Bearbeitung einer Anfrage. Zusätzlich kann mit Hilfe des Verbandes die Qualität der Indexierung von Softwarekomponenten beurteilt werden.

Abstract

Formal concept analysis is an algebraic theory concerning binary relations and closely related complete lattices of so-called concepts. This thesis presents algorithms and data structures to compute concepts and their lattice structure. Since, in the worst case, the number of concepts can grow exponentially with the size of a relation, the complexity of algorithms was given special attention. The speed of algorithms as well as the actual size of concept lattices was tested with a large number of test cases. They show that for practical applications the performance of algorithms and the number of concepts depends only quadratically on the size of the relation. As an application for concept analysis the organization of a library of re-usable software components is proposed. Component based software reuse aims to raise software quality and development productivity by re-using already successfully developed components. The proposed method combines good maintainability of the component library with strong navigation support for the user. The search tool uses the concept lattice as a data structure that is computed once for the library. The lattice essentially contains all decisions a user can take while searching for a component and thus supports efficient searching. Additionally the lattice permits reasoning about the quality of the indexing method used on the components.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In den vergangenen 10 Jahren ist in der Informatik der Anteil der theoretischen Veröffentlichungen beständig gestiegen: Abbildung 1.1 zeigt ihren prozentualen Anteil an den über 280 000 durch den *ACM Guide to Computing Literature* [99] erfaßten Zeitschriften- und Konferenzbeiträgen, Büchern und Dissertationen. Für den Spezialfall der formalen Begriffsanalyse [32] versucht diese Arbeit, Theorie und Praxis zu verbinden [102]. Formale Begriffsanalyse ist eine algebraische Theorie über binäre Relationen und zu ihnen in enger Verbindung stehende vollständige Verbände. Diese Arbeit untersucht die Algorithmen und Datenstrukturen zur Implementierung formaler Begriffe sowie ihre Komplexität. Als Anwendung wird die Organisation von Bibliotheken mit wiederverwendbaren Softwarekomponenten durch formale Begriffe vorgeschlagen und an Hand eines Prototypen demonstriert. Softwarewiederverwendung [56] zielt auf eine Qualitäts- und Produktivitätssteigerung bei der Erstellung von Software; die Ablage von Komponenten zu ihrer späteren Wiederverwendung ist dabei die häufigste Form der Wiederverwendung.

Formale Begriffsanalyse ist eine mathematische Theorie aus dem Gebiet der Algebra [32]. Im Grundsatz beschäftigt sie sich mit binären Relationen $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$, die man sich anschaulich als eine 2-dimensionale Tabelle vorstellen kann, deren Einträge aus Kreuzen bestehen – wie in Abbildung 1.2 skizziert. Die Kreuze einer Tabelle lassen sich zu verschiedenen maximalen Rechtecken zusammenfassen, wenn man eine Umordnung der Zeilen und Spalten zuläßt: die Tabelle in Abbildung 1.2 enthält 4 Rechtecke, die wegen der Größe der Tabelle nur aus schmalen Streifen bestehen. Formale Begriffsanalyse ist, vereinfachend gesagt, die Theorie der maximalen Rechtecke in zweidimensionalen Tabellen. Sie untersucht die algebraischen Eigenschaften dieser Rechtecke und stellt Algorithmen zu ihrer Berechnung bereit.

Die Elemente der beiden Kanten eines maximalen Rechteckes bilden zwei Mengen (O, A); jedes maximale Rechteck wird in der Begriffsanalyse als *Begriff* und seine Bestandteile O und A als *Inhalt* und *Umfang* bezeichnet. Die Bezeichnungen drücken eine gewisse Ähnlichkeit formaler Begriffe mit natürlichsprachlichen Be-

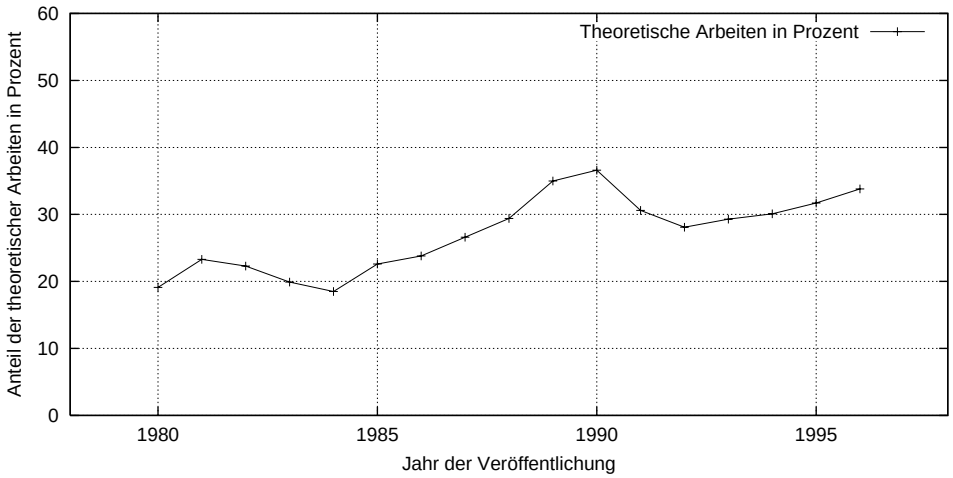

Abbildung 1.1: Anteil der theoretischen im *ACM Electronic Guide to Computing Literature* erfaßten Veröffentlichungen. Eine Veröffentlichung wurde als theoretisch kategoriesiert, wenn sie in ihrem Titel oder ihren Deskriptoren die Wörter *formal* oder *theory* enthielt. Im Durchschnitt wurden im erfaßten Zeitraum 16 654 Arbeiten pro Jahr veröffentlicht.

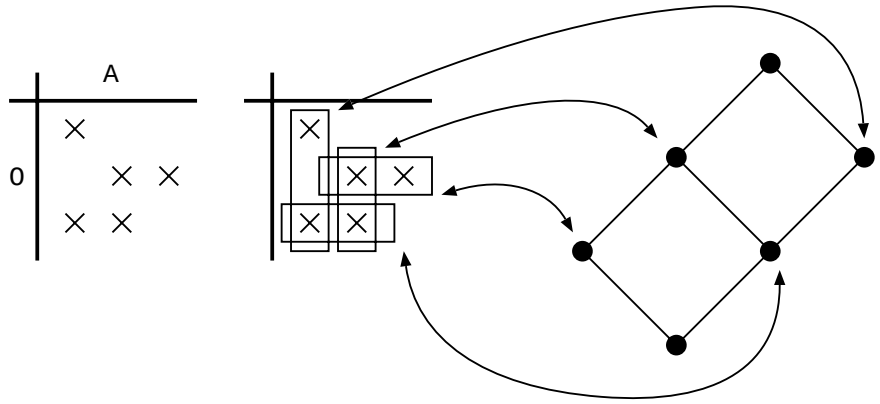

Abbildung 1.2: Das Prinzip der Begriffsanalyse: maximale Rechtecke in einer 2-dimensionalen Tabelle korrespondieren mit Elementen (*Begriffen*) eines zugehörigen Verbandes.

griffen aus. Speziell existiert auch bei formalen Begriffen eine Ordnung, die Ober- und Unterbegriffe unterscheidet. Alle Begriffe einer Tabelle (oder Relation) bilden gemeinsam einen *Begriffsverband* – rechts in Abbildung 1.2 skizziert. In diesem Verband existiert für jede Teilmenge von Begriffen ein eindeutiger kleinster gemeinsamer Oberbegriff und ein größter gemeinsamer Unterbegriff.

Der Begriffsverband einer Relation läßt viele ihrer Eigenschaften deutlich hervortreten, ohne sie allerdings zu interpretieren oder zu verändern. Der Name *Be-*

griffsanalyse dieser Theorie drückt bereits ihren analytischen Charakter aus. In den meisten Anwendungen der Begriffsanalyse werden als Tabellen vorliegende Situationen an Hand ihrer Begriffsverbände bewertet. Dies geschieht überwiegend manuell durch Experten, die den zugehörigen Begriffsverband interpretieren. Einige dieser Anwendungen aus dem Bereich des Software-Engineering werden im Abschnitt 7.3 kurz vorgestellt.

Für die in dieser Arbeit vorgeschlagene Anwendung, der Organisation von Softwarekomponenten, wird der Begriffsverband einer Ausgangsrelation vor allen Dingen als Datenstruktur aufgefaßt und nicht als ein dem Benutzer sichtbarer Verband. Abschnitt 5 beschreibt, wie mit seiner Hilfe Software-Komponenten abgelegt und interaktiv gesucht werden können. Der Begriffsverband der Komponenten und ihrer Indexierung mit Schlüsselwörtern wird dazu einmalig berechnet und als Datenstruktur abgelegt. Bei späteren Suchen von Komponenten erlaubt der Verband dann eine besonders effiziente Bearbeitung von Anfragen. Zusätzlich kann eine statistische Analyse des Begriffsverbandes die Qualität der Komponentenindexierung, und damit das Antwortverhalten auf Anfragen, bewerten.

Die Ablage von Softwarekomponenten dient ihrer späteren Wiederverwendung. Der Rückgriff auf bereits einmal erstellte und getestete Komponenten soll helfen, die Probleme des Programmierens-im-Kleinen zu überwinden, um sich so auf die Erstellung von Softwaresystemen im industriellen Maßstab konzentrieren zu können. Die Ablage von einmal erstellten Komponenten zu ihrer Wiederverwendung ist nur eine mögliche Ausprägung von Softwarewiederverwendung – allerdings die am besten untersuchte. Abschnitt 4 stellt die *prozeßbasierte* und speziell die *komponentenbasierte* Wiederverwendung mit Beispielen aus der Literatur vor. Dieser Abschnitt diskutiert außerdem Kriterien zur Auswahl von Methoden der komponentenbasierten Softwarewiederverwendung. Einen Vergleich zwischen den bekannten komponentenbasierten Methoden und der vorgeschlagenen begriffsbasierten nimmt Abschnitt 7 vor. Es zeigt sich, daß die begriffsbasierte Organisation von Komponenten eine Reihe wünschenswerter Eigenschaften besitzt, ohne den dafür in anderen Methoden nötigen manuellen Aufwand in Kauf nehmen zu müssen.

Neben dem Vorschlag, Begriffe zur Organisation von Softwarekomponenten zu verwenden, ist der wesentliche Beitrag dieser Arbeit die Untersuchung der Implementierung formaler Begriffsanalyse und ihrer Komplexität. Da die Anzahl aller Begriffe einer Relation exponentiell mit der Größe der Relation zunehmen kann, sind sowohl eine effiziente Implementierung aller Operationen, ihre Komplexität, als auch die tatsächliche Größe von Begriffsverbänden von praktischem Interesse. Abschnitt 3 untersucht die Implementierung der Begriffsanalyse, die zwei Hauptaufgaben besitzt: die erste besteht in der Berechnung der Menge aller Begriffe. Da die Verbandsstruktur der Begriffe nur implizit in ihnen enthalten ist, muß sie insbesondere für analytische Anwendungen berechnet und als Datenstruktur bereitgestellt werden; dieses zu leisten, ist die zweite Aufgabe einer Implementierung der Begriffsanalyse.

Wenn tatsächlich jeder Begriffsverband im Vergleich zu seiner Ausgangsrelation exponentiell viele Begriffe enthielte, wäre Begriffsanalyse trotz effizienter Algorithmen praktisch unbrauchbar. Versuche zeigen aber, daß dies zumindest für die vorgeschlagene Organisation von Softwarekomponenten nicht zu erwarten ist. Abschnitt 6 enthält eine Beschreibung dieser Versuche; sie wurden außerdem verwendet, um die Performance verschiedener Implementierungen zu vergleichen, und so Aussagen über den zu erwartenden Aufwand einer Analyse zu gewinnen.

Das nachfolgende Kapitel gibt eine knappe und notwendigerweise formale Einführung in die Begriffsanalyse. Die Schwerpunkte der anderen Kapitel wurden oben bereits kurz vorgestellt; sie orientieren sich konkret an der Aufgabe, Softwarekomponenten zu organisieren und Begriffe algorithmisch zu bestimmen. Der letzte Abschnitt 8 schließlich faßt die Ergebnisse dieser Arbeit zusammen.

Kapitel 2

Formale Begriffsanalyse

Die formale Begriffsanalyse ist eine mathematische Theorie aus dem Gebiet der Algebra. Sie wurde bereits 1940 von Garrett Birkhoff [7] begründet und seit 1982 von der Darmstädter Gruppe um Rudolf Wille weiterentwickelt. Eine Momentaufnahme davon ist in dem 1996 erschienenen Buch *Formale Begriffsanalyse* [32] von Ganter und Wille festgehalten, das ausführliche Literaturhinweise enthält. Der Gegenstand der formalen Begriffsanalyse sind binäre Relationen und zu ihnen in enger Verbindung stehende vollständige Verbände. Anwendungen der formalen Begriffsanalyse untersuchen Relationen aus ihrem Gebiet mittels der Begriffsanalyse. Die Ergebnisse in Form von Verbänden werden dann im Bereich der Anwendung interpretiert und können zu neuen Einsichten verhelfen. Auch die hier beschriebenen Anwendungen folgen diesem Schema. Bevor sie allerdings dargestellt werden können, folgt eine Einleitung in die formale Begriffsanalyse mit ihren zentralen Definitionen und Sätzen.

2.1 Kontext und Begriff

Der Ausgangspunkt der formalen Begriffsanalyse ist ein *formaler Kontext* – er ist der Gegenstand der späteren Analyse. Ein formaler Kontext ist eine binäre Relation:

Definition 1 (Kontext) *Ein formaler Kontext ist ein Tripel* $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ *aus den folgenden Mengen: eine Objektmenge* $\mathcal{O}$*, eine Attributmenge* $\mathcal{A}$ *und einer Relation* $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$ *zwischen ihnen. Die Schreibweise* $(o, a) \in \mathcal{R}$ *wird gelesen als: das Objekt* o *besitzt das Attribut* a*.*

Ein Kontext definiert, in welchen Beziehung eine Menge von Objekten zu einer Menge von Attributen steht[1]. Ein formaler Kontext wird meistens durch eine

[1] Beide Mengen können übrigens unendlich sein.

Kontexttabelle veranschaulicht, in der Kreuze den Elementen der Relation entsprechen. Objekte werden darin per Konvention als Zeilen, Attribute als Spalten dargestellt.

	klein	mittel	groß	nah	entfernt	Mond	kein Mond
Merkur	×			×			×
Venus	×			×			×
Erde	×			×		×	
Mars	×			×		×	
Jupiter			×		×	×	
Saturn			×		×	×	
Uranus		×			×	×	
Neptun		×			×	×	
Pluto	×				×	×	

(a) Kontext

	klein	nah	Mond	entfernt	mittel	groß	kein Mond
Erde	•	•	•				
Mars	•	•	•				
Merkur	×	×					×
Venus	×	×					×
Jupiter			×	×		×	
Saturn			×	×		×	
Uranus			×	×	×		
Neptun			×	×	×		
Pluto	×		×	×			

(b) Neu geordneter Kontext

Tabelle 2.1: Kontext $\mathcal{R}$ über Planeten unseres Sonnensystems

Tabelle 2.1 zeigt als Beispiel eine Kontexttabelle $\mathcal{R}$ über Planeten unseres Sonnensystems. Die Planeten sind die Objekte ($\mathcal{O}$) und Eigenschaften wie *groß* oder *nah* die Attribute ($\mathcal{A}$) des Kontextes. Das Kreuz zwischen *Venus* und *entfernt* repräsentiert das Element $(\mathit{Venus}, \mathit{entfernt}) \in \mathcal{R}$ der Relation $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ (des formalen Kontextes).

Betrachtet man eine Objektmenge $O \subseteq \mathcal{O}$ eines Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, so kann man nach ihren gemeinsamen Attributen fragen. Die gemeinsamen Attribute einer Menge O werden als O' definiert und analog dazu die gemeinsamen Objekte einer Menge A als A':

Definition 2 (Gemeinsame Objekte/Attribute) *Für eine Menge $A \subseteq \mathcal{A}$ eines Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ werden die gemeinsamen Objekte definiert als $A' = \{o \in \mathcal{O} \mid (o, a) \in \mathcal{R}$ für alle $a \in A\}$ und analog die gemeinsamen Attribute für eine Menge $O \subseteq \mathcal{O}$: $O' = \{a \in \mathcal{A} \mid (o, a) \in \mathcal{R}$ für alle $o \in \mathcal{O}\}$.*

Gemeinsame Attribute von $O = \{\mathit{Mars}, \mathit{Pluto}\}$ sind $O' = \{\mathit{klein}, \mathit{Mond}\}$. Für eine leere Objektmenge gilt $O' = \emptyset' = \mathcal{A}$ – alle Attribute sind einer leeren Objektmenge gemeinsam. Die zentrale Definition der formalen Begriffsanalyse ist die des (formalen) Begriffes:

Definition 3 (Begriff) *Ein Begriff eines Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ist ein Paar $(O, A) \in 2^{\mathcal{O}} \times 2^{\mathcal{A}}$ wobei gleichzeitig gilt: $O' = A$ und $A' = O$. Die Menge aller Begriffe eines Kontextes wird mit $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ bezeichnet.*

Ein Begriff ist ein spezielles Paar aus einer Objekt- und einer Attributmenge, das nur relativ zu einem Kontext definiert ist. Ein Begriff faßt zwei Mengen aus Objekten und Attributen eines Kontextes zusammen, die in diesem Kontext synonym sind: jede Menge bestimmt die andere eindeutig *und* umgekehrt.

In dem Beispiel bilden die Mengen $(\{Erde, Mars\}, \{klein, nah, Mond\})$ einen Begriff, weil $\{Erde, Mars\}' = \{klein, nah, Mond\}$ und $\{klein, nah, Mond\}' = \{Erde, Mars\}$ gilt. Begriffe formen in der Kontexttabelle maximale Rechtecke von Kreuzen, wenn man Zeilen und Spalten so vertauscht, daß die Elemente eines Begriffes nebeneinander zu stehen kommen. In Abbildung 2.1(b) sind Zeilen und Spalten für den beispielhaften Begriff entsprechend umsortiert.

Die Wahl der Bezeichnung *Begriff* für ein Paar (O, A) mit $A' = O$ und $O' = A$ entspricht durchaus dem natürlichen Sprachgebrauch, in dem ein Begriff ebenfalls durch Objekte mit gemeinsamen Eigenschaften bestimmt wird. Die Objektmenge eines Begriffes wird deshalb auch als *Umfang*, die Attributmenge als *Inhalt* des Begriffes bezeichnet. Im Gegensatz zur formalen Begriffsanalyse ist ein Kontext im natürlichen Sprachgebrauch meistens nicht explizit vorgegeben. Eine direkte Übertragung zwischen formalen und natürlichen Begriffen ist deshalb nur selten möglich.

Die Operation $'$ zur Bestimmung gemeinsamer Attribute und Objekte in einem Kontext besitzt eine Reihe von Eigenschaften ([32], Hilfsatz 10):

Theorem 1 *Ist $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Kontext und sind $O, O_1, O_2 \subseteq \mathcal{O}$ Mengen von Objekten und $A, A_1, A_2 \subseteq \mathcal{A}$ Mengen von Attributen, so gilt:*

1. $O_1 \subseteq O_2 \Rightarrow O_2' \subseteq O_1'$
2. $O \subseteq O''$
3. $O' = O'''$
4. $O \subseteq A' \iff A \subseteq O' \iff O \times A \subseteq \mathcal{R}$

Die analogen Eigenschaften gelten für Mengen von Attributen.

Wendet man den Operator $'$ zweimalig an, so erhält man einen Hüllenoperator: Die Abbildung $\varphi : \mathcal{X} \to \mathcal{X}, \varphi(X) = X''$ ist (1) monoton, (2) extensiv und (3) idempotent. Beweis: (1) aus $X_1 \subseteq X_2$ folgt $X_1' \supseteq X_2' \Rightarrow X_1'' \subseteq X_2'' = \varphi X_1 \subseteq \varphi X_2$. (2) gilt unmittelbar nach Theorem 1. (3) es gilt: $\varphi X = X'' = (X')' = (X')''' = X'''' = \varphi\varphi X$.

Für Beweise ist auch die folgende Eigenschaft des Operators $'$ nützlich. Sie erlaubt die Gemeinsamkeiten einer Vereinigung von Mengen auf die Gemeinsamkeiten der einzelnen Mengen zurückzuführen ([32], Hilfsatz 11):

Theorem 2 *Ist T eine Indexmenge und ist für jedes $t \in T$ $O_t \subseteq \mathcal{O}$ eine Menge von Objekten, so ist:*

$$(\bigcup_{t \in T} O_t)' = \bigcap_{t \in T} O_t'$$

Entsprechendes gilt für Mengen von Attributen.

Die Begriffe eines Kontextes sind (partiell) geordnet. Dies bedeutet, daß zwei Begriffe entweder in einer Ober-/Unterbegriffsbeziehung stehen, oder unvergleichlich sind.

Definition 4 *Sind $c_1 = (O_1, A_1)$ und $c_2 = (O_2, A_2)$ Begriffe eines Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, so heißt c_1 Unterbegriff von c_2 (und c_2 Oberbegriff von c_1), wenn $O_1 \subseteq O_2$ gilt. Dies wird in Zeichen als $c_1 \leq c_2$ geschrieben.*

Ein Unterbegriff umfaßt weniger Objekte als sein echter Oberbegriff – entsprechend dem natürlichen Gebrauch des Wortes ist ein Unterbegriff spezieller. Gleichzeitig besitzen die Objekte eines Unterbegriffs mehr gemeinsame Attribute als die seiner Oberbegriffe: aus $(O_1, A_1) \leq (O_2, A_2)$ folgt $A_1 \supseteq A_2$ wegen Theorem 1: $O_1 \subseteq O_2 \Rightarrow O_1' \supseteq O_2' = A_1 \supseteq A_2$. Ein Beispiel für die verschiedenen Beziehungen von Begriffen ist in Abbildung 2.1 zu sehen. Zwischen zwei Begriffen ist eine Linie gezogen, wenn sie in einer $\leq$-Beziehung stehen; dabei stehen Oberbegriffe über Unterbegriffen.

C_i	Erde	Mars	Pluto	klein	nah	Mond	entfernt
C_1	×	×	×	×		×	
C_2	×	×		×	×	×	
C_3			×	×		×	×

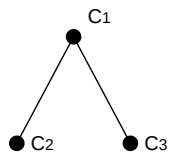

Abbildung 2.1: Ordnung von Begriffen

Zwei verschiedene Begriffe (O_1, A_1), (O_2, A_2) in der $\leq$-Relation unterscheiden sich sowohl in ihrer Objekt- als auch ihrer Attributmenge um mindestens ein Element, weil aus $O_1 \subset O_2$ folgt, daß $A_1 \supset A_2$ gilt. Wenn die beiden Begriffe nicht in der $\leq$-Relation enthalten sind, also unvergleichlich sind, so bedeutet dies *nicht*, daß ihre Objekt- und Attributmengen paarweise disjunkt sind: in Abbildung 2.1 sind c_2 und c_3 unvergleichlich, besitzen aber die gemeinsamen Attribute *klein* und *Mond*.

2.2 Begriffsverband

Jeder formale Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ enthält eine Menge $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ von formalen Begriffen. Das Bemerkenswerte an der Menge aller Begriffe ist, daß sie einen vollständigen Verband, den *Begriffsverband* bildet.

Definition 5 (Verband) *Eine geordnete Menge* $\mathbf{V} = (\mathcal{V}, \leq)$ *ist ein vollständiger Verband, falls zu jeder Teilmenge* $V \subseteq \mathcal{V}$ *das Supremum* $\bigvee V$ *und das Infimum* $\bigwedge V$ *existiert.*

Für jede Menge $C \subseteq B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ von Begriffen eines Kontextes existiert also ein eindeutiger kleinster Oberbegriff $\bigvee C$ und ein größter Unterbegriff $\bigwedge C$.

Theorem 3 (Hauptsatz, [32], Satz 3) *Die Menge* $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ *aller Begriffe eines Kontextes* $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ *ist ein vollständiger Verband, in dem Infimum und Supremum folgendermaßen beschrieben sind:*

$$\begin{aligned} \bigwedge_{t \in T} (O_t, A_t) &= \left(\bigcap O_t, \left(\bigcup A_t\right)''\right) \\ \bigvee_{t \in T} (O_t, A_t) &= \left(\left(\bigcup O_t\right)'', \bigcap A_t\right) \end{aligned}$$

Für das Beispiel der Planeten aus Tabelle 2.1 zeigt Tabelle 2.2 die Begriffe und Abbildung 2.2 das Hasse-Diagramm des Begriffsverbandes. Jeder Knoten des Graphen repräsentiert einen Begriff, jede Kante eine direkte $\leq$-Beziehung. Das Hasse-Diagramm zeigt zur besseren Übersicht nicht die Unter-/Oberbegriffsbeziehung zwischen Begriffen an, die sich durch die Transitivität von $\leq$ ergibt: ein Begriff ist nicht nur kleiner als direkt mit ihm verbundene Oberbegriffe, sondern kleiner als alle direkt *oder indirekt* erreichbaren Oberbegriffe.

n	Merkur	Venus	Erde	Mars	Jupiter	Saturn	Uranus	Neptun	Pluto	klein	mittel	groß	nah	entfernt	Mond	kein Mond
1										×	×	×	×	×	×	×
2									×	×				×	×	
3							×	×			×			×	×	
4	×	×								×			×			×
5					×	×						×		×	×	
6					×	×	×	×	×					×	×	
7			×	×						×			×		×	
8			×	×					×	×					×	
9	×	×	×	×						×			×			
10	×	×	×	×					×	×						
11			×	×	×	×	×	×	×						×	
12	×	×	×	×	×	×	×	×	×							

Tabelle 2.2: Begriffe des Beispiels

Die Operation $\wedge$ (*meet*) bestimmt für zwei Begriffe c_1, c_2 den größten Begriff c des Verbandes, für den sowohl $c \leq c_1$, als auch $c \leq c_2$ gilt. Im Hasse-Diagramm

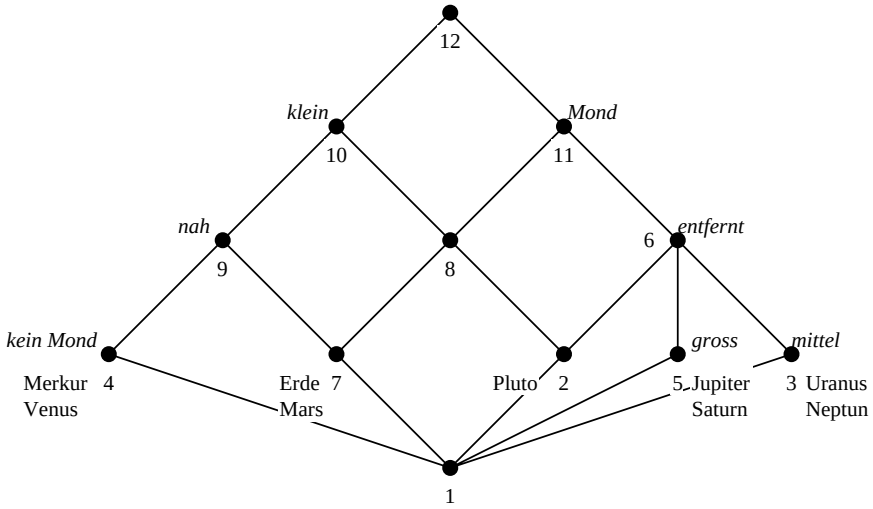

Abbildung 2.2: Hasse-Diagramm des Beispiels. Die Begriffe sind gemäß Theorem 4 markiert.

findet man diesen eindeutigen Begriff, indem man von c_1 und c_2 Kanten nach unten zum ersten gemeinsamen Unterbegriff c verfolgt. Analog erhält man das Supremum durch die Operation $\vee$ (*join*) und als den ersten gemeinsamen Oberbegriff von zwei Begriffen im Hasse-Diagramm.

In dem Verband in Abbildung 2.2 des Beispiels gelten zum Beispiel die folgenden Gleichungen:

$$\begin{array}{rcl@{\qquad}rcl} c_{10} \wedge c_{11} & = & c_8 & c_4 \vee c_7 \vee c_2 & = & c_{10} \\ c_7 \vee c_3 & = & c_{11} & c_9 \wedge c_6 & = & c_1 \end{array}$$

Alle Begriffe in $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ mit einem fest gewähltes Attribut $a \in \mathcal{A}$ in ihrem Inhalt besitzen einen kleinsten gemeinsamen Oberbegriff c. Es ist deshalb naheliegend, nur diesen Begriff c im Hasse-Diagramm mit a zu markieren, um so eine redundanzfreie und übersichtliche Beschriftung zu erhalten. Gleichzeitig existiert für jedes Objekt $o \in \mathcal{O}$ ein größter Begriff, in dessen Umfang o enthalten ist. Dieser Begriff wird im Hasse-Diagramm mit o markiert. Führt man die beschriebene Markierung für alle Objekte und Attribute des Beispiels durch, erhält man den markierten Begriffsverband in Abbildung 2.2. Ein Begriff kann eine oder keine Markierung tragen, oder auch mehrere.

In einem so markierten Begriffsverband lassen sich die Objekt- und Attributmengen jedes Begriffs rekonstruieren: alle Unterbegriffe eines mit a markierten Begriffes besitzen das Attribut a in ihrem Inhalt und alle Oberbegriffe eines mit einem Objekt o markierten Begriffes enthalten o in ihrem Umfang. Man kann sich vorstellen, daß im Hasse-Diagramm Objektmarkierungen an größere und Attributmarkierungen an kleinere Elemente vererbt werden.

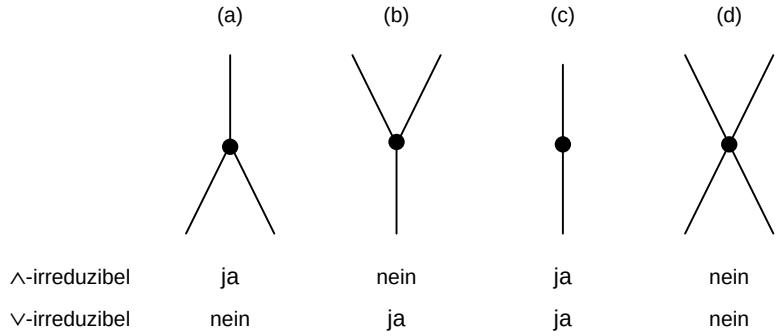

Abbildung 2.3: Skizze reduzibler und irreduzibeler Begriffe in endlichen Verbänden

Weil ein markierter Begriffsverband leichter zu zeichnen ist als ein Verband mit vollständig angetragenen Begriffen, ist die markierte Darstellung die Standarddarstellung in der Literatur. Formal werden Markierungen durch zwei Abbildungen γ und μ beschrieben. Der zu einem Objekt o gehörende Begriff $\gamma(o)$ wird *Objektbegriff* genannt, und analog $\mu(a)$ der *Attributbegriff* von a:

Theorem 4 *Sei $o \in \mathcal{O}$ und $a \in \mathcal{A}$ je ein fest gewähltes Objekt und Attribut in einem Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$. Bezeichne ferner $\gamma(o) = (O_1, A_1)$ den kleinsten Begriff des Kontextes mit $o \in O_1$ und $\mu(a) = (O_2, A_2)$ den größten Begriff mit $a \in A_2$. Dann gilt $\gamma(o) = (\{o\}'', \{o\}')$ und $\mu(a) = (\{a\}', \{a\}'')$ ([32], Satz 3).*

In einem Begriffsverband sind Infimum und Supremum einer Menge von Begriffen eindeutig. Umgekehrt kann man zwischen Begriffen unterscheiden, die sich als Infimum oder Supremum anderer Begriffe darstellen lassen und solchen, wo dies nicht möglich ist. Ein Begriff, der sich nicht als Infimum ($\wedge$) anderer Begriffe darstellen läßt, heißt $\wedge$-*irreduzibel*. Analog heißt ein Begriff $\vee$-*irreduzibel*, wenn er sich nicht als Supremum darstellen läßt. Alle anderen Begriffe heißen $\wedge$-, beziehungsweise $\vee$-reduzibel. Die $\wedge$- und $\vee$-Irreduzibilität eines Begriffes ist unabhängig von einander. Formal wird Irreduzibilität über das Infimum/Supremum aller Ober/Unterbegriffe eines Begriffes definiert, oder allgemein für Verbände:

Definition 6 (Irreduzibilität) *Für ein Element v eines vollständiges Verbandes $\mathcal{V}$ wird definiert:*

$$v^* = \bigwedge\{x \in \mathcal{V} \mid x > v\} \quad \text{und} \quad v_* = \bigvee\{x \in \mathcal{V} \mid x < v\}$$

Das Element v heißt $\wedge$-irreduzibel, wenn $v \neq v^*$ ist und $\vee$-irreduzibel, wenn $v \neq v_*$ ist.

In *endlichen* Verbänden sind die Elemente irreduzibel, die nur einen oberen beziehungsweise unteren Nachbarn haben:

	a_1	a_2	a_3	a_4
o_1	×			
o_2		×		
o_3	×			
o_4		×	×	
o_5		×		×

(a) Ausgangskontext

	a_1	a_2	a_3	a_4
o_1	×			
o_2		×		
o_4		×	×	
o_5		×		×

(b) Bereinigter Kontext

	a_1	a_2	a_3	a_4
o_1	×			
o_4		×	×	
o_5		×		×

(c) Reduzierter Kontext

Tabelle 2.3: Bereinigung und Reduzierung von Kontexten

Theorem 5 (Irreduzibilität in endlichen Verbänden) *Ein Element in einem endlichen Verbandes ist genau dann $\wedge$-irreduzibel, wenn es genau einen oberen Nachbarn hat und genau dann $\vee$-irreduzibel, wenn es genau einen unteren Nachbarn hat ([32], Hilfsatz 2).*

Irreduzibele Begriffe sind wichtig, weil sie die Struktur eines Begriffsverbandes bestimmen. Objekte und Attribute mit reduzibelen Objekt- oder Attributbegriffen können aus einem Kontext entfernt werden, ohne daß sich die Struktur des zugehörigen Begriffsverbandes verändert. Ausgehend von einem beliebigen Kontext kann dieser in zwei Schritten vereinfacht werden:

Definition 7 (Bereinigter Kontext) *Ein Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ heißt bereinigt, wenn für beliebige Objekte $o_1, o_2 \in \mathcal{O}$ aus $\{o_1\}' = \{o_2\}'$ stets $o_1 = o_2$ folgt und außerdem $\{a_1\}' = \{a_2\}' \Rightarrow a_1 = a_2$ für alle $a_1, a_2 \in \mathcal{A}$ gilt.*

Der Ausdruck $\{o\}'$ für $o \in \mathcal{O}$ ist die Menge aller mit o in Relation stehenden Elemente und anschaulich die Zeile o der Kontexttabelle. In einem bereinigten Kontext ist jede Zeile und Spalte eindeutig. Ein Kontext wird demnach bereinigt, indem Spalten entfernt werden, die zu einer anderen Spalte identisch sind; gleiches gilt für Zeilen. Die Struktur des Begriffsverbandes des bereinigten Kontextes entspricht der des unbereinigten Kontextes. Kein Begriff des bereinigten Kontextes besitzt mehr als ein Objekt und ein Attribut als Markierung nach Theorem 4. Der Effekt des Bereinigens kann in Abbildung 2.3 beim Übergang vom linken zum mittleren Kontext beobachtet werden.

Definition 8 (Reduzierter Kontext) *Ein bereinigter Kontext heißt reduziert, wenn jeder Objektbegriff $\vee$-irreduzibel und jeder Attributbegriff $\wedge$-irreduzibel ist.*

Um einen bereinigten Kontext zu reduzieren, berechnet man zunächst seinen Begriffsverband. Danach entfernt man alle Objekte und Attribute aus dem Kontext, die in der Standarddarstellung des Begriffsverbandes Markierungen von

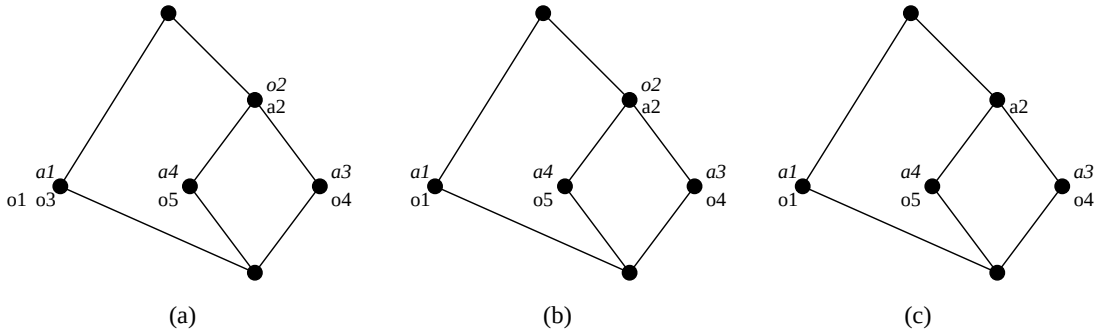

Abbildung 2.4: Begriffsverbände zu den Kontexten in Tabelle 2.3. Bereinigung und Reduktion von Kontexten verändert die Struktur ihrer Begriffsverbände nicht.

reduzibelen Begriffen sind. Formal muß $o \in \mathcal{O}$ entfernt werden, wenn $\gamma(o)$ $\vee$-reduzibel ist und $a \in \mathcal{A}$, wenn $\mu(a)$ $\wedge$-reduzibel ist. Der Einfluß dieser Operation kann wieder in Abbildung 2.3 beim Übergang vom mittleren zum rechten Kontext beobachtet werden: es werden die Zeilen und Spalten entfernt, die sich als Schnittmenge anderer Zeilen und Spalten darstellen lassen.

Die Gewißheit, daß sowohl beim Bereinigen als auch Reduzieren eines Kontextes sich die Struktur des zugehörigen Begriffsverbandes nicht verändert, kommt aus Hilfsatz 12 in [32]. Er besagt nämlich, daß bis auf Isomorphie zu jedem endlichen Verband genau ein reduzierter Kontext existiert.

2.3 Äquivalente Darstellung von Kontexten

Jeder Begriffsverband $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ enthält noch alle Informationen seines Kontextes, so daß sich dieser aus dem Verband rekonstruieren läßt. Die Mengen O und A sind die Objekt- und Attributmengen des größten ($\top$) und kleinsten ($\bot$) Begriffes: $\top = (\emptyset', \emptyset'') = (O, O'')$ und $\bot = (\emptyset'', \emptyset') = (A'', A)$. Alle Objekte und Attribute eines Begriffes (O, A) stehen miteinander in der Relation $\mathcal{R}$, so daß sich ergibt: $\mathcal{R} = \bigcup\{O \times A \mid (O, A) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})\}$. Ein Objekt o und Attribut a stehen genau in der Relation $\mathcal{R}$, wenn der Objektbegriff $\gamma(o)$ größer als der Attributbegriff $\mu(a)$ ist: $(o, a) \in \mathcal{R} \iff \gamma(o) \leq \mu(a)$. ([32], Abschnitt 1.2).

Außer als Begriffsverband können die Zusammenhänge in einem Kontext als logische Implikationen dargestellt werden, die allerdings in dieser Arbeit nicht weiter verwendet werden. Eine Implikation $A_1 \longrightarrow A_2$ beschreibt, daß jedes Objekt mit den Attributen A_1 ebenfalls die Attribute A_2 besitzt.

Definition 9 *Eine Implikation $A_1 \longrightarrow A_2$ mit $A_1, A_2 \subseteq \mathcal{A}$ gilt in einem Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, wenn für alle $o \in \mathcal{O}$ gilt: $\{o\}' \not\subseteq A_1$ oder $\{o\}' \subseteq A_2$.*

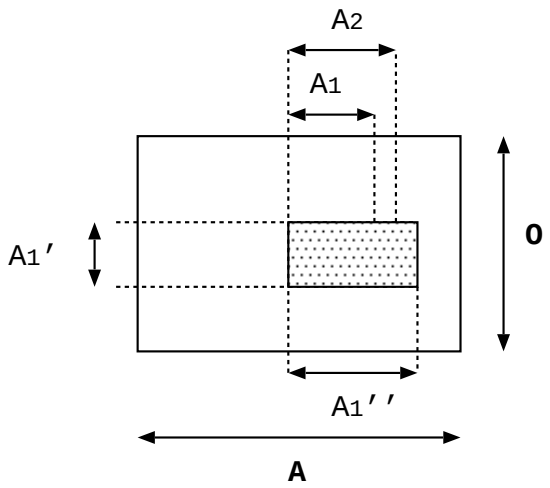

Abbildung 2.5: Skizze zu Theorem 6

Die Gültigkeit einer Implikation kann leicht mit dem folgenden Theorem (aus [32]) überprüft werden; Abbildung 2.3 skizziert nochmals die Zusammenhänge.

Theorem 6 *Eine Implikation* $A_1 \longrightarrow A_2$ *gilt in* $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ *genau dann, wenn* $A_2 \subseteq A_1''$.

Eine Menge $\mathcal{L}$ von Implikationen und eine Menge von Attributen $\mathcal{A}$ bestimmen eine Menge $H(\mathcal{L}) = \{A \subseteq \mathcal{A} \mid A \text{ respektiert } \mathcal{L}\}$ von Attributen, die mit der Menge von Implikationen verträglich ist. Diese Attributmengen bilden über Mengeninklusion einen Verband. Wenn $\mathcal{L}$ gerade alle in einem Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ gültigen Implikationen enthält, dann ist $H(\mathcal{L})$ wieder isomorph zu $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$.

Zu einem Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ existieren in der Regel verschiedene Mengen von Implikationen, die darin gültig sind. Im Fall von endlichen Attributmengen haben Guigues und Duquenne gezeigt, daß für jeden Kontext eine ausgezeichnete minimale Menge von Implikationen existiert, die Duquenne-Guigues-Basis ([32], Satz 8). Wie der Begriffsverband ist sie eine äquivalente Beschreibung seines Kontextes.

Da diese Arbeit keine Implikationen zwischen Merkmalen verwendet, werden sie hier nicht genauer eingeführt. Ihr Hauptanwendung ist die sogenannte Merkmalsexploration, bei der durch eine wachsende Menge von Implikationen interaktiv mit einem Programm ein Kontext konstruiert wird. Der Kontext faßt dann Eigenschaften von Objekten zusammen, die vorher nicht explizit, sondern nur in Form von Implikationen bekannt waren. Die weiterführenden Definitionen, Sätze und Beispiele sind bei Ganter und Wille [32] zu finden.

Kapitel 3

Technische Aspekte der Begriffsanalyse

Der vorherige Abschnitt hat formale Begriffe zusammen mit ihren wichtigsten Eigenschaften eingeführt. Jeder formale Kontext, also jede binäre Relation, besitzt eine Menge von Begriffen, die einen vollständigen Verband bilden. Dieser Abschnitt stellt Datenstrukturen und Algorithmen vor, um die Begriffe eines Kontextes und ihren Verband zu bestimmen. Sie sind die Voraussetzungen, um Begriffsanalyse (im Software-Engineering) praktisch anwenden zu können.

3.1 Die Begriffe eines Kontextes

Jeder formale Kontext besitzt eine nicht leere Menge von Begriffen. Ihre maximale Anzahl kann exponentiell mit der Größe des Kontextes wachsen: ein Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ mit $|\mathcal{O}| = m$ und $|\mathcal{A}| = n$ besitzt maximal 2^l Begriffe, wobei $l = \min(m, n)$. Die kleinere der beiden Kardinalitäten von $|\mathcal{O}|$ und $|\mathcal{A}|$ beschränkt die Zahl der Begriffe, weil jede Objekt- und Attributmenge in einem Begriffsverband eindeutig ist und eine Menge mit l Elementen maximal 2^l Teilmengen besitzt.

Ein Kontext $(\mathcal{X}, \mathcal{X}, \mathcal{R})$ mit einer maximalen Anzahl von Begriffen besitzt eine quadratische Kontexttabelle (Abbildung 3.1), die bis auf die Hauptdiagonale belegt ist: Sei $\mathcal{X} = \{1, \ldots, n\}$ und $\mathcal{R} = \{(i, j) \mid i, j \in \{1, \ldots, n\}, i \neq j\}$. Dieser Kontext enthält 2^n Begriffe. Oder genauer: wenn $X \subseteq \mathcal{X}$ eine beliebige Menge ist, dann ist $(X, \overline{X})$ ein Begriff von $(\mathcal{X}, \mathcal{X}, \mathcal{R})$ mit $\overline{X} = \{x \in \mathcal{X} \mid x \notin X\}$.

Für eine beliebige Objektmenge $X \subseteq \mathcal{X}$ enthält X' alle Attribute $x \in \mathcal{X}$, die den Elementen in X gemeinsam sind. Da die Konfigurationstabelle bis auf die Hauptdiagonale vollständig ausgefüllt ist, sind alle Elemente aus $\mathcal{X}$ auch Elemente von X' – mit Ausnahme von $x \in X$ mit $(x, x) \notin \mathcal{R}$. Also ist das Komplement $\overline{X}$ von X gleich X'. Wegen des symmetrischen Aufbaus der Relation kann man schließen, daß auch $(\overline{X})' = X$ gilt und Begriffe die beschriebene Form besitzen.

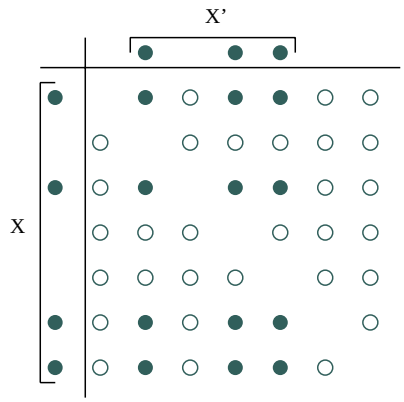

Abbildung 3.1: Beispiel für einen Kontext mit maximaler Anzahl 2^n ($n = 7$) von Begriffen.

3.1.1 Der naive Algorithmus

Eine naive Methode zur Berechnung aller Begriffe eines Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ kann Theorem 1 verwenden: jedes Paar (O'', O') mit $O \subseteq \mathcal{O}$ ist ein Begriff, weil $(O'')' = O'$ gilt. Analog ist auch jedes Paar (A', A'') mit $A \subseteq \mathcal{A}$ ein Begriff. Wenn $|\mathcal{A}| \leq |\mathcal{O}|$ gilt, dann ist $\{(A', A'') \mid A \subseteq \mathcal{A}\} = B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ die Menge aller Begriffe und der naive Algorithmus berechnet $2^{|A|}$ Paare. Der Nachteil dieser Methode ist offensichtlich: Unabhängig von der Anzahl der Begriffe in $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ sind $O(2^n)$, mit $n = \min(|\mathcal{O}|, |\mathcal{A}|)$, Schritte nötig, um sie zu bestimmen. Die Operationen $'$ und $''$ besitzen die Zeitkomplexität $O(|\mathcal{O}| \times |\mathcal{A}|)$; der naive Algorithmus ist deswegen von der Gesamtkomplexität $O(2^n \times |\mathcal{O}| \times |\mathcal{A}|)$ mit $n = \min(|\mathcal{O}|, |\mathcal{A}|)$ und damit für realistische Anwendungen ungeeignet.

3.1.2 Ganters Algorithmus

Der Algorithmus von Ganter [29, 32] berechnet die Menge aller Begriffe eines Kontextes mit einer wesentlich besseren Zeitkomplexität $O(|\mathcal{O}|^2 \times |\mathcal{A}| \times |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|)$ als der naive Algorithmus. Dazu wird eine totale Ordnung auf Begriffen definiert und ausgehend von dem kleinsten Begriff in dieser Ordnung der jeweilige Nachfolger bestimmt. Jeder einzelne Schritt generiert aus dem zuletzt gefundenen Begriff bis zu $|\mathcal{O}|$ Kandidaten. Der nächste Begriff wird also nach höchstens $|\mathcal{O}|$ Schritten gefunden oder es ist sicher, daß keine weiteren Begriffe existieren. Deshalb ist die Komplexität des Algorithmus durch die *tatsächliche* Anzahl der Begriffe eines Verbandes und $|\mathcal{O}|$ beschränkt.

Ganter definiert eine totale *lektische*[1] Ordnung $\prec$ auf Mengen, die auf einer totalen Ordnung der Grundmengen $\mathcal{O}$ und $\mathcal{A}$ aufbaut: $\mathcal{O} = \{o_1, \ldots, o_{\mathcal{O}}\}$ mit $o_1 \prec o_2 \prec \ldots$, und analog für $\mathcal{A}$.

[1] oder auch *lexikalische*

Definition 10 (Lektische Ordnung) *Von zwei verschiedenen Mengen $O_1, O_2 \subseteq \mathcal{O}$ heißt O_1* lektisch kleiner *genau dann, wenn das kleinste Element, in dem sich O_1 und O_2 unterscheiden in O_2 enthalten ist. Formal:*

$$\begin{aligned} O_1 \prec O_2 &\iff O_1 \prec_i O_2 \\ &\iff \exists o_i \in O_2 \setminus O_1 : O_1 \cap \{o_1, \ldots, o_{i-1}\} = O_2 \cap \{o_1, \ldots, o_{i-1}\} \end{aligned}$$

Die Relation $\prec_i$ beschränkt den Vergleich auf o_j mit $i < j$. Die Definition gilt analog für Attributmengen $A_1, A_2 \subseteq \mathcal{A}$.

Die lektische Ordnung ($\prec$) kann auf Begriffen fortgesetzt werden. Zwei verschiedene Begriffe sind im Gegensatz zur Halbordnung ($\leq$) immer durch $\prec$ vergleichbar:

Definition 11 *Von zwei verschiedene Begriffen $c_1 = (O_1, A_1), c_2 = (O_2, A_2) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ heißt c_1 lektisch kleiner als c_2 genau dann, wenn $O_1 \prec O_2$ gilt.*

Zwischen der Halbordnung ($\leq$) und der lektischen Ordnung ($\prec$) von Begriffen besteht ein Zusammenhang: ein Unterbegriff ist lektisch kleiner als sein Oberbegriff. Der lektisch kleinere von zwei Begriffen ist aber nicht zwingend ein Unterbegriff des anderen – die Umkehrung gilt also nicht.

Theorem 7 *Seien $c_1 = (O_1, A_1), c_2 = (O_2, A_2) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ verschiedene Begriffe. Dann gilt: $c_1 < c_2 \Rightarrow c_1 \prec c_2$.*

Beweis: Es ist zu zeigen, daß $O_1 \prec O_2$ gilt. Aus $c_1 < c_2$ folgt $O_1 \subset O_2$. Weil O_1 eine echte Untermenge von O_2 ist, gilt $O_2 \setminus O_1 \neq \emptyset$. Sei o_i das kleinste Element der Menge $O_2 \setminus O_1$, dann gilt: $O_1 \cap \{o_1, \ldots o_{i-1}\} = O_2 \cap \{o_1, \ldots o_{i-1}\}$ und damit $O_1 \prec O_2$.

Ausgehend von einer Objektmenge O eines Begriffes (O, A) definiert Ganter eine Operation $\oplus$, die eine neue Objektmenge bestimmt: $O \oplus o_i = ((O \cap \{o_1, \ldots, o_{i-1}\}) \cup \{o_i\})''$. Mit Hilfe von $\oplus$ läßt sich zu einem Begriff der lektisch nächste Begriff bestimmen:

Theorem 8 *Die kleinste lektische Objektmenge, die bezüglich der lektischen Ordnung größer ist als eine gegebene Menge $O \subseteq \mathcal{O}$, ist $O \oplus o_i$, wobei o_i das größte Element von $\mathcal{O}$ ist mit $O \prec_i O \oplus o_i$. ([32], Satz 5)*

Dieser Satz ist der Kern von Ganters Algorithmus: die lektisch kleinste Objektmenge ist $\emptyset''$, der lektisch kleinste Begriff also $(\emptyset'', \emptyset')$. Die Objektmenge des nächsten Begriffes findet man, indem absteigend von dem größten Element alle Elemente $o_i \in \mathcal{O}$ geprüft werden, bis erstmalig $O \prec_i O \oplus o_i$ erfüllt ist. Dann ist $O \oplus o_i$ die Objektmenge des lektisch nächsten Begriffes. Die Komplexität dieses

	$\mathcal{A}$						
$\mathcal{O}$	1	2	3	4	5	6	7
1				×		×	×
2		×	×			×	
3				×		×	×
4	×			×			×
5		×			×	×	

(a) Kontext

			Begriff	
n	i	O_+	$O'_+ = A$	O''_+
1		$\emptyset$	1 2 3 4 5 6 7	$\emptyset$
2	5	5	2 5 6	5
3	4	4	1 4 7	4
	5	4 5	$\emptyset$	*1 2 3* 4 5
	3	3	4 6 7	*1* 3
4	2	2	2 3 6	2
5	5	2 5	2 6	2 5
	4	2 4	$\emptyset$	*1* 2 *3* 4 5
	3	2 3	6	*1* 2 3 5
6	1	1	4 6 7	1 3
	5	1 3 5	6	1 *2* 3 5
7	4	1 3 4	4 7	1 3 4
	5	1 3 4 5	$\emptyset$	1 *2* 3 4 5
8	2	1 2	6	1 2 3 5
9	4	1 2 3 4	$\emptyset$	1 2 3 4 5

(b) Begriffe

Tabelle 3.1: Beispiel für Ganters Algorithmus zur Berechnung aller Begriffe eines Kontextes

Algorithmus zur Bestimmung des nächsten Begriffes ist $O(|\mathcal{O}|^2 \times |\mathcal{A}|)$: die Operation $''$ mit ihrer Komplexität $O(|\mathcal{O}| \times |\mathcal{A}|)$ muß maximal $|\mathcal{O}|$-malig ausgeführt werden. Dies kann besser als an Theorem 8 an der algorithmischen Darstellung im noch folgenden Abschnitt 3.2 (Seite 21) abgelesen werden.

Tabelle 3.1 zeigt auf der linken Seite einen Kontext, dessen Begriffe mit Ganters Algorithmus bestimmt werden sollen. In der Tabelle auf der rechten Seite werden alle Zwischenergebnisse und die Begriffe $n = 1, \ldots, 9$ des Kontextes aufgeführt. Die Menge $O_+ = (O \cap \{o_1, \ldots, o_{i-1}\}) \cup \{o_i\}$ ist die Objektmenge des zuletzt ermittelten Begriffes, beschränkt auf Elemente kleiner als o_i, und um o_i erweitert. Aus dieser Menge wird ein Kandidat für den nächsten Begriff bestimmt: die Attributmenge dieses Begriffes ist O'_+, die zugehörige Objektmenge $O''_+ = O \oplus o_i$. Die Objektmenge des nächsten Begriffes muß $O \prec_i O \oplus o_i$ erfüllen: gegenüber O darf die neue Objektmenge $O \oplus o_i$ keine neuen Elemente enthalten, die kleiner als o_i sind. Falls die neue Objektmenge dieses Kriterium verletzt, sind die störenden Elemente *kursiv* hervorgehoben. Dann wird der Kandidat verworfen und das nächstkleinere Element o_{i-1} in einem nächsten Test verwendet. Immer wenn erfolgreich der lektisch nächste Begriff gefunden wird, enthält die Tabelle eine horizontale Linie. Die Konstruktion von O_+ läßt die Elemente o_i aus, die bereits in der Objektmenge des vorhergehenden Begriffes enthalten sind. Dies garantiert, daß der nächste Begriff sich von dem vorherigen unterscheidet und ist

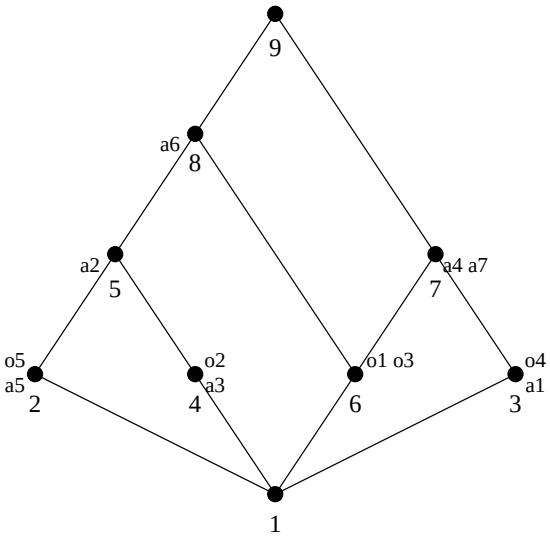

Abbildung 3.2: Begriffsverband des Beispiels. Die Numerierung unter den Begriffen gibt die Reihenfolge ihrer Berechnung und ihre lektische Ordnung an. Unterbegriffe werden vor ihren Oberbegriffen berechnet: sie sind lektisch kleiner als ihre Oberbegriffe.

der Grund, warum die Folge i bei manchen Begriffen ($n = 6$) Lücken aufweist: $i = (5), 4, 3, (2), 1$ – die Elemente 5 und 2 sind bereits in der Objektmenge der vorhergehenden Begriffes enthalten.

Jeder Kandidat für den nächsten Begriff in den beiden rechten Spalten der Tabelle ist per Konstruktion ein Begriff des Kontextes. Allerdings ist er oft nicht der gesuchte lektische Nachfolger, und wird deswegen wieder verworfen. Zum Beispiel der Begriff mit der Objektmenge $\{1, 2, 3, 5\}$ – er wird zweimal verworfen, bevor er als Begriff $n = 8$ festgehalten wird.

Wenn die Begriffe im Begriffsverband in Abbildung 3.2 in der Reihenfolge ihrer Berechnung durch Ganters Algorithmus numeriert werden, läßt sich die Bedeutung von Theorem 7 ablesen: vor einen Begriff bestimmt der Algorithmus erst dessen Unterbegriffe. Dies bedeutet, daß in Tabelle 3.1 die Unterbegriffe eines Begriffes vor ihm selbst in der Tabelle stehen.

3.1.3 Inkrementelle Berechnung aller Begriffe

Wenn sich ein Kontext durch die Ergänzung eines bestehenden Kontextes ergibt, so kann unter Umständen auch der Begriffsverband des neuen Kontextes unter Verwendung des Verbandes des ursprünglichen Verbandes berechnet werden. Der neue Kontext und sein Begriffsverband entstehen also durch eine Ergänzung eines bestehenden Paares aus Kontext und Verband. Gegenüber einer vollständigen Neuberechnung des Verbandes (durch Ganters Algorithmus) ergibt sich eine

Komplexitätsreduktion. Der nachfolgend vorgestellte Algorithmus bietet sich für Anwendungen an, in denen ein Kontext gelegentlich um wenige Elemente ergänzt werden muß. Dies trifft zum Beispiel auf die in Abschnitt 5 vorgeschlagene Organisation von Softwarekomponenten mittels formaler Begriffe zu.

Ganter und Kuznetsov beschreiben den Algorithmus zur inkrementellen Berechnung aller Begriffe in [31]; die dort angegebene Fassung bezieht sich allgemein auf geordnete Mengen und nicht nur auf Begriffsverbände. Ausgehend von einem Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ und seinem Begriffsverband $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ soll der Begriffsverband eines neuen Kontextes $(\mathcal{O} \cup \{o\}, \mathcal{A} \cup \{a\}, \mathcal{R}^+)$ berechnet werden. Gegenüber dem bestehenden Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ sind in dem neuen Kontext ein Objekt o, ein Attribut a und Beziehungen hinzugekommen. Die Beziehung zwischen diesen Elementen muß gewissen Bedingungen genügen: $\mathcal{R}^+ = \mathcal{R} \cup (O \cup \{o\}) \times (A \cup \{a\})$, wobei (O, A) ein *Vorbegriff* in $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ist.

Definition 12 (Vorbegriff) *Sei $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Kontext. Dann ist $(O, A) \in 2^{\mathcal{O}} \times 2^{\mathcal{A}}$ ein* Vorbegriff *genau dann, wenn $O \subseteq A'$ und $A \subseteq O'$ gilt.*

Der Vorbegriff (O, A) definiert die Beziehung der neuen Elemente o und a zu den bestehenden Objekten und Attributen. Leider hat dies auch Konsequenzen für die Beziehung zwischen Objekten und Attributen aus $\mathcal{O}$ und $\mathcal{A}$, da $\mathcal{R}$ um ein kartesisches Produkt von O und A erweitert wird: die neue Relation $\mathcal{R}^+$ enthält nicht nur Einträge für o und a, sondern möglicherweise auch neue Einträge für Elemente aus $\mathcal{O}$ und $\mathcal{A}$. Die mittlere Skizze in Abbildung 3.3 verdeutlicht die Konstruktion von $\mathcal{R}^+$.

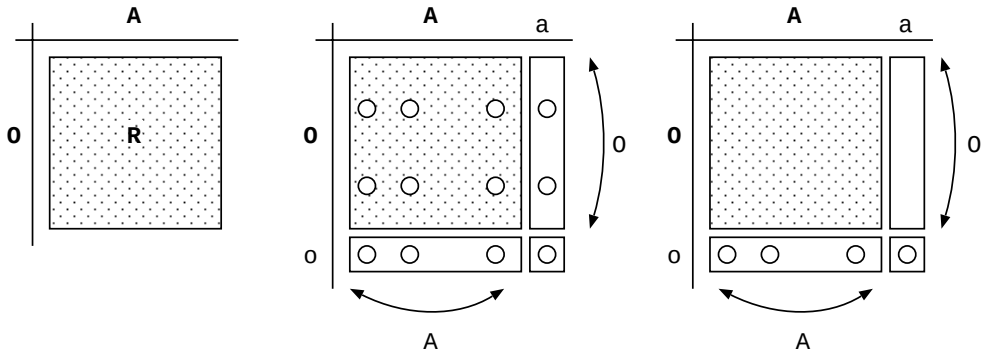

Abbildung 3.3: Inkrementelle Ergänzung von Kontexten durch Vorbegriffe

Eine mögliche Lösung besteht darin, ein neues Objekt o nur mit den Attributen $A \cup \{a\}$ in Relation zu setzen, das ebenfalls neue Attribut a aber mit keinem der alten Objekte zu verbinden. Diese Situation ist in der rechten Skizze in Abbildung 3.3 dargestellt. Das dazugehörende Paar $(O, A) = (\emptyset, A)$ ist immer ein Vorbegriff in $(\mathcal{O}, \mathcal{A}, \mathcal{R})$. Leider setzt die Formulierung des Algorithmus von Kanter und Kuznetsov in [31] die gleichzeitige Einführung zweier neuer Elemente

o und a voraus. Es ist also unmöglich, eine neues Objekt o einzuführen, das nur zu Attributen in $\mathcal{A}$ in Relation steht.

Der Algorithmus zur Berechnung aller Begriffe von $(\mathcal{O} \cup \{o\}, \mathcal{A} \cup \{a\}, \mathcal{R}^+)$ mit Vorbegriff (O, A) betrachtet jeden Begriff in $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ und ersetzt oder ergänzt ihn:

1. $(O \cup \{o\}, A \cup \{a\})$ ist ein Begriff.
2. Betrachte jeden Begriff $(O_1, A_1) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$:
   (a) Wenn $O_1 \subseteq O$ und $A_1 \not\subseteq A$, dann ist $(O_1, A_1 \cup \{a\})$ ein Begriff.
   (b) Wenn $O_1 \not\subseteq O$ und $A_1 \subseteq A$, dann ist $(O_1 \cup \{o\}, A_1)$ ein Begriff.
   (c) Wenn $O_1 \not\subseteq O$ und $A_1 \not\subseteq A$, dann:
      i. (O_1, A_1) ist ein Begriff.
      ii. Wenn $O_1 = (A_1 \cap A)'$, dann ist $(O_1 \cup \{o\}, A_1 \cap A)$ ein Begriff.
      iii. Wenn $A_1 = (O_1 \cap O)'$, dann ist $(O_1 \cap O, A_1 \cup \{a\})$ ein Begriff.

Die Komplexität des Algorithmus wird durch die aufwendigste Operation $'$ mit ihrer Komplexität $O(|\mathcal{O}| \times |\mathcal{A}|)$ bestimmt: die Gesamtkomplexität ist $O(n \times |\mathcal{O}| \times |\mathcal{A}|)$ mit $n = |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$. Eine genauere Abschätzung mit Hilfe des Durchmessers des Verbandes $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ist in [31] angegeben. Im Vergleich dazu erfordert die vollständige Neuberechnung der Menge aller Begriffe durch Ganters Algorithmus einen Aufwand von $O(n \times |\mathcal{O}|^2 \times |\mathcal{A}|)$.

3.2 Implementierung von Ganters Algorithmus

Für praktische Anwendungen ist die Implementierung eines Algorithmus zur Bestimmung aller Begriffe und des Begriffsverbandes unerläßlich. Ganter selbst hat in [29] seinen Algorithmus in einer Pascal-Notation angegeben. In der Darmstädter Gruppe von Wille und Ganter hat Frank Vogt eine Bibliothek für formale Begriffsanalyse in C++ [106, 112] entwickelt, die der kommerziellen begrifflichen Anwendung *Toscana* [111] zu Grunde liegt. In Toscana und der C++ Bibliothek liegt der Schwerpunkt auf der Verwendung mehrwertiger und skalierter Kontexte [32], die für Anwendungen im Software Engineering nicht untersucht wurden. Nachfolgend wird eine abstrakte Implementierung von Ganters Algorithmus für binäre Kontexte vorgestellt. Der sich anschließende Abschnitt untersucht diese Implementierung empirisch und leitet aus den Ergebnissen Hinweise für eine konkrete Implementierung ab.

Zur Notation der Algorithmen wird die funktionale Programmiersprache *Objective Caml* [58, 60, 59] verwendet. Sie ist ein Dialekt der Programmiersprache ML, deren bekanntester Vertreter Standard-ML [75] ist. Eine Notation in Pseudo-Code als Alternative dazu würde immer die Gefahr der Mehrdeutigkeit, Inkonsistenz und auch von Fehlern bergen. Die hier gezeigten Algorithmen in Objective

Funktion	Typ	Erläuterung
`make`	$int \rightarrow bitset$	erzeugt eine leere Menge für n Elemente
`add`	$int \rightarrow bitset \rightarrow unit$	fügt das Element i zu einer Menge per Seiteneffekt hinzu. Die Funktion gibt keinen Wert (*unit*) zurück
`remove`	$int \rightarrow bitset \rightarrow unit$	entfernt das Element i aus einer Menge per Seiteneffekt
`isMember`	$int \rightarrow bitset \rightarrow bool$	*true* genau dann, wenn i Element der Menge ist
`copy`	$bitset \rightarrow bitset$	identische Kopie einer Menge
`asList`	$bitset \rightarrow int\ list$	die Liste aller Elemente der Menge
`isSmaller`	$bitset \rightarrow bitset \rightarrow bool$	implementiert $x \prec y$
`isRestrictedSubset`	$bitset \rightarrow bitset \rightarrow int \rightarrow bool$	implementiert $x \prec_i y$
`intersect`	$bitset \rightarrow bitset \rightarrow unit$	der Schnitt von zwei Mengen: $A := A \cap B$; das erste Argument enthält das Ergebnis
`difference`	$bitset \rightarrow bitset \rightarrow unit$	die Differenz von zwei Mengen: $A := A \setminus B$; das erste Argument enthält das Ergebnis.

Tabelle 3.2: Signatur von *Bitset*

Caml stammen aus einer existierenden und vollständigen Implementierung der Begriffsanalyse. Die Programmiersprache Objective Caml ist polymorph, streng typisiert und besitzt algebraische Datentypen. Neben rein applikativen Sprachelementen stellt sie auch imperative Konstrukte zur Verfügung, die die effiziente Implementierung von Datenstrukturen erleichtern.

Ganters Algorithmus setzt geordnete Mengen von Objekten und Attributen voraus. Deshalb ist eine Mengenabstraktion die fundamentale Datenstruktur für eine Implementierung seines Algorithmus. Das Modul *Bitset* implementiert endliche Mengen von natürlichen Zahlen. Diese Zahlen lassen sich als die Indizes i von Objekten o_i oder Attributen a_i auffassen. Tabelle 3.2 zeigt die Signatur des abstrakten Mengentyps *bitset*, den das Modul *Bitset* implementiert und erläutert die Semantik der einzelnen Funktionen. Gegenüber der realen Implementierung sind Funktionen weggelassen, die zur Vollständigkeit einer Mengenabstraktion beitragen, für die Implementierung von Ganters Algorithmus aber überflüssig sind.

Funktion	Typ	Erläuterung
`make`	$int \to int \to (int \times int)\ list \to relation$	erzeugt eine Relation der Größe $n_x \times n_y$ und fügt die Elemente aus der übergebenen Liste von (x, y) Paaren darin ein.
`xSize`	$relation \to int$	die Größe der x-Dimension der Relation wie sie bei *make* als erster Parameter übergeben wurde.
`commonX`	$int\ list \to relation \to bitset$	bestimmt eine Menge der Größe n_x die alle x-Elemente enthält, die den in einer Liste übergebenen y-Elementen in der Relation gemeinsam ist. Implementiert Y'.
`commonY`	$int\ list \to relation \to bitset$	bestimmt eine Menge der Größe n_y die alle y-Elemente enthält, die den in einer Liste übergebenen x-Elementen in der Relation gemeinsam ist. Implementiert X'.

Tabelle 3.3: Signatur von *Bitrelation*

Funktionen mit einem Ergebnistyp *unit* verändern ihre Argumente durch einen Seiteneffekt (call by reference) und geben unechte Werte zurück; *unit* ist der Typ dieser unechten Werte und dient der Kennzeichnung von Seiteneffekten. Alle anderen Funktionen sind applikativ, das heißt, sie verändern ihre Argumente nicht. Prinzipiell kann auch *Bitset* rein applikativ implementiert werden; dies erschwert aber eine effiziente Implementierung dieses Basisdatentyps.

Die Signatur von *Bitset* enthält die bekannten Funktionen für eine Mengenabstraktion, mit Ausnahme von *isRestrictedSubset* zur Implementierung von $\prec_i$. Die Entscheidung, bei der Erzeugung von Mengen durch *make* eine feste Größe vorzugeben, ist durch Effizienzerwägungen motiviert. Dynamisch wachsende und insbesonders schrumpfende Mengen sind schwieriger effizient zu implementieren. Eine Menge der Größe n kann die Elemente $0, \ldots, n-1$ aufnehmen.

Neben Mengen sind binäre Relationen für die Implementierung formaler Begriffsanalyse zentral. Sie werden durch Mengen implementiert und folglich hängt die Signatur *Bitrelation* ihrer Implementierung von der Signatur *Bitset* ab. Eine Relation ist abstrakt eine Menge von Paaren. In der Implementierung wird die erste Komponente eines Paares als x bezeichnet, die zweite als y. Bislang ist die erste Komponente mit einem Objekt und die zweite mit einem Attribut assoziiert worden. Die Signatur *Bitrelation* des Moduls zeigt Tabelle 3.3.

Der Algorithmus für alle Begriffe eines Kontextes wird in dem Modul *Concept* implementiert. Der kleinste Begriff, dessen Objektmenge eine gegebene Objektmenge O vollständig enthält, ist (O'', O'). Diese Operation wird durch die Funktion `conceptFromX` mit dem Typ $\mathit{bitset} \rightarrow \mathit{relation} \rightarrow \mathit{bitset} \times \mathit{bitset}$ implementiert; ihre asymptotische Komplexität ist $O(|\mathcal{O}| \times |\mathcal{A}|)$ und ihr Ergebnis ist ein Begriff, implementiert als ein Paar von Mengen. Es folgt der kurze Quellcode der Funktion in Objective Caml; Werte und Funktionen werden durch `let ... in` an Bezeichner gebunden. Die Argumente eines Funktionsaufrufes sind ungeklammert: $f\ x_1\ x_2$ statt $f(x_1\ x_2)$ wie in Pascal oder C.

```
let conceptFromX xSet relation =
  let commonY = Bitrelation.commonY
                (Bitset.asList xSet) relation in
  let commonX = Bitrelation.commonX
                (Bitset.asList commonY) relation in
    (commonX,commonY)
```

Der lektisch kleinste Begriff ist $(\emptyset'', \emptyset')$ und wird durch die Übergabe einer leeren Menge an `conceptFromX` erzeugt:

```
let firstConcept relation =
  (* xSet is empty *)
  let xSet = Bitset.make (Bitrelation.xSize relation) in
    Some (conceptFromX xSet relation)
```

Die Funktion `firstConcept` berechnet den lektisch kleinsten Begriff eines Verbandes; sie besitzt den Typ $\mathit{relation} \rightarrow (\mathit{bitset} \times \mathit{bitset})\ \mathit{option}$. Ein Typ $\tau\ \mathit{option}$ besitzt zwei mögliche Werte: entweder *None* oder *Some* t, wobei t ein Wert des Typs τ ist. Die Idee ist einen Datentyp zur Verfügung zu haben, der optionale Werte repräsentiert – daher der Name. Zunächst erscheint es überflüssig, *option* für `firstConcept` zu verwenden, weil immer ein *Some*-Wert zurückgegeben wird. Da die Funktion `nextConcept` ebenfalls einen Rückgabewert vom Typ *option* besitzt, wird ihre gemeinsame Verwendung so vereinheitlicht. Sie berechnet aus dem zuletzt ermittelten Begriff den lektisch nächsten und liefert diesen entweder mit *Some*, oder zeigt durch *None* an, daß kein weiterer existiert. Der Typ von `nextConcept` ist $(\mathit{bitset} \times \mathit{bitset}) \rightarrow \mathit{relation} \rightarrow (\mathit{bitset} \times \mathit{bitset})\ \mathit{option}$; ihre Definition ist kurz, weil der entscheidende Teil von Ganters Algorithmus in die Funktion `findNext` ausgelagert ist.

```
let nextConcept (lastX,lastY) relation =
  findNext lastX (Bitset.copy lastX)
           (Bitrelation.xSize relation - 1) relation
```

Die Funktion `findNext` wird mit vier Parametern aufgerufen: der letzten Objektmenge (`lastX`), eine Kopie davon, der Relation (`relation`) und dem größten

möglichen Element einer Objektmenge. Die Funktion implementiert Theorem 8, indem sie rekursiv eine Menge von Kandidaten für den nächsten Begriff bestimmt und zurückgibt. Die Kopie des letzten Objektmenge wird per Seiteneffekte verändert und wird zur Objektmenge des nächsten Begriffes. Dies geschieht rekursiv (`rec`); die Funktion zeigt durch *None* an, wenn kein weiterer Begriff existiert. Sie besitzt den Typ $bitset \rightarrow bitset \rightarrow int \rightarrow relation \rightarrow (bitset \times bitset)\ option$.

```
let rec findNext lastX nextX x relation =
  if x < 0 then
    None
  else if Bitset.isMember x nextX then
    begin
      Bitset.remove x nextX;
      findNext lastX nextX (x-1) relation
    end
  else
    begin
      Bitset.add x nextX;
      let (xSet',ySet') as nextConcept = conceptFromX nextX
                                                      relation in
        if Bitset.isRestrictedSubset xSet' lastX x then
          begin
            Some nextConcept
          end
        else
          begin
            Bitset.remove x nextX;
            findNext lastX nextX (x-1) relation
          end
    end
```

Der formale Parameter `x` repräsentiert i aus Theorem 8: `x` wird der Objektmenge `nextX` hinzugefügt, um einen neuen Kandidaten `nextConcept` für den nächsten Begriff zu erhalten. Wenn die Objektmenge des letzten Begriffes und die des aktuellen Kandidaten in der $\prec_i$-Relation enthalten sind, ist der nächste Begriff gefunden und wird als *Some* `nextConcept` zurückgegeben. Wenn kein nächster Begriff existiert, ist das Ergebnis *None*. Da maximal $|\mathcal{O}|$-malig die Funktion `nextConcept` aufgerufen wird, beträgt die Komplexität von `findNext` $O(|\mathcal{O}|^2 \times |\mathcal{A}|)$.

Um alle Begriffe eines Kontextes zu erhalten, wird zunächst `firstConcept` aufrufen und dann in einer Schleife `nextConcept`, bis diese Funktion *None* zurückgibt. Die Komplexität zur Bestimmung aller Begriffe mit Ganters Algorithmus ist also $O(n \times |\mathcal{O}|^2 \times |\mathcal{A}|)$ mit $n = |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$.

3.3 Implementierung der Basisdatentypen

Für das Design der Basismodule *Bitset* und *Bitrelation* zur Implementierung von Ganters Algorithmus ist es nützlich, sein Verhalten zu studieren. Dazu wurden in einer Versuchsreihe zufällige Kontexte erzeugt und die Aufrufe zentraler Funktionen beim Bestimmen der Begriffe gezählt.

Die Versuchsreihe umfaßte 100 Kontexte, in deren 30×30 große Kontexttabellen jeweils zufällig 250 Einträge plaziert wurden. Anschließend wurden von jedem Kontext mit Ganters Algorithmus die Begriffe bestimmt und die Aufrufhäufigkeiten verschiedener Funktionen gezählt. Die Beobachtungen für die gesamte Versuchsreihe sind in Tabelle 3.4 zusammengefaßt. Für jede Funktion gibt $\overline{x}$ das arithmetische Mittel der Aufrufhäufigkeit an und $\sigma(x)$ ihre Standardabweichung. Die Spalten $\max(x)$ und $\min(x)$ zeigen die minimale und maximale beobachtete Aufrufhäufigkeit innerhalb der Versuchsreihe.

Funktion	$\overline{x}$	$\sigma(x)$	$\max(x)$	$\min(x)$
`Bitrelaton.commonX`	4074.09	328.08	5199	3245
`Bitrelaton.commonY`	4074.09	328.08	5199	3245
`Bitset.add`	4573.09	328.08	5698	3744
`Bitset.asList`	8148.18	656.15	10398	6490
`Bitset.intersect`	19403.11	1679.79	25470	15022
`Bitset.isMember`	4766.92	326.73	5937	3940
`Bitset.isRestrictedSubset`	4073.09	328.08	5198	3244
`Bitset.make`	8211.18	656.15	10461	6553
`Bitset.remove`	4366.52	320.24	5481	3560
`Concept.conceptFromX`	4074.09	328.08	5199	3245
`Concept.findNext`	4767.92	326.73	5938	3941
`Concept.nextConcept`	401.40	15.36	458	365

Tabelle 3.4: Aufrufhäufigkeiten x verschiedener Funktionen bei einem Kontext der Größe 30×30 mit 250 zufälligen Elementen und $n = 100$ durchgeführten Versuchen.

Durchschnittlich enthielt ein Kontext 401 Begriffe, abzulesen an den Aufrufen von `nextConcept`. Diese Funktion ruft selbst die Funktion `findNext` auf, die zu einem gegebenen Begriff den lektisch nächsten bestimmt, indem sie bis zu $|\mathcal{O}| = 30$ Kandidaten generiert. Im Durchschnitt wurden deshalb 4768 Begriffe für jeden Kontext bestimmt. Im Mittel bedeutet dies, daß für jeden Begriff des Ergebnisses $4767.92/401.40 \approx 12$ Begriffe berechnet wurden. Auffallend ist die hohe Anzahl von Aufrufen von `intersect` – diese Funktion verdient deshalb für die effiziente Implementierung von Ganters Algorithmus besondere Beachtung. Formale Begriffsanalyse bestimmt die Gemeinsamkeiten in einem Kontext mit Hilfe des $'$-Operators, der durch den Schnitt von Mengen implementiert wird. Als Konsequenz wird diese Funktion am häufigsten aufgerufen und ihre Effizienz

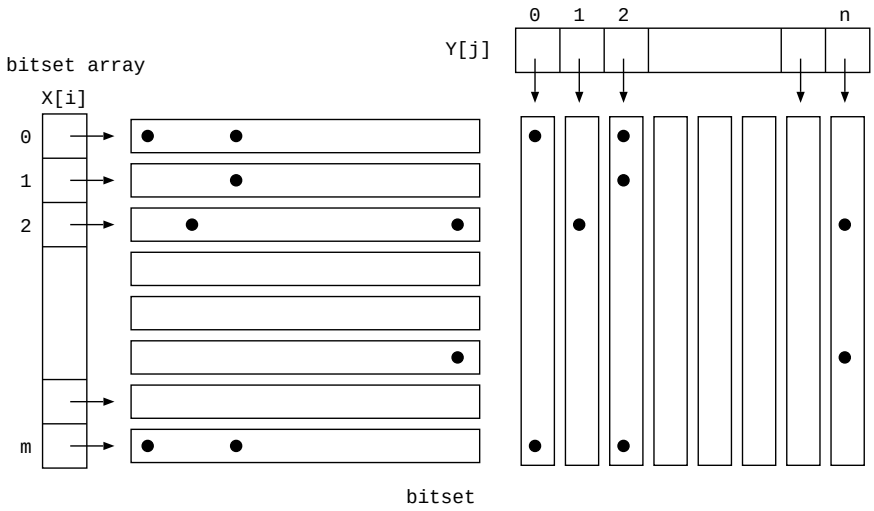

Abbildung 3.4: Implementierung von *relation* durch zwei *bitset* Arrays. Die Redundanz erlaubt den Schnitt von Zeilen und Spalten effizient zu bestimmen.

sollte beim Design der Basisdatentypen im Vordergrund stehen.

Die Implementierung des Moduls *Bitset* verwendet zur Repräsentation von Mengen Bitvektoren. Jedes Element der Menge wird durch ein Bit repräsentiert; Bitvektoren erlauben eine zeit-effiziente Implementierung aller Operationen und sind auch mit Ausnahme von sehr dünn besetzten Mengen platz-effizient. Ihr Nachteil ist, daß eine Implementierung mit dynamisch wachsenden und schrumpfenden Mengen aufwendig ist, weil viele Mengenoperationen vor ihrer Ausführung die Herstellung einer Normalform voraussetzen. Da in der Begriffsanalyse die Größe einer Menge aber bei ihrer Erzeugung bekannt ist, kann dieser Aspekt hier unberücksichtigt bleiben. Bitvektoren sind konzeptionell Mengen von natürlichen Zahlen, so daß eine realistische Implementierung einer Begriffsanalyse ein weiteres Modul benötigt, das die Implementierung einer Bijektion von beliebigen Objekten und Attributen auf natürliche Zahlen bereitstellt.

Das Modul *Bitrelation* stellt binären Relationen zur Verfügung. Die Funktionen `commonX` und `commonY` implementieren den $'$-Operator durch den Schnitt von ausgewählten Zeilen und Spalten der Kontexttabelle. Eine binäre Relation kann durch ein Array repräsentiert werden, das jedes Objekt auf eine Menge von Attributen abbildet, die zu ihm in Relation stehen. Dann kann `commonX` durch den Schnitt von (Attribut-)Mengen implementiert werden.

Der Schnitt von Objektmengen (Spalten) durch die Funktion `commonY` ist dagegen nicht effizient, da dieser Darstellung explizite Mengen von Objekten fehlen. Eine effizientere Implementierung kann Redundanzen in Kauf nehmen und eine binäre Relation durch zwei Arrays von Mengen implementieren: das erste Array bildet Objekte auf Mengen von adjazenten Attributen ab, das zweite Attribute auf Mengen adjazenter Objekte. Diese Implementierung ist in Abbildung 3.4

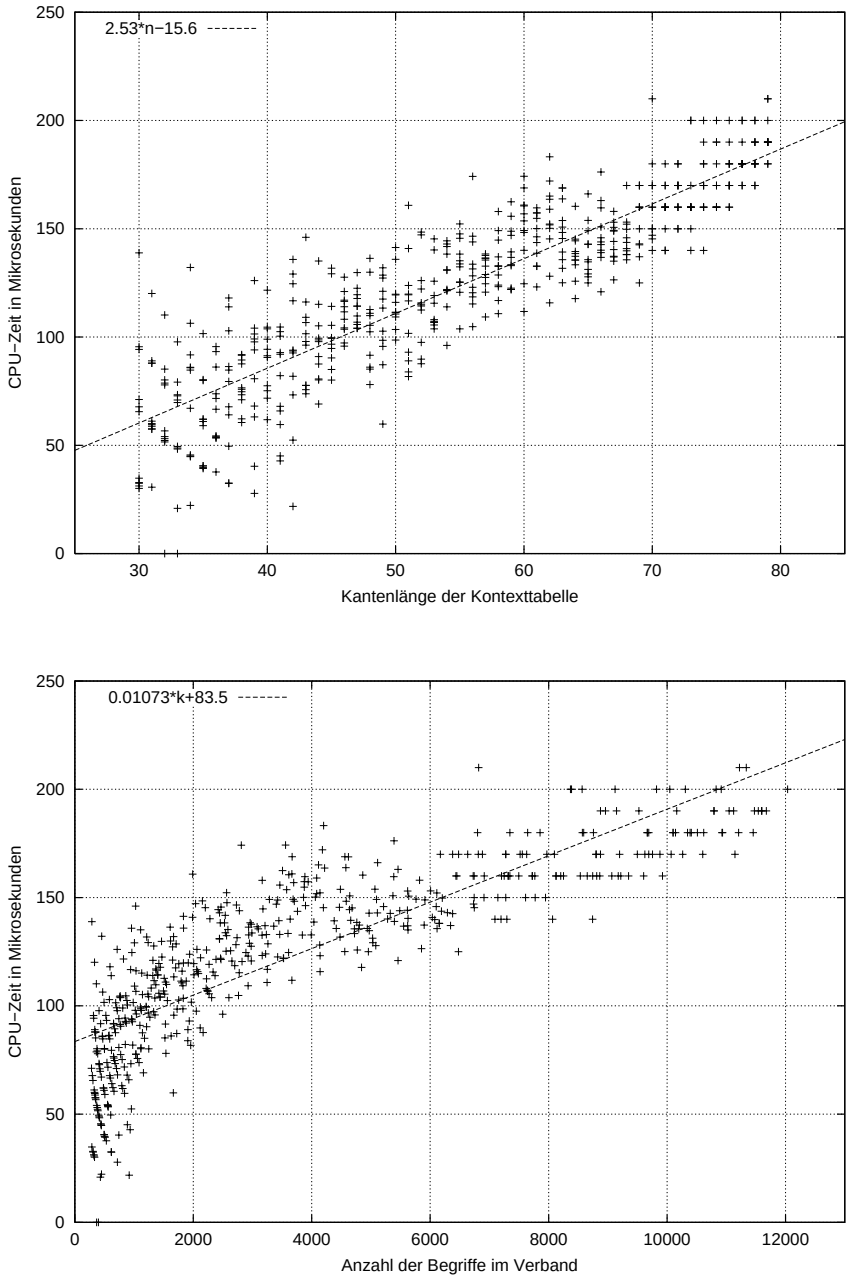

Abbildung 3.5: Zeitbedarf für die Bestimmung eines Begriffs in verschiedenen Kontexten. Die Ausführungszeiten in μs sind oben über der Größe n der Kontexte und unten über der Anzahl k der Begriffe in dem Kontext aufgetragen.

skizziert; in ihr können `commonX` und `commonY` unmittelbar durch den Schnitt von Mengen implementiert werden.

3.4 Performance-Messung

Für eine Implementierung von Ganters Algorithmus in C [53, 68] mit den skizzierten Datenstrukturen wurde eine Versuchsreihe zur Performance-Analyse durchgeführt. Kontexttabellen der Größe $n \times n$ mit $n = 30, \ldots, 79$ wurden darin zufällig mit $n^2/4$ Einträgen belegt und anschließend wurde die Anzahl ihrer Begriffe ermittelt. Für jeden Kontext der Größe $n \times n$ wurde dieser Versuch 11-malig wiederholt, so daß insgesamt die Begriffe von $50 \times 11 = 550$ zufälligen Kontexten bestimmt wurden. Durch Profiling mit dem `gprof`-Profiler [39] wurden die durchschnittlichen Ausführungszeiten der Funktion `nextConcept` ermittelt. Diese Funktion berechnet jeweils den lektisch nächsten Begriff in der Folge von allen Begriffen eines Kontextes. Alle Messungen fanden auf einem Linux-2.0-System mit 200 MHz AMD-K6-Prozessor und dem GNU-C-Compiler `gcc` 2.7 statt. Die Ergebnisse sind in Abbildung 3.5 dargestellt; in dem oberen Abbildung wurden die gemessenen durchschnittlichen Laufzeiten der Funktion in μs gegen die Kantenlänge n der Kontexttabelle aufgetragen. In dem unteren Schaubild wurden die selben Daten gegen die Anzahl k der Begriffe im jeweiligen Kontext aufgetragen. In beiden Schaubildern ist zusätzlich eine lineare Approximation zwischen n, k und der Laufzeit eingezeichnet. Sie wurde mit Hilfe der Methode der kleinsten Fehlerquadrate bestimmt.

Durchschnittlich betrug die Zeitdauer zur Ermittlung des nächsten Begriffes zwischen 50 und 200 μs, ansteigend mit der Größe n der Kontexttabellen und der Anzahl k der Begriffe. Die Begriffsverbände enthielten zwischen 400 und 12 000 Begriffe. Die Streuungen bei $n < 40$ und $k < 1000$ sind auf die geringe Auflösung des Profiling zurückzuführen. Der Profiler berechnet die durchschnittliche Dauer eines Funktionsaufrufs als den Quotienten aus Gesamtlaufzeit aller Aufrufe und der Anzahl der Aufrufe. Bei wenigen Aufrufen und kurzen einzelnen Laufzeiten wirkt sich die beschränkte Zeitauflösung des Profilers besonders aus. Der Anstieg der durchschnittlichen Laufzeit mit n und k hat zwei Gründe: Erstens nimmt die Größe der Mengen mit wachsendem n zu, die in `nextConcept` bearbeitet werden. Speziell der Schnitt von Mengen ist in einer Bitvektor-Implementierung linear von der Größe der Mengen abhängig.[2] Und zweitens wächst die Zahl der potentiellen Kandidaten für den nächsten Begriff in der Funktion `findNext` mit wachsendem n. Bei großen Werten für n werden durchschnittlich mehr Kandidaten bestimmt, bevor der nächste Begriff gefunden wird und dies geht in die Laufzeit von `nextConcept` ein. Die beobachteten Laufzeiten bestätigen, daß bei geeigneten Datenstrukturen die Gesamtlaufzeit von Ganters Algorithmus durch die Anzahl der Begriffe in einem Kontext bestimmt wird und nicht durch den Aufwand, einen einzelnen Begriff zu berechnen. Der steigt zwar mit der Größe von Kontexten und der Anzahl ihrer Begriffe, allerdings nur linear, soweit man

[2] typischerweise sind 32 oder 64 Elemente (Bits) einer Menge in einem Speicherwort zusammengefaßt.

dies durch die Versuchsreihe beurteilen kann.

3.5 Optimierung durch Caching

Die empirische Auswertung von Ganters Algorithmus zur Berechnung aller Begriffe in Abschnitt 3.3 legt eine einfache Optimierung nahe: die Funktion `findNext` (siehe 3.2, Seite 21) berechnet aus dem zuletzt bestimmten Begriff einen Kandidaten für den lektisch nächsten Begriff. Wenn dieser Begriff nicht dem Kriterium für den lektisch nächsten Begriff genügt, wird er verworfen, um später erneut berechnet zu werden. Bei der empirischen Auswertung (Tabelle 3.4) wurden durchschnittlich 4768 Kandidaten für einen 401 Begriffe umfassenden Begriffsverband berechnet. Im Durchschnitt wurde ein Begriff also 11-mal berechnet, verworfen und erst bei seiner zwölften Berechnung im Ergebnis festgehalten.

Statt den Kandidaten (O_2, A_2) als Nachfolger für (O_1, A_1) zu verwerfen, wenn $O_1 \prec_i O_2$ nicht erfüllt ist (siehe Theorem 8), kann er in einem Cache unter dem Schlüssel A_2 abgelegt werden. Die Funktion `findNext` berechnet den Kandidaten für die nächste Objektmenge O_2 aus der Objektmenge O_1 des zuvor berechneten Begriffes, indem sie ein neues Objekt o hinzufügt und $A_2 = (O_1 \cup \{o\})'$ bestimmt. Falls A_2 sich in dem Cache befindet, muß $O_2' = A_2'$ nicht mit einem Aufwand von $O(|\mathcal{O}| \times |\mathcal{A}|)$ berechnet werden, sondern kann dem Cache entnommen werden. Durch Caching wird sich insbesondere die Zahl der Schnittmengen-Operationen (`intersect`) drastisch reduzieren, weil weniger $'$-Operationen ausgeführt werden müssen.

Der Cache enthält Begriffe, die durch Ganters Algorithmus zu früh berechnet werden. Deswegen ist garantiert, daß sie genau einmal in dem Cache erfolgreich gesucht werden. Nachdem ein Begriff in dem Cache gefunden wurde, kann er daraus gelöscht werden. Weil ein Begriff eindeutig ist, kann er nicht ein zweites Mal in der lektischen Folge aller Begriffe auftreten.

Eine um Caching erweiterte Implementierung des Algorithmus bietet sich insbesondere an, wenn alle Begriffe vollständig im Hauptspeicher gehalten werden sollen. Der Platzbedarf des Algorithmus ist dann der gleiche wie der des zuvor beschriebenen Algorithmus. Der Cache enthält lediglich Begriffe, die sowieso Teil des Ergebnisses sind und niemals überflüssige Begriffe. Er speichert sie zwischen, bis sie an der richtigen Stelle im Ergebnis eingefügt werden können.

Wenn die Größe berechenbarer Begriffsverbände nicht durch den Hauptspeicher beschränkt sein soll, müssen die Begriffe des Ergebnis extern abgelegt werden. In diesem Fall bietet sich ein Cache begrenzter Größe an. Weil einmal im Cache gefundene Begriffe dort gelöscht werden können ist zu erwarten, daß auch mit einem relativ zur Anzahl aller Begriffe kleinen Cache Begriffe nicht mehrfach berechnet werden müssen. Das Verhalten von Ganters Algorithmus mit einem beschränkten Cache hängt davon ab, wie (lektisch) nahe die generierten Kandidaten dem gesuchten nächsten Begriff typischerweise sind.

3.6 Berechnung des Verbandsstruktur

Ganters Algorithmus berechnet die Menge aller Begriffe eines Kontextes, die eine Verbandsstruktur besitzen. Diese in der Menge lediglich implizit enthaltene Struktur wird durch ein Hasse-Diagramm explizit: es zeigt Begriffe als Knoten eines ungerichteten Graphen und die direkte[3] $\leq$-Relation zwischen zwei Begriffen als eine Kante zwischen Knoten des Graphen. Viele Anwendungen formaler Begriffsanalyse benötigen die Verbandsstruktur der Begriffe; sie muß ihnen ähnlich wie ein Hasse-Diagramm als Datenstruktur zu Verfügung gestellt werden. Dieser Abschnitt beschreibt zwei Algorithmen, die die Verbandsstruktur aus einer Menge von Begriffen berechnen, und ihn in als Datenstruktur ablegen.

Der Begriffsverband als Datenstruktur enthält nicht nur die Begriffe eines Kontextes, sondern stellt zusätzlich Beziehungen zwischen ihnen her. Ein Begriff c im Begriffsverband kann zum Beispiel die folgenden Informationen bereithalten:

- Objekt- und Attributmenge
- Verweise auf direkte Ober- und Unterbegriffe
- Eine Menge aller Objekte, die nicht in den Objektmengen von Unterbegriffen enthalten sind: $\{o \in \mathcal{O} \mid \gamma(o) = c\}$. Dies ist die Menge aller Objekte, die den aktuellen Begriff c im Hasse-Diagramm markiert. Zusätzlich die Menge von Attributen, die nicht in Oberbegriffen enthalten ist: $\{a \in \mathcal{A} \mid \mu(a) = c\}$. Diese Mengen enthalten die Objekte und Attribute, deren Objekt- und Attributbegriff gerade der aktuelle Begriff c ist.

Je nach Anwendung kann ein Begriff im Begriffsverband weitere Informationen enthalten. Für die Repräsentation des Verbandes bieten sich Arrays und mit Zeigern realisierte Graphen an. Die Elemente des Arrays beziehungsweise des Graphen sind die Begriffe, die neben ihren Objekt- und Attributmengen weitere Mengen und Verweise auf anderen Begriffe enthalten.

3.6.1 Algorithmus I

Der nachfolgende Algorithmus berechnet aus einem Array von Begriffen den Begriffsverband und läßt sich auch auf Graphen übertragen. Das Modul *Lattice* implementiert den Begriffsverband. Darin wird jeder Begriff durch einen Record repräsentiert und der Verband selbst als ein Array von Begriffen:

```
type concept = {  concept:      Concept.concept;
                  newX:         Bitset.bitset;
                  newY:         Bitset.bitset;
                  super:        Bitset.bitset;
```

[3] das transitive Redukt der $\leq$-Relation

```
                    sub:          Bitset.bitset;
                }
type lattice =    concept array
```

Ein Begriff (`concept`) enthält: die Objekt- und Attributmengen (`concept`), die Menge von Objekten, die nicht in Unterbegriffen enthalten sind (`newX`), analog die Menge von Attributen, die nicht in Oberbegriffen enthalten sind (`newY`) und Mengen der Array-Indizes direkter Ober- und Unterbegriffe (`super`, `sub`). Die Begriffe in dem Array vom Typ *lattice* sind lektisch aufsteigend sortiert, befinden sich also in der Reihenfolge, die sich durch Ganters Algorithmus und den zyklischen Aufruf von `firstConcept`/`nextConcept` ergibt.

Der Speicherbedarf von `concept array` ist für große Begriffsverbände problematisch, wenn zur Implementierung von Mengen Bitvektoren verwendet werden. Die Komponenten `newX`, `newY`, `super` und `sub` sind dann Bitvektoren, deren Länge der Anzahl der Elemente im Begriffsverband entspricht. Abgesehen von dem größten und kleinsten Element eines Verbandes sind diese Mengen dünn besetzt und zwar um so dünner, je größer der Begriffsverband ist. Abbildung 3.6 zeigt den Speicherbedarf in Bytes beispielhaft für einen Kontext mit 200 Objekten ($|\mathcal{O}|$), 100 Attributen ($|\mathcal{A}|$) und einer variablen Anzahl n von Begriffen.

$m_1(n)$	$=$	$n/8 \times (\|\mathcal{O}\| + \|\mathcal{A}\|)$	Speicher in Bytes für Begriffe
$m_2(n)$	$=$	$n/8 \times 2 \times (\|\mathcal{O}\| + \|\mathcal{A}\| + n)$	Speicher in Bytes für `concept array`

Die Berechnung des Speicherplatzes berücksichtigt nicht den Speicherverschnitt und den Bedarf für zusätzliche Infrastruktur und ist daher nur ein grober Anhalt. Abbildung 3.6 enthält eine Kurve für den Speicherbedarf zur Ablage der Begriffe m_1 und eine weitere m_2 für die oben angegebene Datenstruktur `lattice`, die Begriffe und ihre Verbandsstruktur explizit macht. Die Kurven verdeutlichen, warum eine dynamische Datenstruktur für Mengen (von Verweisen auf Begriffe) nötig ist, die Platz nur für tatsächlich aufgenommene Elemente belegt. Die dynamische Speicherverwaltung von Begriffsmengen ist algorithmisch ungünstiger als die bislang verwendete Implementierung durch Bitvektoren; Operationen auf Begriffsmengen sind gleichzeitig wesentlich seltener als auf Objekt- oder Attributmengen,

Ein Begriffsverband entsteht als Datenstruktur, wenn alle Begriffe eines Kontextes lektisch aufsteigend in ein *concept array* geschrieben werden. Die Felder `newX`, `newY`, `sub` und `super` werden zunächst mit Standardwerten belegt. Anschließend durchläuft der Algorithmus das Array und trägt die schon implizit in den Objekt- und Attributmengen enthaltenen Informationen in die Felder ein. Er basiert auf den folgenden Beobachtungen über die Begriffe im Array:

- Wenn $c[i]$ und $c[j]$ Begriffe im Array an den Positionen i und j sind, dann folgt aus $c[j] \leq c[i]$ mit Theorem 7: $j \leq i$. Ein Unterbegriff besitzt einen kleineren Index als seine Oberbegriffe.

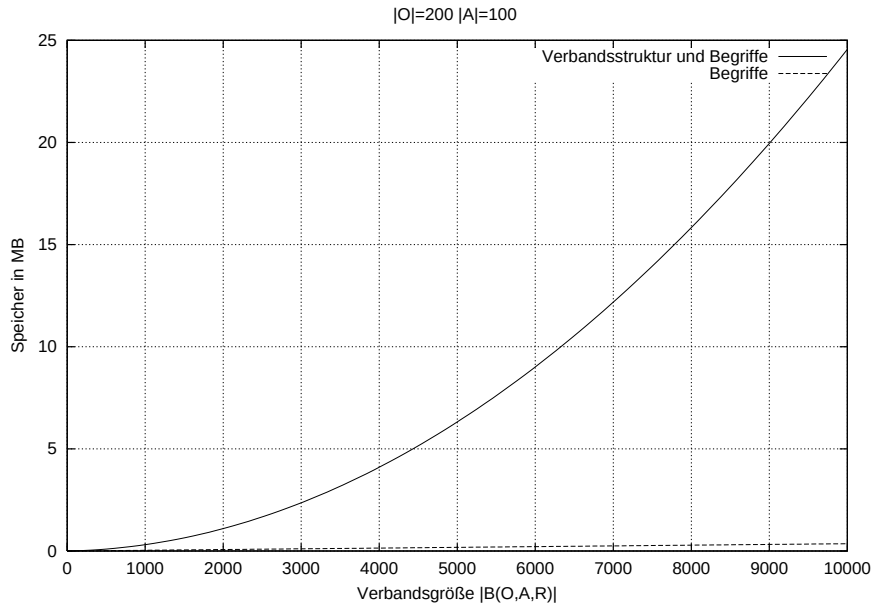

Abbildung 3.6: Speicherbedarf in Megabytes für die Ablage von Begriffen und ihre Verbandsstruktur für $|\mathcal{O}| = 200$ und $|\mathcal{A}| = 100$.

- Aus $c[j] \leq c[i]$ und $\forall k \in \{j+1 \ldots i-1\} : c[k] \not\leq c[i]$ folgt, daß $c[j]$ ein direkter Unterbegriff von $c[i]$ ist. Wenn nämlich ein Begriff $c[k]$ mit $c[j] \leq c[k] \leq c[i]$ existiert, dann muß wegen der Sortierung des Arrays $j \leq k \leq i$ gelten.

- Wenn $c[j]$ ein direkter Unterbegriff von $c[i]$ ist, dann gilt $j \in \mathit{sub}[i]$ und $i \in \mathit{super}[j]$. Außerdem enthält $\mathit{newX}[i]$ kein Objekt aus $c[j]$ und umgekehrt $\mathit{newY}[j]$ kein Attribut aus $c[i]$.

Der Algorithmus in Objective Caml formuliert ergibt die Funktion `fillLattice` mit dem Typ *Concept.concept array* → *int* → *concept list* → *unit*. Sie trägt eine lektisch aufsteigend geordnete Liste von n Begriffen in ein Array der Größe n ein und bestimmt alle Felder der *concept*-Records.

```
let fillLattice lat size concepts  =
    let reachable  = Bitset.make size in
    let rec loop i = function
        | []      -> ()
        | c::cs   -> lat.(i) <- nullEntry size c;
                     Bitset.empty reachable;
                     for j = i-1 downto 0 do
                         if Concept.isSubConcept lat.(j).concept
                                                 lat.(i).concept
                         then if Bitset.isMember j reachable then
```

```
                                    Bitset.union reachable
                                                 lat.(j).sub
                                else begin
                                    Bitset.add j lat.(i).sub;
                                    Bitset.add i lat.(j).super;
                                    Bitset.add j reachable;
                                    Bitset.union reachable
                                                 lat.(j).sub;

                                    Bitset.difference lat.(i).newX
                                        (Concept.x lat.(j).concept);
                                    Bitset.difference lat.(j).newY
                                        (Concept.y lat.(i).concept);
                                end
                        done;
                        loop (i+1) cs
    in
        loop 0 concepts
```

Die lokale rekursive Funktion `loop` mit dem Typ $int \rightarrow Concept.concept\ list \rightarrow unit$ fügt den ersten Begriff `c` der übergebenen Liste `c::cs` an der Position `i` im Array `lat` ein. Dabei werden durch die Funktion `nullEntry` die Felder `sub` und `super` mit leeren Mengen, die Felder `newX` und `newY` mit der Objekt- und Attributmenge vorbesetzt. Die Menge `reachable` enthält alle Unterbegriffe von `c` – nicht nur die direkten Unterbegriffe – und dient der Erkennung von direkten Unterbegriffen.

Jedesmal, wenn ein Begriff c an der Position i im Array eingetragen wird, werden in einer `for`-Schleife alle Positionen $0 \ldots i-1$ im Array besucht. Die Operationen innerhalb der `for`-Schleife werden deshalb $O(n^2)$-malig ausgeführt, wobei $n = |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$. Innerhalb der Schleife ist die Komplexität des `union` Aufrufes $O(n)$, da die Menge `reachable` maximal n Elemente besitzt; die beiden Aufrufe der Funktion `difference` besitzen die Komplexität $O(|\mathcal{O}|)$ und $O(|\mathcal{A}|)$. Da für große Begriffsverbände $O(n) > \max O(|\mathcal{A}|), O(|\mathcal{O}|)$ gilt, ist die Komplexität des Schleifenrumpfes $O(n)$ und damit die Gesamtkomplexität des Algorithmus $O(n^3)$.

3.6.2 Algorithmus II

Die Ursache der kubischen Komplexität des vorgestellten Algorithmus ist, daß für jeden Begriff alle potentiellen Unterbegriffe daraufhin getestet werden müssen, ob sie tatsächlich direkte Unterbegriffe sind. Dies sind maximal $|B(\mathcal{O}, \mathcal{A}, \mathcal{R})| - 1$ Tests für einen einzelnen Begriff. Ganter und Kuznetsov haben einen Algorithmus [31, 57] vorgeschlagen, der weniger Begriffe testet und deswegen eine wesentlich geringere Komplexität aufweist. Er basiert auf dem nachfolgenden Theorem:

Theorem 9 *Sei $(O, A) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ und $O \neq \mathcal{O}$. Die Objektmengen der direkten Oberbegriffe von (O, A) sind die minimalen Mengen der Form*

$$(O \cup \{o\})'', \quad o \notin O$$

(in [31] ohne Beweis)

Für einen Begriff (O, A) werden alle Mengen $(O \cup \{o\})''$ mit $o \notin \mathcal{O}$ betrachtet, also maximal $|\mathcal{O}|$ Mengen, wenn $O = \emptyset$ gilt. Da in praktisch allen Verbänden $|\mathcal{O}| < |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ gilt, ist dieser Algorithmus wesentlich effizienter.

Nicht jede Menge der Form $(O \cup \{o\})''$ ist allerdings die Objektmenge eines direkten Oberbegriffes von (O, A). Nur die minimalen Mengen unter allen Mengen der Form $(O \cup \{o\})''$ gehören zu direkten Oberbegriffen. Dazu werden die Objektmengen bezüglich der $\subseteq$-Relation geordnet. Eine Menge $O_1 = (O \cup \{o_1\})''$ ist genau dann minimal unter allen anderen, wenn $O_1 \not\supset (O \cup \{o\})''$ für alle $o \notin O \cup \{o_1\}$ gilt.

Theorem 9 ist für eine direkte Umsetzung in einen Algorithmus nur bedingt geeignet, weil es offen läßt, wie die minimalen Mengen effizient bestimmt werden. Zur Charakterisierung der direkten Oberbegriffe existiert ein weiteres Theorem, das sich für eine algorithmische Umsetzung anbietet.

Theorem 10 *Sei $(O, A) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ und $O \neq \mathcal{O}$. Dann ist $(O \cup \{o\})''$ mit $o \notin O$ genau dann Objektmenge eines direkten Oberbegriffes von (O, A), wenn gilt: $\forall x \in ((O \cup \{o\})'' \setminus O) : (O \cup \{x\})'' = (O \cup \{o\})''$.*

Beweis. (1) $(O \cup \{o\})''$ ist die Objektmenge eines Oberbegriffes von (O, A), da $(O \cup \{o\})'' \supset O$ und $((O \cup \{o\})'', (O \cup \{o\})')$ ein Begriff ist. Sei $(O \cup \{o\})''$ die Objektmenge eines direkten Oberbegriffes (O_1, A_1) von (O, A). Für ein beliebiges (also alle) $x \in (O \cup \{o\})'' \setminus O$ gilt: weil $x \notin O$, $x \in O_1$ ist $\gamma(x) \vee (O, A) = (O_1, A_1)$. Speziell für die Attributmengen gilt: $\{x\}' \cap A = A_1 \iff \{x\}' \cap O' = A_1 \iff (\{x\} \cup O)' = A_1 \iff (\{x\} \cup O)'' = A_1' = O_1 = (O \cup \{o\})''$.

(2) Gelte $\forall x \in ((O \cup \{o\})'' \setminus O) : (O \cup \{x\})'' = (O \cup \{o\})''$. $(O \cup \{o\})''$ ist die Objektmenge eines Oberbegriffes von (O, A). Unter der Annahme, daß $(O \cup \{o\})''$ die Objektmenge eines indirekten Oberbegriffes ist, muß ein direkter Oberbegriff (O_1, A_1) existieren mit $(O \cup \{o\})'' \supset O_1 \supset O$. Deshalb existiert $x \in O_1$, $x \notin O$ und es gilt $(O \cup \{x\})'' = O_1$. Da $x \in (O \cup \{o\})''$ und $(O \cup \{x\})'' = O_1 \neq (O \cup \{o\})''$ ist dies ein Widerspruch zur Voraussetzung. Also ist die Annahme falsch und $(O \cup \{o\})''$ die Objektmenge eines direkten Oberbegriffes.

In einem direkten Oberbegriff erzeugen alle gegenüber dem Ausgangsbegriff hinzugekommenen Objekte selbst den Oberbegriff: Wenn O_1 wie in Abbildung 3.7 eine Objektmenge des direkten Oberbegriffes von (O_0, A_0) ist, dann enthält O_1 gegenüber O_0 zusätzliche Objekte. Jedes dieser zusätzlichen Objekte $o_1, \ldots o_3$ erzeugt wieder $O_1 = (O_0 \cup \{o_i\})''$. Die Menge O_2 ist ebenfalls eine Objektmenge eines Oberbegriffes von (O_0, A_0) und enthält auch $o_1, \ldots, o_3$, sie ist allerdings kein

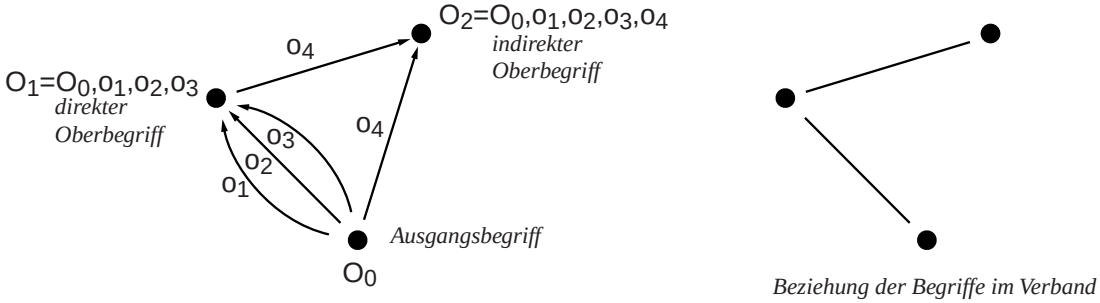

Abbildung 3.7: Skizze für Theorem 10

```
foreach o ∈ 𝒪 \ O do
   min[o] := true
foreach o ∈ 𝒪 \ O do
   X := (O ∪ {o})″ \ O
   if min[x] = false for all x ∈ X \ {o} then
      print X ∪ O
   else
      min[o] := false
```

Abbildung 3.8: Algorithmus II von Ganter und Bittner zur Bestimmung aller direkten Oberbegriffe

direkter Oberbegriff. Zwar ist $(O_0 \cup \{o_4\})'' = O_2$, aber $(O_0 \cup \{o_1\})'' = O_1 \neq O_2$. Also erzeugt $o_1 \in O_2$ nicht wieder O_2 und deshalb ist O_2 keine Objektmenge eines direkten Oberbegriffes.

Ganter und Bittner haben auf Basis von Theorem 10 den in Abbildung 3.8 dargestellten Algorithmus entwickelt [30], der effizient die Objektmengen der direkten Obermengen eines Begriffes bestimmt. Dazu muß er maximal $|\mathcal{O}|$ Begriffe durch die Operation $''$ berechnen. Die Menge aller direkten Oberbegriffe eines Begriffes kann deshalb mit der Komplexität $O(|\mathcal{O}|^2 \times |\mathcal{A}|)$ bestimmt werden.

Der Algorithmus betrachtet in einer beliebigen Reihenfolge alle Objekte o, die nicht bereits in der Objektmenge O des Ausgangsbegriffes enthalten sind. Zu jedem Objekt o generiert er die Objektmenge eines (direkten oder indirekten) Oberbegriffes $(O \cup \{o\})''$ und betrachtet darin die durch die $''$-Operation neu hinzugekommenen Objekte $(O \cup \{o\})'' \setminus (O \cup \{o\})$. Wenn für mindestens eines der neu hinzugekommenen Objekte x `min[`x`] = true` gilt, wird `min[`o`] = false` gesetzt. Dies führt im Falle der Objektmenge O_1 eines direkten Oberbegriffes dazu, daß (mit einer Ausnahme) für alle Objekte o_i, die $O_1 = (O \cup \{o_i\})''$ erzeugen, `min[`o_i`] = false` gesetzt wird. Nachdem nämlich alle bis auf eines der möglichen Objekte zur Generierung von O_1 verwendet wurden, findet der Algorithmus beim letzten zur Generierung verwendete Objekt o_j alle anderen mit `min[`x`] = false`

vor und erkennt, daß O_1 ein direkter Oberbegriff ist. Im Falle dieses letzten Objektes o_j bleibt der initiale Wert `min[`o_j`] = true` bestehen. Jeder indirekte Oberbegriff (wie O_2 in Abbildung 3.8) enthält Objekte, die direkte Oberbegriffe erzeugen und deshalb mindestens ein Objekt o_j mit `min[`o_j`] = true`, das einen (anderen) *direkten* Oberbegriff O_1 generiert. Folgerichtig wird die Bedingung zur Erkennung direkter Oberbegriffe niemals erfüllt.

Der Algorithmus von Ganter und Bittner kann zu einem Algorithmus erweitert werden, der alle Begriffe eines Kontextes und ihre Verbandsstruktur berechnet. Ausgehend von dem kleinsten Begriff $(\emptyset'', \emptyset')$ eines Verbandes bestimmt er rekursiv deren direkte Oberbegriffe. Dies führt zu dem effizienten Algorithmus in Abbildung 3.9.

Die Funktion `super` berechnet ausgehend von der Objektmenge eines Begriffes die Menge direkter Oberbegriffe. Jeder Begriff `c` besitzt vier Mengen: `obj` und `atr` für Objekte und Attribute, sowie `sub` und `super` für (Verweise auf) direkte Unter- und Oberbegriffe. Die Ober- und Unterbegriffsbeziehungen eines Begriffes werden im Verlauf des Algorithmus abschnittsweise berechnet. Jeder berechnete Begriff wird deshalb unter seiner Objektmenge als Schlüssel in einem Suchbaum `B` abgelegt, um ihn so später zu identifizieren, und seine Verbandsbeziehungen inkrementell ergänzen zu können. Die Schlüssel des Suchbaumes sind lektisch geordnet; die Operation `insert` fügt einen neuen Begriff in den Suchbaum ein, `lookup` ermittelt ihn in dem Suchbaum und `next` liefert den lektischen Nachfolger eines Begriffes zurück.

Während laufend in den Suchbaum neue Begriffe eingetragen werden, verwendet der Algorithmus gleichzeitig die Funktion `next`, um alle Begriffe zu erreichen. Das Zusammenspiel von `insert` und `next` ist deshalb subtil: jeder (direkte) Oberbegriff ist lektisch größer als sein Unterbegriff (Theorem 7). Deshalb werden bei der Bearbeitung der direkten Oberbegriffe eines Begriffes `c` nur Begriffe in `B` eingetragen, die lektisch *größer* als `c` sind und zu einem späteren Zeitpunkt deshalb durch `next` erreicht werden. Dies garantiert, daß trotz des gleichzeitigen Einfügens und Traversierens der Elemente in `B` alle eingefügten Begriffe erreicht werden. Gleichzeitig entsteht dadurch eine Gefahr, daß der Suchbaum zu einer Liste degeneriert. Die Implementierung des Suchbaumes sollte deswegen den Baum nach dem Einfügen von Begriffen (annähernd) ausgleichen. Die Ausgleichoperationen werden auch durch das Nutzungsprofil gerechtfertigt: jeder Begriff wird nur einmal in dem Suchbaum eingetragen und erfordert dabei möglicherweise eine Ausgleichsoperation, aber entsprechend der Anzahl seiner direkten Oberbegriffe darin gesucht.

Das `loop`-Konstrukt des Algorithmus wird für jeden Begriff des Kontextes einmal durchlaufen, also $n = |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ mal. Für jeden Begriff wird die Menge seiner Oberbegriffe durch `super` berechnet; der asymptotische Aufwand dafür ist $O(|\mathcal{O}|^2 \times |\mathcal{A}|)$. Weil maximal $|\mathcal{O}|$ direkte Oberbegriffe für einen Begriff existieren, wird die innere `foreach`-Schleife maximal $|\mathcal{O}|$-malig durchlaufen. Die darin enthaltenen Operationen auf dem Suchbaum `B` besitzen die Komplexität

```
super(O) =
     min := 𝒪 \ O
     result := ∅
     foreach o ∈ 𝒪 \ O do
        c.atr := (O ∪ {o})′
        c.obj := c.atr′
        X := c.obj\O
        if (min ∩ (X \ {o})) ≠ ∅ then
           result := result ∪ {c}
        else
           min := min \ {o}
     return result

lattice () =
     c.obj := ∅″
     c.atr := ∅′
     insert(c,B)
     loop
        foreach i in super(c.obj) do
           try j:= lookup(i.obj,B)
           with NotFound ->
              j := i
              insert(i,B)
           j.sub := j.sub ∪ {c}
           c.super := c.super ∪ {j}
         try c := next(c,B)
         with NotFound -> exit
     endloop
     return B
```

Abbildung 3.9: Algorithmus zur Bestimmung aller Begriffe und ihrer Verbandsstruktur

$O(|\mathcal{O}| \times \log n)$, da Objektmengen als Schlüssel in dem Suchbaum dienen. Insgesamt ergibt sich für die Komplexität des Algorithmus $O(n \times |\mathcal{O}|^2 \times |\mathcal{A}| + n \times |\mathcal{O}| \times |\mathcal{O}| \times \log n)$. Da $n \leq 2^{|\mathcal{A}|}$ gilt, kann $\log n$ durch $|\mathcal{A}|$ abgeschätzt werden: $O(n \times |\mathcal{O}|^2 \times |\mathcal{A}| + n \times |\mathcal{O}|^2 \times |\mathcal{A}|) = O(n \times |\mathcal{O}|^2 \times |\mathcal{A}|)$. Die Funktion `super` mit der größten Komplexität innerhalb der `loop`-Anweisung bestimmt also die Gesamtkomplexität des Algorithmus.

Ganters Algorithmus zur Berechnung aller Begriffe und der Algorithmus in Abbildung 3.9 sind sich ähnlich: sie besitzen die gleiche asymptotische Komplexität und generieren ausgehend von einem Begriff neue Begriffe als Kandidaten, die ein bestimmtes Kriterium erfüllen sollen. Ganters Algorithmus generiert $|\mathcal{O}|$

Begriffe als Kandidaten und testen jeden Kandidaten unmittelbar nach seiner Generierung. Der Algorithmus in Abbildung 3.9 berechnet erst eine Menge von bis zu $|\mathcal{O}|$ Kandidaten, die anschließend gemeinsam auf ein Kriterium hin getestet werden. In der Praxis sind deshalb vergleichbare Laufzeiten zu erwarten.

Theorem 9 bestimmt die Objektmengen von direkten Oberbegriffen eines Begriffes (O, A), indem $(O \cup \{o\})''$ für $o \notin O$ berechnet wird. Tatsächlich genügt es $(O \cup \{o\})'$ zu berechnen, wie das nächste Theorem zeigt. Der Vorteil bei der Anwendung in einem Algorithmus wäre die Vermeidung einer komplexen Operation $'$. Möglicherweise läßt sich dieser Satz ebenfalls in einem ähnlichen Algorithmus wie in Abbildung 3.8 umsetzen, der die maximalen Mengen effizient berechnet.

Theorem 11 *Sei $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Begriffsverband und (O, A) nicht der maximale Begriff. Die Attributmengen der direkten Oberbegriffe von (O, A) sind genau die maximalen Attributmengen der Form*

$$(O \cup \{o\})' \quad o \in \mathcal{O} \setminus O$$

Eine Attributmenge X ist genau dann maximal, wenn $X \not\subset (O \cup \{o\})'$ für alle $o \in \mathcal{O} \setminus O$ gilt.

Beweis. Aus $O \cup \{o\} \supset O$ folgt $(O \cup \{o\})' \subset O' = A$ und $(O \cup \{o\})'' \supset O'' = O$. Also ist jeder Begriff der Form $((O \cup \{o\})'', (O \cup \{o\})')$ ein Oberbegriff von (O, A).

Sei (O_1, A_1) ein direkter Oberbegriff von (O, A), also $O_1 \supset O$, mit $o_1 \in O_1 \setminus O$. Der kleinste Begriff mit o_1 in seiner Objektmenge ist $\gamma(o_1) = (\{o_1\}'', \{o_1\}')$. Weil (O_1, A_1) ein direkter Oberbegriff von (O, A) ist, ist er zugleich das Supremum aus (O, A) und $\gamma(o_1)$: $(O_1, A_1) = (O, A) \vee \gamma(o_1)$ und es gilt speziell: $A_1 = A \cap \{o_1\}' = O' \cap \{o_1\}' = (O \cup \{o_1\})'$. Wäre A_1 nicht maximal, könnte (O_1, A_1) kein direkter Oberbegriff sein, also ist jeder direkte Oberbegriff von der behaupteten Form.

Sei umgekehrt $(O \cup \{o_1\})'$ mit $o_1 \in \mathcal{O} \setminus O$ maximal, also ein Oberbegriff: $(O_1, A_1) = ((O \cup \{o_1\})'', (O \cup \{o_1\})') > (O, A)$. Wenn ein Oberbegriff (O_2, A_2) mit $(O_1, A_1) > (O_2, A_2) > (O, A)$ existiert, dann läßt sich seine Attributmenge als $(O \cup \{o_2\})'$ darstellen. Dies ist ein Widerspruch zur Maximalität von A_1 und deshalb ist (O_1, A_1) ein direkter Oberbegriff von (O, A).

3.7 Operationen auf dem Begriffsverband

Die Menge aller Begriffe eines Kontextes bildet einen Verband. Für jede Menge $C \subseteq B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ von Begriffen existiert ein eindeutiges Infimum $\bigwedge C$ und ein ebenfalls eindeutiges Supremum $\bigvee C$ in $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$. Die Operationen *meet* ($\wedge$) und *join* ($\vee$) können auf einem Array des Typs *concept array* implementiert werden. Sie benötigen dafür keines der zusätzlichen Felder, die ihm Abschnitt 3.6.1 durch die Funktion `fillLattice` (im Algorithmus-I) berechnet werden, sondern benutzen nur die Objekt- und Attributmengen der Begriffe. Dies bedeutet, daß

die Verbandsstruktur nicht explizit mit Algorithmus I oder II aus den vorhergehenden Abschnitten berechnet werden muß, sondern lediglich die Menge aller Begriffe. Die Algorithmen dieses Abschnitts setzen dabei eine lektische Sortierung der Begriffe voraus.

Der Hauptsatz (Theorem 3, Seite 9) der Begriffsanalyse gibt vor, wie Infimum und Supremum von Begriffsmengen berechnet werden:

$$\bigwedge_{t\in T}(O_t, A_t) = \left(\bigcap O_t, \left(\bigcup A_t\right)''\right) \quad \text{und} \quad \bigvee_{t\in T}(O_t, A_t) = \left(\left(\bigcup O_t\right)'', \bigcap A_t\right)$$

Das Infimum zweier Begriffe c_i und c_j ist $c_i \wedge c_j = c_k$. Es kann entweder direkt mit Hilfe des Hauptsatzes und des Kontextes berechnet werden, oder alleine in der Menge aller Begriffe bestimmt werden. Die zweite Option ist sinnvoll, wenn mit jedem Begriff weitere Informationen verbunden sind, die nicht unmittelbar aus c_i und c_j abgeleitet werden können. Die vorberechnete Verbandsstruktur aus dem Abschnitt 3.6.1 ist dafür ein Beispiel.

Das Infimum von zwei Begriffen ist ein Unterbegriff beider Ausgangsbegriffe: die Objektmenge von c_k ist der Schnitt der Objektmengen von c_i und c_j. Wenn das lektisch geordnete Array c die Begriffe $c[i]$, $c[j]$ und $c[k]$ enthält, dann muß nach Theorem 7 gelten: $0 \leq k \leq \min(i, j)$ [4]. Ein Begriff mit der gesuchten Objektmenge *muß* wegen der Verbandseigenschaft in dem Intervall $0 \leq k \leq \min(i, j)$ existieren. In einem lektisch geordneten Array sind die Begriffe nach ihren *Objektmengen* lektisch aufsteigend geordnet (Definition 11). Dies kann ein Algorithmus zum Suchen des Begriffes mit der gewünschten Objektmenge nutzen: er verwendet die Objektmenge $o[k] = o[i] \cap o[j]$ als Schlüssel in einer binären Suche [54] durch das Intervall $0 \ldots \min(i, j)$ des Arrays. Die Komplexität dieses Algorithmus, der das Infimum zweier Begriffe in einem lektisch geordneten Array aller Begriffe ermittelt, ist $O(|\mathcal{O}| \times \log\ n)$ wobei $n = |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$.

Die Implementierung in Objective Caml als Funktion `meet` besitzt den Typ *int → int → concept array → int*. Sie bestimmt den Index des Infimums aus den Indizes der beiden Ausgangsbegriffe und verwendet für die binäre Suche die Funktion `bsearch`. Ihre Parameter sind die Vergleichsfunktion (`cmp`), die Ober- und Untergrenze des zu durchsuchenden Bereiches und das durchsuchte Array selber.

```
let meet a b lattice =
  let aX        = Concept.x lattice.(a).concept in
  let bX        = Concept.x lattice.(b).concept in
  let meetX     = Bitset.functionalIntersect aX bX in
  let min a b   = if a < b then a else b in

  let cmp entry =
```

[4] der kleinste Index des Arrays ist 0

```
    let entryX  = (Concept.x entry.concept) in
      if entryX = meetX then
        0
      else if Bitset.isSmaller meetX entryX  then
        -1
      else
        1
in
  match bsearch cmp 0 (min a b) lattice with
      Some index -> index
    | None       -> assert false (* the meet must exist! *)
```

Das Dual zum Infimum ist das Supremum einer Menge von Begriffen. Es kann ebenso wie das Infimum direkt mit Hilfe des Hauptsatzes und des Kontextes, oder ohne den Kontext in der Menge aller Begriffe ermittelt werden. Das Supremum $c_k = c_i \vee c_k$ von zwei Begriffen c_i und c_j ist ein Oberbegriff von c_i und c_j, und sein Index k befindet sich in einem lektisch geordneten Array in dem Intervall $\max(i, j) \leq k \leq n - 1$, wobei $n = |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$. Der Schlüssel zur Suche der Attributmenge $a[k]$ des Infimums, und damit zur Suche von k, ist die Attributmenge $a[k] = a[i] \cap a[j]$. Die lektische Ordnung des Arrays basiert auf der lektischen Ordnung der Objektmengen – das Array ist in Bezug auf Attributmengen aber ungeordnet. Eine Suche nach der Menge $a[k]$ kann deshalb keine Ordnung ausnutzen und muß das Intervall $\max(i, j) \leq k \leq n - 1$ linear durchsuchen. Die Zeitkomplexität für die Berechnung des Infimums zweier Begriffe ist also $O(|\mathcal{A}| \times n)$ mit $n = |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$. Die Implementierung der zugehörigen Funktion `join` ähnelt der Implementierung von `meet` mit dem Unterschied, daß eine lineare Suchfunktion verwendet wird.

3.8 Alternativen

Bei einem Vergleich der Operationen *meet* und *join* fällt ihre Asymmetrie auf. Das Infimum einer Menge läßt sich effizienter bestimmen als das Supremum, weil die lektische Ordnung von Begriffen auf der lektischen Ordnung ihrer Objektmengen basiert. Die ungeordneten Attributmengen verhindern dagegen eine effiziente Suche. Dieser Abschnitt untersucht Lösungen, die diese Asymmetrie überwinden.

3.8.1 Ablage in einer Hashtabelle

Statt eines Arrays kann eine Hashtabelle alle berechneten Begriffe aufnehmen ([11], Kapitel 12). Dabei werden sowohl die Objekt- als auch die Attributmenge eines Begriffes über eine Hashfunktion auf den zugehörigen Begriff abgebildet. Diese Datenstruktur ist symmetrisch in ihrem Verhalten in Bezug auf die *meet-*

und *join*-Operation. Weil sie keine Eigenschaften der Begriffe ausnutzt, ist sie weniger elegant als ein Array und erfordert mehr Speicherplatz. Ihr Hauptnachteil ist, daß sie nur für Ablage des Begriffsverbandes im Hauptspeicher geeignet ist – ihre verzeigerte Struktur erschwert eine Implementierung als externe Datenstruktur. Im Gegensatz dazu sind alle bislang vorgestellten Datenstrukturen geeignet, Begriffsverbände zu berechnen und zu verwenden, die die Größe des Hauptspeichers überschreiten und persistent abgelegt werden. Wegen des beträchtlichen algorithmischen Aufwands zur Berechnung von Begriffen ist ihre persistente Ablage ein Entwurfsaspekt ihrer Datenstrukturen. Das bislang verwendete Array zur Ablage des Begriffsverbandes kann eine Datei sein, die sequentiell beschrieben wird und durch die Funktionen `fillLattice` (Abschnitt 3.6), `meet` und `join` bearbeitet wird.

Das kompakte Array des Begriffsverbandes wurde in der *TkConcept*-Implementierung [67] der Begriffsanalyse zur persistenten Ablage verwendet. Der Verband wird allerdings im Hauptspeicher berechnet und für begriffliche Operationen auch vollständig in den Hauptspeicher geladen. Eine ausschließlich externe Speicherung des Verbandes wäre aber möglich. *TkConcept* ist eine modulare Erweiterung der *Tool Command Language* [81] für Begriffsanalyse. Sie stellt die Berechnung des Begriffsverbandes, seine persistente Ablage und die wichtigsten Operationen auf Begriffsverbänden bereit. Die *Tool Command Language* ist eine interpretierte Sprache zur Steuerung von Applikationen.

3.8.2 Sortierung der Attributmengen I

Wenn in einer Applikation besonders häufig das Supremum von Begriffen bestimmt wird, das Infimum aber selten, kann das Array aller Begriffe lektisch nach seinen Attributmengen geordnet werden, statt nach seinen Objektmengen. Dies kann entweder nachträglich durch eine explizite Sortierung erreicht werden, oder schon bei der Berechnung aller Begriffe. Der Algorithmus von Ganter arbeitet statt auf Objektmengen auch auf Attributmengen und erzeugt dann lektisch geordnete Attributmengen. Auf diese Weise wird die Asymmetrie zugunsten der *join*-Operation verschoben, bleibt aber bestehen.

Wenn sowohl die *meet*-, als auch die *join*-Operation benötigt wird, können beide effizient durch redundante Datenstrukturen implementiert werden: zusätzlich zu einer lektisch sortierten Liste von Objekt-Attribut-Paaren wird eine sortierte Liste von Attributmengen angelegt, die Verweise auf die zugehörigen Objektmengen enthält. Die effiziente Implementierung durch Redundanz wurde bereits in Abschnitt 3.3 zur Implementierung von binären Relationen vorgeschlagen.

3.8.3 Sortierung der Attributmengen II

Ganters Algorithmus benutzt eine totale Ordnung $\prec$ auf den Objekt- und Attributmengen $\mathcal{O}$ und $\mathcal{A}$ und definiert mit ihrer Hilfe die lektische Ordnung für

i	O mit $\prec$					A mit $\prec$							A mit $\prec^*$						
	1	2	3	4	5	1	2	3	4	5	6	7	5	1	3	2	4	7	6
1						×	×	×	×	×	×	×	×	×	×	×	×	×	×
2					×			×		×	×		×		×				×
3				×		×			×			×		×			×	×	
4		×					×	×			×				×	×			×
5		×			×		×				×					×			×
6	×		×						×		×	×					×	×	×
7	×		×	×					×			×					×	×	
8	×	×	×		×						×								×
9	×	×	×	×	×														

Tabelle 3.5: Eine neue Ordnung $\prec^*$ der Attribute $\mathcal{A}$ führt zu einer lektischen Ordnung der Attributmengen

Mengen (Definition 10). Die Ordnung der Grundmengen $\mathcal{O}$ und $\mathcal{A}$ entsteht durch eine einfache Anordnung ihrer Element: $o_1 \prec o_2 \prec o_3 \prec \ldots$ und $a_i \prec a_{i+1}$.

Der Algorithmus verwendet ausschließlich die Ordnung von Objekten und die lektische Ordnung von Objektmengen – die Ordnung der Attribute aber nicht. Nachdem alle Begriffe bestimmt sind, kann durch eine nachträgliche Änderung der Ordnung der Attribute erreicht werden, daß Attributmengen ebenso wie Objektmengen lektisch geordnet sind. Alleine durch eine passend gewählte Ordnung der Attribute würden dann sowohl Objekt- als auch Attributmengen in dem Array geordnet sein und Infimum und Supremum könnten durch eine effiziente Suche in dem Array implementiert werden.

Ein Beispiel soll die Idee illustrieren, wie durch die Umordnung von Attributen lektisch geordnete Attributmengen entstehen können. Tabelle 3.5 zeigt die Begriffe des Kontextes aus Tabelle 3.1(b) (Seite 18), der bereits zur Erläuterung von Ganters Algorithmus diente. Auf der linken Seite stehen die Objekt- und Attributmengen, wie sie der Algorithmus bestimmt. Objektmengen sind lektisch aufsteigend geordnet, Attributmengen sind ungeordnet und sowohl Objekte als auch Attribute sind auf die zuvor definierte Weise total geordnet.

Wenn die Attributmenge $\mathcal{A}$ mit einer neuen Ordnung $\prec^*$ versehen wird und die Spalten der Attribut-Tabelle entsprechend geordnet werden, ergeben sich die Attributmengen auf der rechten Seite von Tabelle 3.5. Die neue Ordnung definiert $a_5 \prec^* a_1 \prec^* a_3 \prec^* \cdots \prec^* a_6$; die Ordnung der Objekte bleibt unverändert. Wie bisher sind die Objektmengen der Begriffe lektisch aufsteigend geordnet (Definition 10): das kleinste Element, in dem sich $o[i]$ und $o[i+1]$ unterscheiden, befindet sich in $o[i+1]$. Mit der neuen Ordnung $\prec^*$ sind die Attributmengen lektisch absteigend geordnet: das kleinste Element, in dem sich $a[i]$ und $a[i+1]$ unterscheiden befindet sich in $a[i]$, also $a[i+1] \prec^* a[i]$. Die neue lektische Ordnung könnte in einem Algorithmus zur Berechnung des Supremums von Begriffen ausgenutzt werden, wenn die Attributmenge des Supremums durch eine binäre

Suche in dem Array ermittelt wird.

Es bleibt die Frage, wie eine passende Ordnung $\prec^*$ definiert werden kann und ob sie für jeden Begriffsverband existiert. Die in dem Beispiel angegebene Ordnung ist aus dem Begriffsverband selbst abgeleitet worden:

Definition 13 (Ordnung $\prec^*$) *Sei $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Kontext und $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ der zugehörige Begriffsverband. Auf den Mengen $\mathcal{O}$ und $\mathcal{A}$ sei eine totale Ordnung $\prec$ definiert, die eine lektische Ordnung $\prec$ zwischen Teilmengen von $\mathcal{O}$, $\mathcal{A}$, sowie den Begriffen gemäß Definition 10 induziert. Die Ordnung $\prec^*$ auf den Mengen $\mathcal{O}$ und $\mathcal{A}$ wird wie folgt definiert; $o_1, o_2 \in \mathcal{O}$ und $a_1, a_2 \in \mathcal{A}$.*

$$\begin{aligned} o_1 \prec^* o_2 \quad &\Longleftrightarrow \quad o_1 \prec o_2 \\ a_1 \prec^* a_2 \quad &\Longleftrightarrow \quad \mu(a_1) \prec \mu(a_2) \text{ oder} \\ &\qquad\quad \mu(a_1) = \mu(a_2) \text{ und } a_1 \prec a_2 \end{aligned}$$

Die neue Ordnung $\prec^*$ übernimmt die bestehende Ordnung von Objekten und verändert lediglich die Ordnung von Attributen und damit die lektische Ordnung von Attributmengen. Weil die lektische Ordnung von Begriffen (Definition 11) allein auf der Ordnung von Objektmengen basiert, berechnet Ganters Algorithmus auch unter Verwendung von $\prec^*$ Begriffe in einer lektisch aufsteigenden Folge.

Hinter der neuen Ordnung $\prec^*$ steht die Beobachtung, daß Unterbegriffe mehr Attribute tragen als ihre Oberbegriffe. Oberbegriffe sind lektisch größer als ihre Unterbegriffe und deshalb nimmt die Zahl der Attribute zu lektisch größeren Begriffen hin ab. Für eine lektische Ordnung für Attributmengen bietet es sich daher an, eine lektisch absteigende Ordnung der Begriffe im Array erzielen zu wollen. Deshalb müssen Attribute, die nur in lektisch kleinen Begriffen auftreten, bezüglich der Ordnung $\prec^*$ möglichst klein sein.

Betrachtet man die Begriffe $c_1 \prec c_2, \prec \ldots$ in lektisch aufsteigender Folge, so sind einige von ihnen der *Attributbegriff* (Theorem 4, Seite 4) eines Attributes a: $\mu(a) = c$. Jeder lektisch größere Begriff als c kann a nicht in seiner Attributmenge enthalten. Deswegen wird die lektische Ordnung der Attributbegriffe für die Definition der neuen Ordnung verwendet. Wenn ein Begriff Attributbegriff mehrerer Attribute ist, wird auf die ursprüngliche Ordnung zurückgegriffen. Man erhält diese Ordnung anschaulich, wenn man die Begriffe des Begriffsverbandes (Abbildung 3.10) in lektisch aufsteigender Folge besucht und dabei die als Markierungen verwendeten Attribute einsammelt.

Die Ordnung $\prec^*$ erzwingt nicht in jedem Fall eine lektisch absteigende Sortierung der Attributmengen. Es lassen sich Beispiele angeben, in denen die lektisch aufsteigende Sortierung der Begriffe nicht genau mit einer lektisch absteigenden Sortierung ihrer Attributmengen einhergeht. Meistens sind es nur wenige Begriffe, die die Ordnung stören. Durch Einführung von geeigneten und für die Anwendung nicht benötigten Pseudo-Attributen ließe sich die gewünschte Ordnung wieder herstellen, um so die zuvor beschriebenen Vorteile zu nutzen.

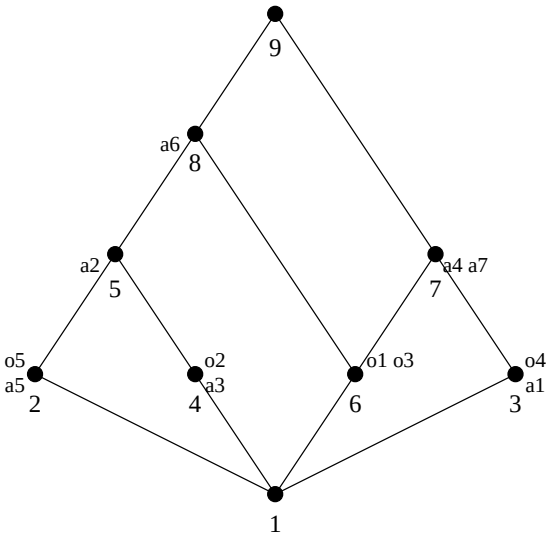

Abbildung 3.10: Begriffsverband des Beispiels. Die Zahlen unten den Begriffen korrespondieren mit i in Abbildung 3.5 und der lektischen Ordnung der Begriffe.

Die vorgeschlagene Ordnung $\prec^*$ ist also keine allgemeine Lösung für das Problem, eine Sortierung für die Attributmengen des Begriffsverbandes zu finden. Ob überhaupt eine allgemeine Lösung existiert und welche Form sie besitzt, ist eine offene Frage.

3.8.4 Kodierung der Ordnung durch kompakte Bitvektoren

Typ- und Vererbungshierarchien bilden in objektorientierten Programmiersprachen oftmals (Halb-) Verbände. Eine effiziente Implementierung des Verbandes und der *meet*-Operation ist deshalb dort ebenfalls von Interesse. Aït-Kaci et al. geben in [4] verschiedene Implementierungen von Halbordnungen und Untersuchungen über die Effizienz ihrer Operationen an.

Aït-Kaci et al. untersuchen zunächst Repräsentationen für geordneten Elemente, die ihre Ordnung implizit enthalten. Die Grundidee ist, jedes Element durch einen Bitvektor zu repräsentieren, der ein gesetztes Bit für jedes im Sinne der Ordnung kleinere oder gleich großes Element enthält. Das Infimum von zwei Elementen ist dann eindeutig durch den Schnitt dieser Vektoren bestimmt und kann effizient durch eine binäre Und-Operation implementiert werden. Die entstehenden Bitvektoren sind durch diese naive Konstruktion größer als nötig, um die Ordnung und ihre Elemente zu repräsentieren. Kompaktere Kodierungen, die weiterhin eine effiziente *meet*-Operation erlauben, nehmen deshalb den Hauptteil in [4] ein. Weil Typhierarchien, wenn überhaupt, nur Halbverbände sind, beschränkt sich die Diskussion auf Halbverbände und Mengen ohne Verbandseigenschaften.

In der array-basierten Implementierung eines Begriffsverbandes werden die Eigenschaften der Objekt- und Attributmengen genutzt, um die implizite Verbandsstruktur zu erkennen und die *meet*- und *join*-Operationen zu implementieren. Bei der Anwendung der in [4] vorgeschlagene Techniken auf Begriffsverbände bieten sich zwei Möglichkeiten an. Erstens: jeder Begriff wird zusätzlich mit einem (kompakten) Code versehen, der seine Ordnung definiert und die effiziente Implementierung von *meet* und *join* erlaubt. Und zweitens: die vorhandenen Objekt- und Attributmengen werden durch kompaktere Codes ersetzt, die dann die Basis für Operationen auf dem Begriffsverband bilden.

Ein Begriffsverband mit um Codes ergänzten Begriffen ist einer Implementierung mit Hashtabellen (Abschnitt 3.8.1) ähnlich: statt der den Begriffen impliziten Struktur beschreibt eine externe Struktur den Begriffsverband. Eine Platzersparnis, wie sie durch die kompakten Codes erzielt werden soll, wird damit nicht erreicht, da im Gegenteil der Anteil der Infrastruktur wächst.

Statt neue Codes hinzuzufügen könnten die bestehenden Objekt- und Attributmengen kompaktifiziert werden, und damit eine platz- und zu geringem Maße zeiteffizientere Realisierung erreicht werden. Damit würden aber zugleich Informationen über die Objekte und Attribute jedes Begriffes verlorengehen, was für die meisten Anwendungen nicht tragbar ist. Die in [4] vorgeschlagenen Kompaktifizierungen sind also nur in Spezialfällen sinnvoll.

Eine direkte Anwendung der von Aït-Kaci et al. vorgeschlagenen Techniken zur Repräsentation von Begriffsverbänden bringt offensichtlich kaum Vorteile mit sich. Erschwerend kommt hinzu, daß die von ihnen vorgeschlagenen Techniken zunächst auf vollständige Verbände erweitert werden müßten, da sie bislang nur für Halbverbände beschrieben sind. Im Gegensatz zu Hashtabellen sind sie auch als externe Datenstruktur in Dateien geeignet, da Begriffsverbände darin kompakte, unverzeigerte Speichereinheiten sind. Sowohl die in Abschnitt 3.6 vorgeschlagenen Implementierung, als auch die von Aït-Kaci et al., verwendet Bitvektoren und ihren Schnitt zur Berechnung des Infimums in einem Verband. Die weitere Kompaktifizierung dieser Bitvektoren wie von Aït-Kaci et al. vorgeschlagen läßt im Allgemeinen aber keine Vorteile erwarten.

3.9 Blockrelation

Bei der Analyse von Kontexten in einer Anwendung können sehr große Begriffsverbände entstehen, die nur schwer zu interpretieren sind. Dann kommt häufig der Wunsch auf, diese sinnvoll zu vereinfachen, um sie verständlicher zu machen. Eine Möglichkeit dafür stellen Blockrelationen dar [32]. Sie vereinfachen allerdings einen Verband nicht direkt, sondern seinen Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$. Durch Aufnahme neuer Elemente in $\mathcal{R}$ wird die Kontexttabelle „angedickt“, die dann einen kleineren Verband als der ursprüngliche Kontext besitzt. Das Hinzufügen von Kreuzen in die Kontexttabelle geschieht systematisch – dieser Abschnitt beschreibt einen

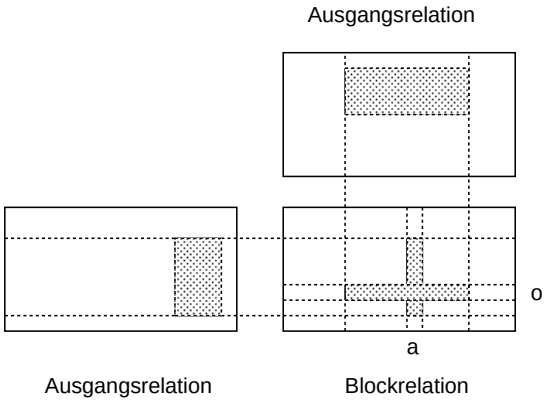

Abbildung 3.11: Kennzeichen einer Blockrelation $\mathcal{S}$: für jede Zeile und Spalte existiert in der Ausgangsrelation $\mathcal{R}$ ein Begriff.

entsprechenden Algorithmus.

In der Kontexttabelle einer Blockrelation $\mathcal{S}$ eines Kontextes $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ist jede Zeile und Spalte eine Attribut- oder Objektmenge eines Begriffes in $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$; Abbildung 3.11 skizziert die Situation.

Definition 14 (Blockrelation) *Eine* Blockrelation *eines Kontextes* $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ *ist eine Relation* $\mathcal{S} \subseteq \mathcal{O} \times \mathcal{A}$*, die folgenden Bedingungen genügt:*

1. $\mathcal{R} \subseteq \mathcal{S}$,
2. *für jedes Objekt* $o \in \mathcal{O}$ *ist* $\{o\}^{\mathcal{S}}$ *ein Inhalt (Attributmenge) in* $(\mathcal{O}, \mathcal{A}, \mathcal{R})$,
3. *für jedes Attribut* $a \in \mathcal{A}$ *ist* $\{a\}^{\mathcal{S}}$ *ein Umfang (Objektmenge) in* $(\mathcal{O}, \mathcal{A}, \mathcal{R})$.

Dabei bezeichnet $\{o\}^{\mathcal{S}}$ *die Attributmenge* $\{o\}'$ *bezogen auf die Relation* $\mathcal{S}$.

Eine Blockrelation entsteht aus einem Kontext $\mathcal{R}$, indem ein (zunächst beliebiges) neues Element $(o, a) \notin \mathcal{R}$ in die Relation aufgenommen wird: $R_1 = \mathcal{R} \cup \{(o, a)\}$. Die neue Relation R_1 verletzt im Allgemeinen die letzten beiden Punkte in Definition 14. Deswegen wird in der Kontexttabelle von $\mathcal{R}_1$ die Zeile o_1 durch $\{o\}^{\mathcal{R}\mathcal{R}}$ ersetzt und Spalte a durch $\{a\}^{\mathcal{R}\mathcal{R}}$, so daß eine neue Relation $\mathcal{R}_2$ entsteht. Diese Ersetzungen führen zwar dazu, daß die Zeile o und a den Anforderungen an eine Blockrelation genügen, gleichzeitig können sie aber in anderen Zeilen und Spalte wieder neue Elemente einfügen, so daß diese wieder entsprechend ergänzt werden müssen. Dieser Prozeß muß fortgesetzt werden, bis sein Fixpunkt erreicht ist: $\mathcal{R}_i = \mathcal{R}_{i+1} = \mathcal{S}$ ist eine Blockrelation. Abbildung 3.12 skizziert den Prozeß der *Hüllenbildung* für das Element (o, a) nochmals.

Ein Kontext besitzt eine Menge $\mathcal{S}_1, \mathcal{S}_2, \ldots$ von Blockrelationen. Die Blockrelationen sind durch Mengeninklusion $\subseteq$ (halb-) geordnet, so daß die Menge aller Blockrelationen einen Verband bildet. Darin ist die Ausgangsrelation $\mathcal{R}$ das

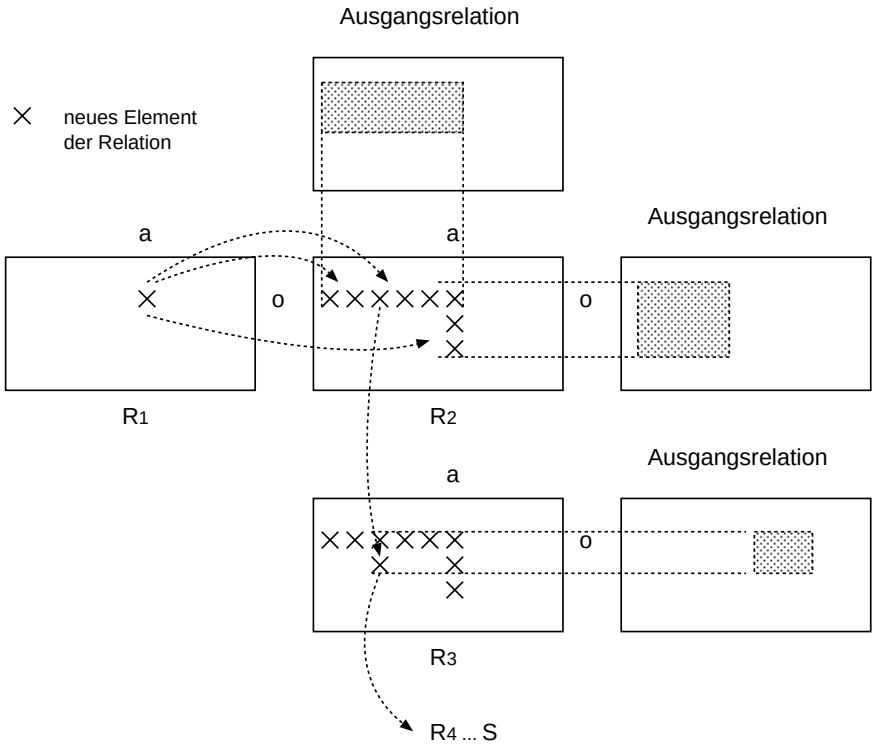

Abbildung 3.12: Hüllenbildung nach dem Einfügen eines Elementes zur Bestimmung einer Blockrelation.

kleinste Element, und $\mathcal{O} \times \mathcal{A}$ (mit vollständig gefüllter Kontexttabelle) das größte Element.

Eine Variante von Ganters Algorithmus (Abschnitt 3.1.2) berechnet die Menge aller Blockrelationen. Dazu wird eine zweidimensionale Kontexttabelle der Größe $m \times n$ als binärer Vektor der Länge $m \times n$ aufgefaßt und die Operation $'$ wird durch die Hüllenbildung ersetzt. Die für Ganters Algorithmus vorgenommenen Definition der lektischen Ordnung kann dann unverändert für Relationen verwendet werden. Zur Illustration wird der Algorithmus zur Berechnung der jeweils nächsten Blockrelation nachfolgend durch Skizzen von Kontexttabellen beschrieben.

Der Ausgangspunkt zur Berechnung der lektisch nächsten Blockrelation $\mathcal{S}_{i+1}$ ist die zuletzt berechnete Blockrelation $\mathcal{S}_i$; dies ist beim Start des Algorithmus die Ausgangsrelation $\mathcal{R} = \mathcal{S}_0$. Dann wird ein neuer Eintrag (o, a) hinzugefügt und die entstandene Relation $\mathcal{R}_1$ zu einer Blockrelation durch Hüllenbildung vervollständigt. Der Eintrag wird in der Kontexttabelle an dem am weitesten rechts und unten stehenden freien Eintrag vorgenommen (Abbildung 3.13 (a)). Sollte ein solcher nicht existieren, ist die Kontexttabelle vollständig gefüllt und keine weitere Blockrelation existiert.

Durch die Vervollständigung zur Blockrelation werden weitere Elemente in die

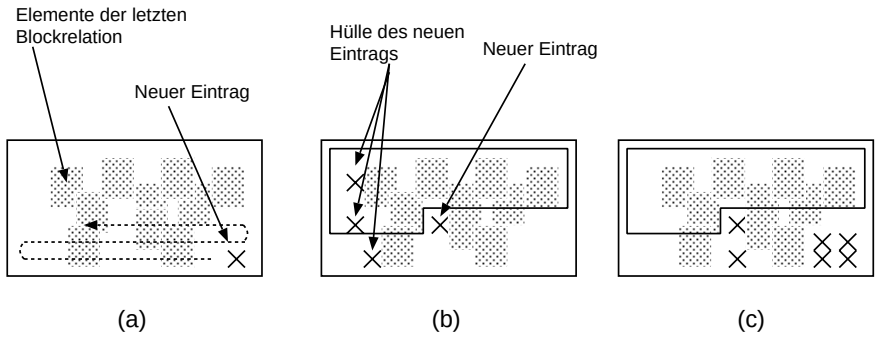

Abbildung 3.13: Suche der lektisch nächsten Blockrelation

Relation aufgenommen. Die gesuchte lektisch nächste Blockrelation entsteht nur, wenn keines der zusätzlichen Elemente links oder oberhalb des zuerst gesetzten Elementes auf einen freien Platz eingetragen werden muß. In Abbildung 3.13 (b) ist diese Region durch einen Rahmen gekennzeichnet; die dort skizzierte Blockrelation ist also nicht die gesuchte lektisch nächste. Dagegen ist die Blockrelation in Abbildung 3.13 (c) die gesuchte lektisch nächste Blockrelation.

Wenn eine Blockrelation verworfen werden muß, wird ein nächster Versuch unternommen, und zwar ausgehend von der letzten gültigen Blockrelation. Der erste Eintrag wird jetzt auf den ersten freien Platz links und oberhalb des zuletzt versuchten Eintrages vorgenommen. Die Plätze für die ersten Einträge der verschiedenen Versuche liegen also entlang der in Abbildung 3.13 (a) eingetragenen Schlangenlinie. Je weiter der erste Eintrag oben und links plaziert ist, um so kleiner wird die verbotene Zone und damit steigt die Wahrscheinlichkeit, daß darin keine Einträge vorgenommen werden müssen. Im Extremfall müssen alle freie Einträge der Relation durchprobiert werden, um die nächste Blockrelation zu finden. Für die nächste Blockrelation beginnt der Algorithmus wieder in der rechten unteren Ecke der Kontexttabelle.

Die Berechnung aller Blockrelationen ist aufwendig; eine genaue Abschätzung des Aufwandes aber nicht trivial: bei der Suche der nächsten Blockrelation müssen bis zu $|O| \times |\mathcal{A}|$ Kandidaten durch jeweils eine Fixpunktoperation bestimmt werden. Jede Fixpunktoperation muß jede Zeile und Spalte der Kontexttabelle möglicherweise mehrfach durch die $'$-Operation mit einem Aufwand von $O(|\mathcal{O}| \times |\mathcal{A}|)$ vervollständigen. Ohne Untersuchungen über den Aufwand der Fixpunktberechnung kann keine sinnvolle Gesamtkomplexität abgegeben werden.

Für die Implementierung eines Algorithmus zur Bestimmung aller Blockrelationen bieten sich zwei Optimierungen an: wie schon bei Ganters Algorithmus ist jeder berechnete Kandidat für die nächste Blockrelation eine Blockrelation, nur möglicherweise nicht die gesuchte nächste. Ein Ablage in einem Cache erspart später ihre Neuberechnung – vergleiche Abschnitt 3.5. Die zweite Optimierung besteht darin, die Berechnung der Hülle eines Elementes sofort abzubrechen, wenn

dabei ein Element aufgenommen werden muß, das links oder oberhalb des ersten Elementes liegt.

Kapitel 4

Komponentenbasierte Software-Wiederverwendung

Ende der 60er Jahre hat man erkannt, daß ein detailiertes Wissen über das Programmieren-im-Kleinen – die Spezifikationen von Software, Algorithmen, Datenstrukturen, Semantik von Programmiersprachen – wenig bei der Erstellung von komplexen Softwaresystemen hilft. Damit entstand das neue Teilgebiet *Software Engineering*, das untersucht, wie komplexe Software durch eine Gruppe von Programmierern zuverlässig und wirtschaftlich erstellt werden kann [35]. Beim Programmieren-im-Großen stehen Aspekte der Softwarequalität, Produktivität, Wartbarkeit, Modularisierung und Versionierung im Vordergrund des Interesses. Die Erkenntnisse darüber sind in die Entwicklung von Programmiersprachen [115, 106, 38, 75] eingeflossen und haben neben anderen zu einem stetigen Anstieg der Produktivität bei der Programmentwicklung beigetragen [118]. So wird Programmieren auch nicht länger als eine Kunst [55], sondern als ein Ingenieurleistung verstanden, die in einem industriellen Rahmen erbracht werden kann.

Seit Beginn des Software-Engineering wird die *Wiederverwendung von Software* als eine zentrale Möglichkeit gesehen, hochwertige Software wirtschaftlich zu erstellen. Unter Software-Wiederverwendung werden Techniken der Programmerstellung zusammengefaßt, die auf vorhandenes Wissen aus früheren Softwareprojekten *systematisch* zurückgreifen [56, 72]. Dieses Wissen kann die Form von Dokumenten oder automatisierten Prozessen besitzen, die aus allen Phasen der Softwareerstellung stammen können: Entwurf, Spezifikation, Codierung, Integration, Test, Qualitätssicherung. Softwarewiederverwendung bezieht sich also nicht alleine auf die Wiederverwendung von Programmcode und schließt andererseits das informale Wissen aus, das Entwickler durch Erfahrung gewonnen haben, da dieses nicht in Form von Dokumenten vorliegt und deswegen nicht systematisch genutzt werden kann. Softwarewiederverwendung bei der Erstellung von Software überspringt durch die Verwendung erprobter Dokumente und Prozesse die Details des Programmieren-im-Kleinen und konzentriert sich auf die Probleme bei der Erstellung von Software-*Systemen*. Das Hauptaugenmerk liegt nicht mehr auf

algorithmischen Details, sondern auf der Skalierbarkeit der Entwicklungsarbeit [72]. Aus betriebswirtschaftlicher Sicht ist eine erfolgreiche Softwarewiederverwendung ebenfalls wünschenswert: Wissen über Software und ihre Entwicklung manifestiert sich dann in Dokumenten und geht so aus den Köpfen der Softwareentwickler in den Besitz eines Betriebes über, der damit seine Abhängigkeit von dem nur bei seinen Mitarbeitern vorhandenen Wissen verringert. Insgesamt ist eine erfolgreiche Wiederverwendung von Software ein lang gehegter Traum, sowohl von Programmierern, weil der sie von Routinearbeiten entlastet, als auch von ihren Managern.

Obwohl die Bedeutung von Software-Engineering und speziell der Wiederverwendung von Software allgemein anerkannt ist, ist die systematische und wirtschaftlicher Entwicklung von komplexer Software immer noch schwierig. Softwareprojekte werden gar nicht, oder zu spät bei zu hohen Kosten abgeschlossen oder erfüllen die an sie gestellten Erwartungen nicht. Die erfolgreiche Wiederverwendung von Software bildet in der Praxis eine Ausnahme; weil sie nach wie vor ein großes Potential für qualitätsvolle und wirtschaftliche Software verspricht, gibt es ein reges wissenschaftliches Interesse für die Gründe des Scheiterns bisheriger Versuche und eine Suche nach neuen Wegen [56]. Mili et al. nennen neben der Jugendlichkeit des Software-Engineerings als wissenschaftliche Disziplin eine unterentwickelte Ausbildung und ungenügendes Management, neben technischen Unzulänglichkeiten, als Gründe für niedrige Wiederverwendungsraten [72].

Wissen über die Konstruktion von Software kann prinzipiell auf zwei verschiedene Weisen systematisch wiederverwendet werden: erstens, indem die bei der Konstruktion von Software entstehenden Dokumente abgelegt und später für neue Projekte wiederverwendet werden. Dies ist ein *komponentenbasierter* Ansatz. Und zweitens, indem der Prozeß, in dem Dokumente entstehen, automatisiert wird, und bei Bedarf wiederholt werden kann. Dies ist der *prozeßbasierte* Ansatz. Dokumente umfassen in diesem Zusammenhang alle Arten, die bei der Erstellung von Software von Bedeutung sind: Entwürfe, Spezifikationen, Protokolle, Code, Testfälle, Dokumentation. Technisch wird ein prozeßbasierter Ansatz durch Generatoren implementiert, die die gewünschten Dokumente aus einer abstrakten Beschreibung generieren. Komponentenbasierte Ansätze sind die gebräuchlichsten Methoden der Wiederverwendung und werden durch Komponentenbibliotheken realisiert. Sie unterscheiden sich in der Granularität der abgelegten Komponenten und in der Organisation der Bibliothek. Der nächste Abschnitt präsentiert beide Ansätze mit Beispielen.

4.1 Methodenübersicht

4.1.1 Generatoren

Ein *Generator* erzeugt in einem prozeßbasierten Ansatz Quellcode oder Applikationen aus einen Spezifikation, ähnlich einem Compiler [46]. Im Unterschied zu universellen Programmiersprachen hat ein Generator einen spezialisierten Anwendungsbereich, der mit deskriptiven (im Gegensatz zu operationalen) Spezifikationen beschrieben werden kann. Das Verhältnis der Größen von Spezifikation und erzeugtem Code ist deshalb größer als bei universellen Programmiersprachen. Bekannte Beispiele für Generatoren sind Lex und Yacc zur Generierung von Scannern und Parsern im Compilerbau [2], StateMate zur Spezifikation reaktiver Systeme [42] und interaktive Werkzeuge zur Erstellung von graphischen Benutzungsoberflächen [109]. Generatoren können nicht nur programmiersprachliche Quelltexte erzeugen, sondern ganze Applikationen: das PSG-System [5] zum Beispiel erstellt eine Programmierumgebung, bestehend aus einem syntaxgesteuerten Editor und einem Interpreter, aus einer formalen Definition von Syntax und Semantik einer Programmiersprache. Generatoren bringen nicht alleine unter dem Aspekt der einfachen Erzeugung von Programmcode Vorteile im Entwicklungsprozeß, sondern erlauben durch ihre Abstraktion auch Nicht-Programmierern am Designprozeß teilzunehmen [79]. So viele Vorteile der generative Ansatz besitzt, so schwierig ist er auch zu realisieren [90]. Generatoren sind nur für vollständig verstandene Anwendungsbereiche möglich. Dies setzt voraus, daß ihnen eine Theorie zu Grunde liegt – wie im Falle von Generatoren im Compilerbau – oder sie konzeptionell einfache, aber manuell mühsame Aufgaben übernehmen, wie im Falle von Codegeneratoren für Benutzungsoberflächen oder Werkzeugen, die eine Software-Design-Methodologie wie die Unified Modeling Language [8] unterstützen.

4.1.2 Libraries und Toolkits

Softwarebibliotheken (*Libraries*), und *Toolkits* als ihre objektorientierten Entsprechungen, sind Sammlungen programmiersprachlicher Module, Klassen, Funktionen und Methoden, die gemeinsam eine Basisfunktionalität bereitstellen. Beispiele dafür sind die Schnittstelle zu einem Betriebssystem, die C++ I/O-Stream-Bibliothek [106], die Standard Template Library [78] oder das X11-Toolkit [22]. Toolkits und Libraries sind so allgemein, daß sie nur einen geringen Einfluß auf die Architektur einer Applikation nehmen. Unter den komponentenbasierten Ansätzen besitzen sie entsprechend die feinste Granularität. Sie fassen häufig benötigte Funktionalität zusammen und sind die verbreiteste Form der Wiederverwendung von Programmcode. Ein Grund dafür ist, daß die Implementierungen vieler Programmiersprachen Programmteile als Module in Bibliotheken auslagern können und ebenso einen Mechanismus für ihre Einbindung bereitstellen. Bibliotheken fügen sich technisch also oftmals nahtlos in die vorhandene Infrastruk-

tur ein. Programmbibliotheken können sowohl als übersetzter Objektcode, als auch als Quelltexte vorliegen. Bekannte Programmiersprachen mit Bibliotheksunterstützung sind C [53], C++ [106], Modula-2 [115], Perl [113], SML [75], Java [38] und CommonLISP [105].

4.1.3 Frameworks

Frameworks sind ebenso wie Toolkits/Libraries programmiersprachliche Bibliotheken und die technische Ausprägung eines grobgranularen Ansatzes. Im Gegensatz zu Toolkits und Libraries legen sie die Architektur einer Anwendung weitgehend fest. Funktionen eines Toolkits oder einer Library werden nur von anderen Teilen einer Applikation benutzt, verwenden umgekehrt aber keine Funktionen der Applikation. Ein Framework dagegen erwartet auch die Bereitstellung von Funktionen durch die Applikation, die es dann aufruft. Kontrollfluß und Abhängigkeiten sind bei der Verwendung von Toolkits dadurch weitgehend vorgegeben. Beispiele für Frameworks sind graphische Editoren, die auf ein Anwendungsgebiet spezialisiert werden oder ein Compiler, dessen Codegenerierung auf verschiedenen Zielarchitekturen angepaßt werden kann [51, 25]. Frameworks bilden also den unspezialisierten Kern einer Klasse von Anwendungen.

4.1.4 Design-Patterns

Design-Patterns sind Muster für das Zusammenwirken von Klassen in objektorientierten Programmiersprachen [28]. Im Gegensatz zu den zuvor beschriebenen Methoden der Wiederverwendung werden Design-Patterns nicht durch programmiersprachliche Quelltexte oder formale Spezifikationen beschrieben, sondern durch Beispiele, Diagramme und Erläuterungen. Erst konkrete Anwendungen von ihnen lassen sich als Programm ausdrücken. Design-Patterns sind kleine architektonische Bausteine, die nur für einen einzelnen Aspekt Abhängigkeiten und Verantwortlichkeiten in einer Applikation festlegen. Die meisten Design-Patterns beschreiben das Zusammenspiel von weniger als drei Klassen in einem objektorientierten System. Sie sind so allgemein wie Funktionen in einem Toolkit oder einer Library, aber mit einem geringeren algorithmischen Charakter.

Ein Beispiel für ein Design-Pattern ist ein *Proxy*: ein Proxy ist ein Platzhalter für ein anderes Objekt. Eine Anwendung kommuniziert statt mit einem Objekt mit dessen Proxy, der das Objekt von der Anwendung abschirmt. Die Aufgabe des Proxy ist es, die gewünschte Funktionalität mit Hilfe des von ihm verdeckten Objektes herzustellen. Eine spezielle Instanz des Proxy ist ein *Cache-Proxy*, der die von seinem Objekt zurückgegebene Ergebnisse zwischenspeichert, und so die Anzahl der Anfragen an dieses Objekt verringern kann, ohne daß die Applikation davon betroffen wird. Design-Patterns abstrahieren und dokumentieren also das Wissen, wie einzelne Aspekte in objektorientierten Systemen zusammenwirken. Sie setzen explizit die Techniken von Klassen, Vererbung und dynamische

Bindung objektorientierter Programmiersprachen voraus und lassen sich deshalb nicht auf andere Paradigmen übertragen. Frameworks können durch Kombinationen von Design-Pattern beschrieben werden [51]; ein Vergleich von Frameworks und Design-Patterns ist in [28] enthalten.

4.1.5 Wiederverwendung ist häufig quelltextorientiert

Die meisten Ansätze zur Softwarewiederverwendung, unabhängig ob komponenten- oder prozeßbasiert, konzentrieren sich auf ausführbaren Programmcode. Der wahrscheinlichste Grund ist, daß Programmcode der zentrale Bestandteil jeder Software ist, deshalb am besten verstanden ist und seine Wiederverwendung einen unmittelbaren Vorteil verspricht. Für Programmcode existiert eine reichhalte Kette von Werkzeugen um ihn zu be- und verarbeiten (Editoren, Quelltext-Browser, Compiler, Debugger, Profiler) und so ist es naheliegend, den Anfang dieser Kette um Werkzeuge zu verlängern, die diesen Programmcode bereitstellen. Der in der Praxis geringe Erfolg von Softwarewiederverwendung könnte aber auch gerade mit dieser Konzentration auf ausführbaren Programmcode zu tun haben [72, 90]. Er entsteht am Schluß im Prozeß der Softwareerstellung und ist zwangsläufig spezialisiert und in ihm ist das zugrundeliegende Wissen in Programmanweisungen atomisiert. Eine unmittelbare (Wieder-) Verwendung ist oft nicht möglich, sondern erfordert eine Adaption des Codes.

Generische, also leicht wiederverwendbare, Komponenten zu entwickeln erfordert neben der nötigen Planung möglichst eine Unterstützung durch die verwendete Programmiersprache. Module, Polymorphismus und möglichst allgemeine Parametrisierung, wie sie in den neueren Programmiersprachen C++, Java oder Standard ML zu finden sind, begünstigen die Wiederverwendung von Programmcode. Design-Patterns und Generatoren sind Beispiele für Wiederverwendungsmethoden, die im Softwarentwicklungsprozeß früher als ausführbarer Code stehen, da sie selber welchen erzeugen. Die selbstverständliche Verwendung von Generatoren im Bereich von Compilern, graphische Benutzungsoberflächen und Designwerkzeugen könnten ein Hinweis sein, daß Softwarewiederverwendung in frühen Phasen der Softwareerstellung vielversprechend ist. Dafür würde auch das große Interesse an Design-Patterns [28, 48] sprechen, deren konkrete Instanzen Quellcode sind, die selbst aber auf einer Meta-Ebene beschrieben werden.

4.1.6 Der Einfluß von Programmiersprachen

Prinzipiell sind die angesprochenen Techniken der Wiederverwendung mit Ausnahme der Design-Patterns von der verwendeten Programmiersprache unabhängig. Bei an programmiersprachlichem Programmcode orientierten Methoden der Wiederverwendung ist ein Einfluß der gewählten Programmiersprachen natürlich trotzdem vorhanden.

Bestimmte Eigenschaften von Programmiersprachen erleichtern die Wiederverwendung ihres Codes: Klassenbasierte objektorientierte Sprachen wie C++, Java, Eifel und Smalltalk [106, 38, 70, 37] können durch *Vererbung* bestehende Klassen wiederverwenden und verfeinern [20]. Die Wiederverwendung der ererbten Klassen muß in diesen nicht explizit gekennzeichnet sein, kann also nachträglich erfolgen, sogar alleine an Hand der Kompilate und ohne Zugriff auf den vollständigen Quelltext der ererbten Klassen.

Die *Parametrisierung von Modulen* in Ada und Standard ML [75] sowie Klassen in C++ erlaubt es, generische Komponenten geplant anzulegen und somit Wiederverwendung zu antizipieren. In imperativ/funktionale Sprachen wie Standard ML und rein funktionale Sprachen wie Haskell [47] enthält Programmcode nur wenige oder *keine imperativen Seiteneffekte*. Die Semantik des Codes ist dadurch leichter vollständig zu erfassen und ist damit besser wiederverwendbar. Ein *Typsystem* vergrößert die Sicherheit bei der Integration von Programmen in Java, C++, Eifel, C [53], Standard ML oder Modula-2 [115] und ein *Modulsystem* erlaubt Komponenten in ihnen separiert zu entwickeln und zu testen.

Programmiersprachen bringen also mehr oder weniger Eigenschaften von Hause aus mit, die Wiederverwendung fördern. Dabei ist es kein Zufall, daß die jüngeren Sprachen mehr dieser wünschenswerten Eigenschaften vereinen, da ihre Bedeutung auch von den Sprachdesignern erkannt wurde. Daraus zu schließen, daß für diese Sprachen keine weitere Unterstützung für Wiederverwendung nötig ist, wäre voreilig: auch für sie muß wiederverwendbarer Code abgelegt und wiedergefunden, beziehungsweise generiert werden. Die Organisation dieser Aspekte liegt außerhalb der Sprachen und muß von externen Methoden und Werkzeugen übernommen werden, wie sie zuvor vorgestellt worden sind.

4.2 Komponentenbasierte Wiederverwendung

Libraries und Toolkits als Sammlungen programmiersprachlicher Komponenten für eine spätere Wiederverwendung sind ausführlich untersucht worden [26, 69, 80, 83, 92, 93, 116, 77, 73]. Neben prozeßorientierten Ansätzen mit Hilfe von Generatoren sind Komponentensammlungen die häufigste Form der Wiederverwendung. Eine ausführliche Übersicht über diese Ansätze bieten Mili et al. in [71]. Komponentenbibliotheken lassen sich nach Mili et al. in sechs Kategorien unterscheiden [71], die im Folgenden vorgestellt werden. Andere Autoren ([9, 45]) nehmen ähnlich Kategorisierungen vor, die durch die von Mili et al. subsumiert werden.

4.2.1 Information-Retrieval

Die Organisation von Bibliotheken aus unregelmäßig[1] formatierten Texten und anderen Informationen wird durch das Teilgebiet *Information-Retrieval* untersucht [95]. Die dort verwendeten Techniken wurden für eine Reihe von Systemen zur Komponentenwiederverwendung adaptiert. Charakteristisch für diesen Ansatz ist, daß Softwaremodule an Hand natürlichsprachlicher Texte aus ihrer Dokumentation indexiert werden. Aus der Dokumentation wird ein sogenanntes *Profil* erstellt. Damit wird eigentlich nicht eine Komponente selbst, sondern stellvertretend ihre Dokumentation in einer Sammlung aufgenommen. Anfragen bestehen ebenfalls aus freien Texten und selektieren die Komponenten der Sammlung, die die größte syntaktische Ähnlichkeit zwischen ihrem Profil und dem Profil der Anfrage aufweisen. Durch Häufigkeitsanalysen von Wörtern in der Dokumentation von Komponenten kann die Relevanz einzelner Wörter für das Profil einer Komponente oder Anfrage gewonnen werden.

Die Vorteile von Information-Retrieval-Systemen sind eine weitestgehend automatische Indexierung von Komponenten, der Rückgriff auf bewährte Implementierungen und ihre Bereichsunabhängigkeit. Die freie Eingabe von Anfragen schränkt den Anwender nicht ein, läßt ihn aber möglicherweise auch im Unklaren, wie die Anfrage für eine gesuchte Komponente formuliert wird. Dieses Problem hängt damit zusammen, daß Komponentenbeschreibung und Anfragesprache informal sind und Selektion alleine auf syntaktischen Ähnlichkeiten beruht – eine Anfrage oder Komponente wird in keiner Weise semantisch erfaßt (*anders als Quicksort, ...*). Deshalb sind diese Techniken besser für ein lokales Umfeld geeignet, weil sich dort am ehesten eine informale Nomenklatur etabliert, die Mehrdeutigkeiten verhindert. Die freie Verwendung von Text macht die Methoden für jede Art von Komponenten geeignet, sie ist nicht etwa an Komponenten einer bestimmten Programmiersprache gebunden.

Beispiele für Komponentensammlungen, die einen Information-Retrieval-Ansatz verwenden, sind Frakes und Nehmeh [23] und Maarek et al. [69]. In dem zweiten der beiden Ansätze werden alle Komponenten zusätzlich durch eine Cluster-Analyse in hierarchischen Clustern zusammengefaßt. Ausgehend von einem Suchergebnis kann ein Benutzer durch die zugehörigen Cluster-Strukturen navigieren. Dieser Ansatz kombiniert also die freie Eingabe von Anfragen mit Browsing von Ergebnissen. In [43] wird ebenfalls Information-Retrieval mit Browsing verbunden, dort allerdings entlang von Klassenhierarchien objektorientierter Komponenten. Browsing schränkt die Form einer Eingabe ein, in dem sie Inhalte zur Auswahl vorgibt. Das Ziel ist, den Benutzer von Schwierigkeiten bei der syntaktischen Gestaltung seiner Eingabe zu befreien und inhaltlichen Aspekte in den Vordergrund zu stellen.

[1] streng formatierte Daten werden in Standard-Datenbanken organisiert, siehe [15]

4.2.2 Deskriptive Methoden

Wie Information-Retrieval-Methoden verwenden deskriptive Methoden die Dokumentation von Komponenten und besitzen damit die gleiche lose semantische Kopplung zwischen Beschreibung und Komponente. Zur Charakterisierung von Komponenten wird aber nicht der gesamte vorhandene Text verwendet, sondern nur ausgewählte Schlüsselwörter[2]. Die Anfrage besteht ebenfalls aus Schlüsselwörtern und selektiert genau die Komponente, mit der exakten syntaktischen Übereinstimmung zwischen beiden Schlüsselwortmengen. Die Indexierung von Komponenten durch Schlüsselwörter erfolgt manuell und folgt einem zuvor vereinbartes Schema. Dies erhöht den Aufwand bei der Erstellung und Pflege von Komponentensammlungen, verringert aber Mehrdeutigkeiten und steigert die semantische Kopplung zwischen Komponenten und ihrer Beschreibung. Eine Anfrage kann nur aus dem kontrollierten Vokabular der Sammlung formuliert werden; diese Einschränkung kann unterstützend für einen Benutzer sein, wenn er mit dem Vokabular vertraut ist und hinderlich andernfalls [19]. Wenn verschiedene Aspekte einer Komponente unabhängig von einander durch ein oder mehrere Schlüsselwörter klassifiziert werden, spricht man von *facettierter Klassifikation* [89]. Sie ist eine Weiterentwicklung der numerischen Klassifikation, die ansonsten in Bibliotheken verwendet wird. Die einzelnen Aspekte oder Facetten spannen jeweils eine Dimension in dem Klassifikationsraum auf. Die Facetten werden bei der Formulierung der Anfrage mit Wildcards oder Schlüsselwörtern belegt, wobei das Vokabular jeder einzelnen Facette kontrolliert ist.

Die bekanntesten Beispiele für deskriptive Methoden zur Organisationen einer Komponentenbibliothek sind die Arbeiten von Prieto-Diáz [89, 88]. Sie verwenden eine Kombination aus einer facettierten Klassifikation und einem semantisches Netzwerk: die gültigen Schlüsselwörter einer Facette sind durch ein semantisches Netzwerk verknüpft. Dieses Netzwerk beschreibt über ein Abstandsmaß die Ähnlichkeit eines Schlüsselwortes mit anderen Schlüsselwörtern innerhalb einer Facette (ein *backspace* ähnelt einem *blank*, *digit* und *character*). Dieses Netzwerk wird verwendet um verwandte Komponenten vorzuschlagen, wenn eine gesuchte Komponente nicht in der Sammlung vorhanden ist. Einzelne Schlüsselwörter werden dann für eine Anfrage durch ähnliche Schlüsselwörter ersetzt. Ebenso wie die Indexierung erfolgt der Aufbau des semantischen Netzwerkes manuell. Der erforderliche Aufwand ist beträchtlich und die Aufgabe, sinnvolle Distanzmaße anzugeben, nicht trivial [80]. Dafür selektiert eine Anfrage statt einer einzelnen Komponente eine Menge von ähnlichen Komponenten und bietet dadurch eine stärkere Rückkopplung für den Anwender.

Liao et al. [61] verwenden facettierte Klassifikation zur Beschreibung von Softwarekomponenten zusammen mit einem hierarchischen Thesaurus. Der hierarchische Thesaurus unterscheidet vier Stufen mit deren Hilfe bei der Suche nach Syn-

[2]Der Begriff Schlüsselwörter wird hier nicht in dem exakten Sinne der Datenbanktheorie [15] verwendet

onymen die Breite der Suche gesteuert werden kann. Bei einer Anfrage werden alle Komponenten ermittelt, die der Anfrage und aus ihr abgeleiteten Synonymen gleichen. Je mehr Synonyme für die Anfrage bestimmt werden, desto größer (und ungenauer) ist die Ergebnismenge. Die Intention der Autoren ist eine Adaption des Anfrageverhaltens an Anfänger und Experten. Ebenso wie Prieto-Diáz verwenden Liao et al. Abstandsmaße zwischen Wörtern des Thesaurus und erhalten damit auch zwischen der Anfrage und den Elementen der Ergebnismenge ein Ähnlichkeitsmaß, das bei der Präsentation eines Ergebnisses zu seiner Ordnung verwendet wird.

4.2.3 Operationale Methoden

Operationale Methoden beschreiben das Verhalten einer Komponente nicht durch eine Abstraktion, sondern verwenden es selbst als Unterscheidungsmerkmal. Dazu müssen Komponenten ausführbar sein: eine Anfrage besteht aus der Signatur[3] der gesuchten Komponente, einer frei gewählten Eingabe und einer erwarteten Ausgabe. Alle Komponenten der Bibliothek mit passender Signatur werden mit der Eingabe ausgeführt und ihre Ausgabe mit der erwarteten Ausgabe verglichen. Komponenten, deren Ausgabe die Erwartung erfüllen, werden in das Ergebnis der Anfrage aufgenommen.

Die Eleganz des Ansatzes liegt darin, daß keine Abstraktion zur Beschreibung von Komponenten benutzt wird, sondern die Komponenten sich durch ihr Verhalten selbst beschreiben. Dennoch stößt dieser Ansatz in der Praxis auf zahlreiche Probleme: schon eine flexible Auswahl von Komponenten aus einer Sammlung an Hand ihrer Signatur ist nicht trivial [92, 120], dies gilt erst Recht, wenn statt Basisdatentypen benutzerdefinierte oder abstrakte Datentypen verwendet werden. Die Eingabe von Daten für Komponentenparameter und das erwartete Ergebnis erfordern eine aufwendige Infrastruktur. Komponenten in imperativen Programmiersprachen liefern nicht immer ein Ergebnis zurück, sondern können einen Seiteneffekt erzeugen (*(malloc(), write()*), der durch operationale Methoden nicht direkt erfaßt werden kann. Das Verfahren ist deshalb besser für funktionale Sprachen geeignet, für die auch Signature-Matching untersucht ist. Die algorithmische Komplexität (Laufzeit, Speicherbedarf) jeder Komponente geht unmittelbar in die Suche ein; Nicht-Terminierung oder Fehler bei der Abarbeitung sind weitere theoretische und praktische Probleme dieses Ansatzes.

Der erste Vorschlag, das operationale Verhalten von Komponenten zur Organisation einer Komponentensammlung zu verwenden, stammt von Podgurski und Pierce [85]. Ihre Komponenten sind C-Funktionen mit einem funktionalen[4] Verhalten und Basisdatentypen als Ein- und Ausgabeparameter. Sie untersuchen, ob schon wenige Ein- und Ausgaben das Verhalten von Funktionen ausreichend

[3] Anzahl und Datentypen der formalen Parameter sowie der Typ ihres Ergebnisses

[4] frei von Seiteneffekten

charakterisieren. Für das Problem komplexer und abstrakter Datentypen schlagen sie vor, das Verhalten eines abstrakten Datentyps durch die in seiner Signatur verwendeten Datentypen zu beschreiben. Dies läuft auf eine Beschreibung von abstrakten Datentypen durch algebraische Spezifikationen hinaus [114], die das funktionale Verhalten komplexer Typen durch ihre Interaktion mit anderen Typen über Gleichungen definieren. Während algebraische Spezifikationen eine (vollständige) Menge von Gleichungen sind, müßte ein Benutzer operationaler Methoden nur einen Spezialfall in Form einer Gleichung angeben.

Chou et al. spezifizieren das Verhalten objektorientierter Klassen durch endliche Automaten [9]. Im Gegensatz zu Podgurski und Pierce kann das Verhalten nicht durch Ausführung der Software ermittelt werden, sondern muß manuell spezifiziert werden. Chou et al. verwenden bei der Spezifikation auch Instanzvariablen einer Klasse und kodieren so Wissen in einer Spezifikation, das nicht für die Verwendung einer Klasse von Bedeutung sein sollte. Die Suche von Komponenten verwendet zur Selektion die syntaktische Ähnlichkeit zwischen einer Anfrage und der Spezifikation einer Komponente. Da Spezifikationen Interna von Klassen enthalten und gleichzeitig eine syntaktische Übereinstimmung zur Selektion von Komponenten gefordert wird erscheint fraglich, ob Benutzer mit dieser Methode Komponenten allein an Hand ihrer Verhaltens selektieren können.

In *Generalized Behavior-based Retrieval* ermittelt Robert J. Hall nicht nur die Komponenten einer Sammlung, die ein vorgegebenes Verhalten zeigen, sondern setzt auch primitive Komponenten zu neuen zusammen, um die gewünschte Funktionalität zu erhalten [41]. Der Basismechanismus entspricht dem von Podgurski und Pierce: Durch Eingabe einer Signatur und Beispieldaten für Ein- und Ausgabe werden funktionale Common-Lisp Komponenten selektiert, wenn sie das gewünschte Verhalten zeigen. Hall adressiert eine Reihe der oben angesprochenen Probleme in seinem Ansatz: komplexen Datentypen werden durch Konstruktoren erzeugt, die dann als Argumente für Anfragen nach Komponenten mit diesen Argumenten eingesetzt werden. Seiteneffekte von Komponenten werden modelliert, indem auch die Umgebung, auf die dieser Seiteneffekt wirkt, explizit gemacht wird – also ein Modell dieser Umgebung verwendet wird. Das verwendete Typsystem für Signaturen ähnelt dem einfachen Ansatz von Podgurski und Pierce und berücksichtigt weder Polymorphismus oder Isomorphie von Signaturen [92, 76]. Die Signaturen von Komponenten leiten das Zusammensetzen von primitiven Komponenten bei der Suche nach komplexen Komponenten, die das gesuchte Verhalten zeigen.

4.2.4 Formal-Logische Methoden

Alle bislang vorgestellten Methoden (Abschnitte 4.2.1, 4.2.2, 4.2.3) etablieren eine lose Kopplung zwischen den Eigenschaften einer Komponente und ihrer Beschreibung. Dies reicht von Information-Retrieval-Methoden, in denen Eigenschaften durch freien Text beschrieben werden, bis zur facettierten Klassifikation,

die durch ein kontrolliertes Vokabular Komponenten beschreibt. Die Bedeutung der Begriffe (*Sortierfunktion*, *Ordnung*, *fügt ein*, ...) ist nur durch informale Konventionen definiert. Als Folge davon muß eine Anfrage syntaktisch weitgehend mit einer Komponentenbeschreibung übereinstimmen, um sie zu selektieren. Anfragen können praktisch nicht semantisch äquivalent umgeformt werden, um so auch eine Komponente zu selektieren, deren Beschreibung syntaktisch nicht mit der Anfrage übereinstimmt. Theasauri und semantische Netze adressieren diesen Problem, lösen es aber zumindest unter formalen Gesichtspunkten nur unbefriedigend. Formal-logische Methoden[5] definieren Eigenschaften von Komponenten mit Hilfe formaler Spezifikationen, die die semantischen Eigenschaften der Komponenten genauer erfassen als die vorangegangenen Methoden. Eine Anfrage selektiert darin alle Komponenten, die die Anfrage im Sinne des Formalismus semantisch erfüllen, und geht so über die syntaktische Gleichheit von Anfrage und Beschreibung hinaus. Zur semantischen Spezifikation von Software existieren eine Vielzahl von Methoden, die allerdings in erster Linie für den formalen Entwurf von Software entwickelt wurden und nicht zur Spezifikation wiederverwendbarer Komponenten. Spezifikationssprachen können modell-orientiert wie VDM [16] oder Z [103], algebraisch wie OBJ [36], prozeßorientiert wie CCS [74], logisch oder konstruktiv sein.

Signature-Matching

Rittri schlägt vor, Signaturen von Funktionen als Schlüssel zur Ablage und Suche zu verwenden. Er demonstriert *signature matching* an Hand von Komponenten der funktionalen Programmiersprache *Lazy ML*, einem Dialekt von *Standard ML* [75, 92]. Eine Funktion mit der gesuchten Signatur ($real \rightarrow real$) garantiert zwar nicht, daß diese Funktion insgesamt die gewünschte Funktionalität (zum Beispiel $\sin(x)$) besitzt, erfüllt aber das notwendige Kriterium der Typkorrektheit. Bei Programmierern typisierter Programmiersprachen existiert ein starkes Bewußtsein für Typen, so daß ihre Verwendung als Suchschlüssel natürlich ist.

Um von den relativ beliebigen Aspekten einer Signatur wie Parameterreihenfolge und Currying zu abstrahieren, definiert Rittri einen geeigneten Isomorphismus für Signaturen. Zusätzlich erlaubt er polymorphe Signaturen [14], so daß auch allgemeinere als durch die Signatur vorgegeben Lösungen gefunden werden. Er entwickelt eine Theorie, die zu einer Signatur alle Komponenten mit gleicher oder allgemeinerer Signatur ermittelt, unter Berücksichtigung des zuvor erwähnten Isomorphismus. Die Theorie zu diesem Problem wurde noch genauer von Di Cosmo untersucht [12]. Sie ist bislang nur für Basistypen (wie *bool* und *int*) sowie Typkonstruktoren für Funktionen und Tupel beschrieben; die wichtige Behandlung benutzerdefinierter Typkonstruktoren steht noch aus. *Signature matching* wird auch in operationalen Methoden verwendet (siehe 4.2.3), allerdings weitge-

[5] diese Methoden werden von Mili et al. als *Denotational Semantics Methods* bezeichnet [71].

hend auf syntaktische Gleichheit beschränkt.

Zaremski und Wing haben *signature matching* am Beispiel von Standard ML [75] für die Suche von Funktionen und Modulen untersucht. Wie Rittri betrachten sie ein polymorphes Typsystem und berücksichtigen Isomorphie, Generalisierung und Spezialisierung von Typausdrücken bei der Suche. Ihre Untersuchung ist im Vergleich stärker von Software-Engineering-Aspekten geprägt und geht nicht auf die theoretischen Aspekte wie Entscheidbarkeit, Eindeutigkeit und Terminierung der zu Grunde liegenden Termersetzungssysteme ein.

Specification-Matching

Da Signaturen nur einen Teil der Semantik einer Komponente beschreiben, muß man noch einen Schritt weitergehen, wenn eine möglichst vollständige Beschreibung der Komponenten in einer Sammlung das Ziel ist. Moormann-Zaremski und Wing benutzen sortierte Prädikatenlogik ersten Stufe als formalen Spezifikationen (Larch [40]) zur Beschreibung und Suche von Standard ML-Komponenten [119]. Sorten werden mit Typen in (Standard ML-) Komponenten assoziiert und Vor- und Nachbedingungen spezifizieren das Verhalten der Komponenten.

Eine Anfrage (P_1, Q_1) besteht aus einer Vorbedingung P_1 und einer Nachbedingung Q_1, die in einer logischen Relation zu der Vor-/Nachbedingung (P_2, Q_2) ausgewählter Komponenten steht. Gesuchte und gefundene Komponenten können funktional äquivalent, spezieller oder allgemeiner sein. Moormann-Zaremski und Wing untersuchen 8 verschiedene Relationen, die im Rahmen der Wiederverwendung sinnvoll sind. Ob eine bestimmte Relation zwischen einer Anfrage und einer Komponente besteht, ist im allgemeinen unentscheidbar. Implementierungen verwenden Theorem-Beweiser [6], um die Relation zwischen Anfrage und Komponentenbeschreibung zu überprüfen. Die Implementierung einer Suche mit Spezifikationen muß in einem ersten Schritt *signature matching* verwenden, um die Variablen einer Anfrage auf Variablen in einer Spezifikation abzubilden und überhaupt nur von ihrer Signatur her geeignete Spezifikationen zu überprüfen. Weil die Behandlung benutzerdefinierter Datentypen im Signature-Matching weitgehend ungelöst ist, gilt dieses auch für die spezifikationsbasierte Suche von Komponenten.

Der logische Vergleich einer Anfrage-Spezifikationen mit jeder Komponentenspezifikation durch automatische Theorem-Beweiser ist extrem rechenzeitaufwendig. Um spezifikationsbasierte Suche interaktiv verwenden zu können, versuchen Fischer et al. den Einsatz von Beweisern zu minimieren [18]. Dazu durchlaufen Komponenten eine graphisch konfigurierbare Kette von Filtern, deren letztes Glied erst der Theorem-Beweiser ist. Die Glieder der Kette bestehen aus Modulen wie *signature matching*, Widerlegungsfilter, Model-Checker und Simplifizierer basierend auf Termersetzung. Da alle Zwischenergebnisse sichtbar gemacht werden, kann ein Anwender auch schon vor der endgültigen Bestätigung durch den Theorembeweiser Komponenten untersuchen. Fischer et al. haben eine mittel-

große Sammlung formaler Komponenten-Spezifikationen in VDM [16], verschiedene Theorem-Beweiser, Filter und Relationen für Spezifikationen in einer großen Testreihe untersucht. Durch Parallelisierung der Filter und Vereinfachung der Beweisaufgaben erreichen sie für interaktive Anwendungen geeignete Antwortzeiten.

Die Komplexität des Vergleichs formaler Spezifikationen zwischen eine Anfrage und jeder Komponente ist eine technische Schwierigkeit. Ein weitere ist, überhaupt eine korrekte Anfrage für eine gesuchte Komponente zu formulieren. Spezifikationssprachen besitzen wie Programmiersprachen eine reichhaltige Syntax und Semantik, die ein Benutzer erst erlernen muß. Im Vergleich zu den oben vorgestellten Methoden besitzen formal-logische Methoden die komplexeste Syntax und Semantik. Obwohl formale Spezifikationen von Implementierungsdetails einer Komponente abstrahieren, wird die Formulierung einer Spezifikation von vielen Programmierern als fast so kompliziert empfunden, wie das Kodieren einer entsprechenden Komponente. Dies gilt besonders in den überwiegenden Entwicklungsprozessen, die formale Spezifikationen nicht natürlich integrieren.

Um sowohl die Komplexität der Benutzungsschnittstelle als auch die algorithmische Komplexität zu reduzieren, entkoppeln neuere Ansätze den Suchvorgang von dem Vergleich der Spezifikationen: mit Hilfe automatischer Theorembeweiser werden abstrakte Eigenschaften von Komponenten in einem einmalig ablaufenden Indexierungsprozeß extrahiert und in eine neue, semantisch fundierte, Navigationsstruktur übersetzt. Bei der Suche nach einer Komponente wird nur noch die zuvor gewonnene Navigationsstruktur verwendet, die eine einfachere Benutzungsschnittstelle zuläßt. Beispiele für diese hybride formal-logische Methode sind die Arbeiten von Penix und Alexander [84], Jeng und Cheng [50] und Fischer [17]. Die Methode von Fischer verwendet formale Begriffsanalyse zur Berechnung der Navigationsstruktur und wird in Abschnitt 7 noch einmal aufgegriffen.

4.2.5 Wissensbasierte Methoden

Formal-logische Methoden zielen darauf ab, die programmiersprachliche Semantik von Komponenten genau zu erfassen. Zwischen diesem Extrem auf der einen Seite, und den zuvor vorgestellten eher syntaktischen Methoden auf der anderen, liegen die wissensbasierten Methoden. Sie verwenden Verfahren aus der Wissensrepräsentation und künstlichen Intelligenz; Mili et al. bezeichnen sie als *topologische Methoden* [71], da sie ein Abstandsmaß zwischen gesuchten und präsentierten Komponenten minimieren.

Wissensbasierte Verfahren gehen von einer inhärenten Ungenauigkeit bei der Beschreibung von Komponenten und der Anfrage aus. Ein Benutzer wisse selten genau, welche Funktionalität er suche und könne entsprechend keine detailierte Anfrage stellen, so Henninger [45]. Wissensbasierte Methoden präsentieren deshalb nicht nur die Komponenten, die eine Anfrage genau erfüllen, sondern auch ähnliche. Durch Inspektion der präsentierten und tatsächlich in einer Sammlung vorhandenen Komponenten präzisiert sich die Vorstellung des Benutzers

von der gesuchten Komponente. Implementierungen wissensbasierter Methoden unterstützen in der Regel eine schrittweise Verfeinerung einer Anfrage [19]. Weil auch andere Methoden näherungsweise Suchen erlauben, oder durch die Organisation ihrer Sammlung einen Verallgemeinerungs- und Spezialisierungsbegriff kennen (zum Beispiel [69, 61, 17]), ist die Abgrenzung zu wissensbasierten Methoden nicht immer scharf.

Die klassische Veröffentlichung über wissensbasierte Methoden bei der Organisation von Komponentenbibliotheken stammt von Ostertag et al. [80]. Komponenten und Teilsammlungen (*packages*) werden im AIRS-System durch *frames* beschrieben. Ein Frame ist eine Menge von Name-Wert-Paaren (*features*) und zur Beschreibung von Komponenten und *packages* existieren verschiedene Typen von Frames. Frames werden im Bereich der Wisssensrepräsentation zur Modellierung von Wissen eingesetzt; die bekannteste Frame-Logik ist KL-One [117]. Die Autoren fassen Frames als eine Verallgemeinerung von Facetten der facettierten Klassifikation (siehe 4.2.2) auf. Komponenten sind durch einen manuell spezifizierten Subsumptionsgraphen untereinander verbunden. Er beschreibt quantitativ, inwieweit eine Komponente zur Realisierung einer anderen herangezogen werden kann. Die möglichen Werte eines *features* sind durch ein weiteres semantisches Netzwerk für Ähnlichkeit (*closeness*) verbunden, das ebenfalls manuell spezifiziert werden muß. Der Aufwand, um eine Komponenten mit dem Feature a in eine Komponente mit dem Feature a' umzuschreiben, wird durch den Abstand von a und a' im semantischen Netzwerk beschrieben.

Anfragen werden ebenfalls als Frames formuliert, zu denen das AIRS-System mit Hilfe der semantischen Netzwerke exakt passende Komponenten, semantisch ähnliche (subsumierende und subsumierte) oder syntaktisch ähnliche ermittelt. Durch die Übernahme von Features aus gefundenen Komponenten kann ein Benutzer seine Anfrage iterativ verfeinern und so durch die Komponentensammlung navigieren.

Hennigner beschreibt eine zweiteilige Implementierung bestehend aus PEEL und CodeFinder, die AIRS ähnlich ist [45]: sie verwendet ebenfalls Frames und semantische Netze und erlaubt eine iterative Verfeinerung von Anfragen. Im Gegensatz zu Ostertag et al. betont Henninger die Bedeutung eines möglichst kleinen Aufwandes zum Aufbau einer initialen Komponentensammlung. Die manuelle Spezifikation der semantischen Netze bei AIRS ist in der Praxis ein gewichtiger Nachteil. PEEL bereitet existierenden ELisp-Quelltexte für den GNU Emacs-Editor [104] zur Ablage in einer Komponentensammlung auf. Komponenten werden durch Frames repräsentiert. Bei ihrer Erfassung versucht die Textanalyse von PEEL möglichst viele Name-Wert-Paare automatisch vorzubesetzen, um so die Eingabe zu beschleunigen.

Die Frames bilden eine Hierarchie, in die sich neu erfaßte Komponenten automatisch einfügen. CodeFinder ist zuständig für die Suche von Komponenten in ihrer Frame-Repräsentation, wobei eine Anfrage ebenfalls als Frame vorliegt. Zum Zeitpunkt der Anfrage wird eine Art neuronales Netzwerk zwischen den Werten

von Features aus der Anfrage und Komponenten berechnet und mit Hilfe der berechneten Gewichte auf den Verbindungen eine Menge von Komponenten als Ergebnis ausgewählt. Mit Hilfe der ausgewählten Komponenten kann ein Benutzer seine aktuelle Anfrage interaktiv verfeinern und so sich iterativ der gesuchten Komponente nähern. Zusätzlich werden die Aktionen der Benutzer ausgewertet, um die Indexierung der Komponenten automatisch zu verbessern. Das Ziel ist, die Indexierung der Komponenten zu adaptieren, so daß sie das gemeinsame Wissen der Benutzer über die Komponentensammlung widerspiegelt.

Wissensbasierte Methoden erfassen die Funktionalität von Komponenten prinzipiell nicht genauer als deskriptive Methoden, da sie wie diese ein kontrolliertes Vokabular verwenden, das (intendiert) ebenfalls nicht formal definiert ist. Ihr frame-basiertes Klassifikationschema ist eine Verallgemeinerung der starren Indexierung der deskriptiven Methoden. Im Vergleich zu allen anderen Methoden legen die Beispiele für wissensbasierte Methoden besonderen Wert auf eine leichte Formulierbarkeit von Anfragen und unterstützen das Verfeinern und Inspizieren von Anfragen. Der Benutzer wird bei der Formulierung der *Form* seiner Anfrage stark unterstützt und kann sich auf die inhaltlichen Aspekte seiner Suche konzentrieren.

4.2.6 Strukturelle Methoden

Mit Ausnahme der operationalen Methoden verwenden alle Methoden nicht Komponenten selbst, sondern Abstraktionen oder Beschreibungen von ihnen. Operationale Methoden suchen Komponenten an Hand ihres charakteristischen Laufzeitverhaltens, also einer inhärenten Eigenschaft. Nach Mili et al. [71] verwendet die sechste und letzte Methode zur Organisation von Komponenten direkt den Quelltext von Komponenten. Die implizite Annahme dieser strukturellen Methode ist, daß bis zu einem gewissen Grad syntaktisch ähnliche Komponenten auch ähnliche Funktionalität besitzen.

Die Unterscheidbarkeit von Funktionen und Modulen der meisten universellen Programmiersprachen an Hand ihrer syntaktischen Struktur dürfte (zu) gering sein. Diese Methode ist nur in speziellen Bereichen sinnvoll einsetzbar, deren Komponenten sehr charakteristische Strukturen aufweisen; eine Bibliothek von Design-Patterns in Form von Struktur-Diagrammen [28] wäre ein denkbares Beispiel.

Das von Mili et al. [71] zitierte Beispiel von Paul und Prakash [82] zur Suche von Mustern in C-Quelltexten dient weniger der Suche nach Komponenten, als nach Anomalien und nicht portablen Stellen im Quelltext. Das System erlaubt Muster einzugeben, zu denen dann passende Stellen des Quelltextes bestimmt werden. Diese Muster können zwar auch ganze Funktionen charakterisieren, in der typische Anwendung beschreiben die Muster aber kleinere Einheiten und dienen mehr der Inspektion der gefundenen Stellen als ihrer Wiederverwendung.

Für Anwendungen dieser Art sind eine Reihe von Werkzeugen entstanden,

weil Standardwerkzeuge, die häufig alleine auf regulären Ausdrücken basieren, zur Untersuchung von stark strukturierten Daten nicht geeignet sind [49, 10]. Das Werkzeug von Paul und Prakash arbeitet auf dem Quelltext der Komponenten, also ihrer *konkreten* Syntax. ASTLOG von Roger Crew [13] erlaubt Suchen in der *abstrakten* Syntax von C-Programmen; da Anfragen dann von semantisch unbedeutenden syntaktischen Feinheiten abstrahieren können, sind sie im Vergleich einfacher, das gesamte Werkzeug zwangsläufig aber auch wesentlich spezialisierter. Es wird typischerweise ebenfalls zur Analyse von C-Quelltexten eingesetzt.

Zusammenfassend gesagt sind strukturelle Merkmale zur Selektion von Software geeignet, allerdings nicht im Bereich der universellen Programmiersprachen, um Softwarekomponenten für ihre Wiederverwendung zu finden.

4.3 Kriterien zur Auswahl von Methoden

Die Vielfalt der vorgestellten Methoden wirft die Frage auf, wie diese sich vergleichen und welche dem Ziel der Softwarewiederverwendung am besten dienen. Eine erfolgreiche Softwarewiederverwendung ist das Produkt vieler Faktoren und beschränkt sich nicht auf die Auswahl einer Lösung zur Ablage von Komponenten. Erst ein Zusammenspiel aus Werkzeugen und Entwicklern in einem Prozeß, der Komponenten bereitstellt und verwendet, verwirklicht Softwarewiederverwendung. Somit ist das richtige Management mindestens so wichtig wie die richtige Methode. Vor diesem Hintergrund sollten die Unterschiede zwischen den Methoden zur Wiederverwendung im Allgemeinen, und Ablage von Komponenten im Speziellen, nicht überbewertet werden. Die nachfolgenden Kriterien beantworten dann auch die eingangs gestellte Frage nicht umfassend, aber sie helfen das Spektrum der im vorangegangenen Abschnitt vorgestellten Methoden unter zusätzlichen Aspekten zu betrachten.

Für die Bewertung der Leistungsfähigkeit von Informationssystemen geben Salton et al. [95] und Stürmer [107] Kriterien an. Daran orientieren sich zum Teil die Kriterien zur Bewertung von Systemen zur Ablage von Softwaresystemen von Mili et al. [72, 71], Henninger [45] und Krueger [56]. Die von den verschiedenen Autoren erhobenen Kriterien sind durchaus widersprüchlich und es scheint keine Übereinkunft über verbindliche Kriterien zu geben. Selbst die Aussagekraft der häufig verwendeten Kennzahlen *Precision* und *Recall* ist umstritten – die Probleme zur Bewertung von Informationssystemen werden ausführlich von Stürmer [107] dargestellt. Die nachfolgend vorgestellten Kriterien orientieren sich an Salton et al. [95] und Mili et al. [71]

4.3.1 Precision und Recall

Der Wunsch nach Kriterien und Maße für den Vergleich von Leistungsfähigkeit von Informationssystemen existiert, seit es diese Systeme selber gibt. Die er-

sten Informationssystem dienten Anfang der sechziger Jahre bibliographischen Zwecken und arbeiteten batch-orientiert, später kommandoorientiert mit textuellen Benutzungsschnittstellen. Die Erkenntnis, daß auch die Zugangsmöglichkeiten über den Wert von Informationen entscheiden, führte zusammen mit leistungsfähigeren Rechnersystemen zu dialogbasierten und interaktiven Informationssystemen. Die bekanntesten Maße zur Bewertung der Leistungsfähigkeit von Informationssystemen sind *Precision* und *Recall*, die die Genauigkeit und Vollständigkeit von Systemantworten in Bezug auf die Relevanz von Dokumenten messen. Sie stammen ursprünglich aus der Zeit batchorientierter Systeme; weil sie viele zwischenzeitlich ebenfalls als wichtig erachtete Faktoren ignorieren, wird ihre alleinige Aussagekraft heute bezweifelt.

Zur Bestimmung von Precision und Recall wird jedes Dokument einer Dokumentensammlung als relevant/unrelevant in Bezug auf ein Informationsbedürfnis eines Anwenders klassifiziert.

Definition 15 (Precision, Recall) *Sei $\mathcal{C}$ eine Menge von Dokumenten und $r : \mathcal{C} \to \{0, 1\}$ ein Prädikat, das die Relevanz von Dokumenten bezüglich einer gegebenen Fragestellung bestimmt. Dann sind die* Precision $P(C)$ *und der* Recall $R(C)$ *eines Teilmenge $C \subseteq \mathcal{C}$ definiert als:*

$$P(C) = \frac{|\{c|c \in C, r(c) = 1\}|}{|C|} \qquad R(C) = \frac{|\{c|c \in C, r(c) = 1\}|}{|\{c|c \in \mathcal{C}, r(c) = 1\}|}$$

Precision und Recall umfassen einen Wertebereich von 0 bis 1. Precision mißt den relativen Anteil der relevanten Dokumente einer Antwortmenge und Recall ihre Vollständigkeit im Vergleich zu allen relevanten Dokumenten. Beide Maße wirken in der Praxis gegeneinander, da hohe Präzision und hoher Recall sich im Trivialfall ($C = \emptyset$ beziehungsweise $C = \mathcal{C}$) nur auf Kosten des jeweils anderen Maßes erzielen lassen. Bei der Bewertung von Informationssystemen durch Precision und Recall werden gleichzeitig hohe Werte für beide Maße angestrebt. Eine Diskussion, welches Maß im Zweifel im Fall von Softwarekomponentenbibliotheken vorzuziehen ist, fehlt allerdings in der betrachteten Literatur.

Die Definition von Precision und Recall ist problematisch, weil sie von einer stark vereinfachten Relevanzauffassung ausgeht. Es existiert keine Einigkeit darüber, was Relevanz genau ist und ob die zur Definition von Precision und Recall nötige Klassifizierung in relevante und unrelevante Dokumente überhaupt konsistent ist. Stürmer [107] legt die Problematik und ihre Auswirkung auf relevanzbasierte Maße ausführlich dar; sie wurde auch von Mili et al. im Rahmen der Softwarewiederverwendung in [72] erkannt, in der jüngeren Veröffentlichung [71] aber nicht mehr erwähnt. Das zur Definition von Precision und Recall verwendete Modell von Relevanz setzt die beiden folgenden Eigenschaften von Relevanz voraus:

- Jedes Dokument kann als relevant oder nicht eingestuft werden.

- Die Relevanz eines Dokumentes ist unabhängig von anderen Dokumenten.

Dem stehen die folgenden Beobachtungen und Probleme bei der Beurteilung von Dokumentrelevanz entgegen:

- Relevanz wird von verschiedenen Nutzern unterschiedlich beurteilt. Vorwissen beeinflußt die Beurteilung besonders stark.
- Die Beurteilung von Relevanz ist sensitiv gegenüber der Präsentation mit anderen Dokumenten.
- Ist ein Dokument relevant, das nur mit Hilfe eines anderen verständlich ist?
- Ist ein Dokument relevant, das einen Verweis auf ein relevantes Dokument enthält?

Diese, und weitere bei Stürmer [107] dargestellte Probleme haben zu Forderungen nach anderen Maßen für die Bewertung von Informationssystemen geführt. Offensichtlich ist die Relevanz von Dokumenten weder binär, noch unabhängig von der Relevanz anderer Dokumente, wie es die Definition von Precision und Recall voraussetzt.

Eine Ausnahme bildet die Beurteilung von Relevanz bei formal-logischen Methoden (Abschnitt 4.2.4): diese erfassen formal die Semantik von Komponenten und erlauben deswegen auch eine semantisch fundierte Definition von Relevanz. Je umfassender die erfaßte Semantik ist, desto besser läßt sich Relevanz automatisch beurteilen.

Signature Matching [77, 92] betrachtet die Typ-Kompatibilität von Komponenten, *Specification Matching* [93, 119, 18] erfaßt darüberhinaus die Funktionalität von Komponenten. Unter Signature Matching erfüllt eine Komponente (P, Q) eine Anfrage (P_q, Q_q), wenn die logische Vorbedingungen P, P_q und Nachbedingungen Q, Q_q in einer zuvor gewählten logischen Relation $\sqsubseteq$ stehen. Eine diese Bedingung erfüllende Komponente kann berechtigt als relevant betrachtet werden.

Die meisten Anfrage- und Klassifikationsmethoden erfassen nur einen sehr kleinen Tei der Semantik einer Komponente. Deswegen fallen bei ihnen die Erfüllung einer Anfrage und ihre Relevanz in Bezug auf eine Fragestellung nicht zusammen. Die Bestimmung von Precision und Recall erfordern daher eine Testreihe mit möglichst vielen Teilnehmern, um statistisch signifikante Aussagen zu erhalten. Jede Anwender bearbeitet dazu eine Reihe von Anfragen, die etwa die folgenden Punkte umfassen:

1. Die natürlichsprachlich gegebene Formulierung der Anfrage. Sie drückt das Informationsbedürfnis des Anwenders aus.

Methode	Precision	Recall
Information Retrieval	○ ○ ○	○ ○ ○ ○
Deskriptive Methoden	○ ○ ○ ○	○ ○ ○ ○
Operationale Methoden	○ ○ ○ ○ ○	○ ○ ○ ○
Formal-logische Methoden	○ ○ ○ ○ ○	○ ○ ○ ○
Wissensbasierte Methoden	○ ○ ○ ○ ○	○ ○ ○ ○ ○
Strukturelle Methode	○ ○ ○ ○ ○	○ ○ ○ ○ ○

Tabelle 4.1: Precision und Recall verschiedener Methoden nach Mili et al. [71]. Precision und Recall sind um so größer, je mehr ○ (maximal fünf) einer Methode zugeordnet sind.

2. Die Bearbeitung der Anfrage durch den Anwender mit dem System. Dies schließt die Umsetzung der Anfrage in die Eingabesprache des Systems ein.

3. Die Beurteilung der Relevanz aller Dokumente in der Bibliothek in Bezug auf die Anfrage durch den Anwender und die Berechnung von Precision und Recall der Systemantwort.

Der Beurteilung der Relevanz durch den Anwender ist nötig, weil Relevanz subjektiv empfunden wird. Da Untersuchungen von Precision und Recall ohnehin aufwendig sind, wird dieser Punkt häufig vereinfacht: die Relevanz von Dokumenten bezüglich einer Fragestellung wird global durch einen Experten vorgenommen und nicht individuell durch jeden Anwender. Die wenigen vorgenommenen Messungen von Precision und Recall bei Softwarekomponentenbibliotheken [1, 69] wiesen dieses Testdesign auf, allerdings ohne den Begriff der Relevanz zu problematisieren.

Mili et al. haben Precision und Recall der verschiedenen Methoden zur Organisation von Softwarekomponentenbibliotheken verglichen und in fünf Stufen charakterisiert [71]. Sie sind dabei zu dem Ergebnis in Tabelle 4.1 gekommen. Die Beurteilung der verschiedenen Methoden erfolgte nicht an Hand durchgeführter Messungen sondern durch Auswertung der verfügbaren Literatur und durch prinzipielle Überlegungen zu den einzelnen Verfahren. Danach können prinzipiell alle Methoden hohe Werte für Precision und Recall erreichen; je größer der manuelle und intellektuelle Aufwand bei der Indexierung und Anfrage ist, desto eher ordnen Mili et al. den Methoden auch sehr hohe Potentiale zu.

Formal-logische Methoden, und zum gewissen Grad operationale Methoden, unterscheiden sich von den anderen Methoden darin, daß sie unentscheidbare oder semi-entscheidbare Verfahren verwenden. Im Falle von formal-logischen Methoden kommen zum Beispiel Theorembeweiser zum Einsatz, die nicht immer eine vorhandene logische Beziehung zwischen Anfrage und einer Komponente nachweisen können. Während die anderen Methoden zu einer Anfrage die durch diese Anfrage bestimmten Komponenten exakt ermitteln, ist dies bei formal-logischen

und operationalen Methoden (durch Nichtterminierung, Fehler) nicht notwendig der Fall. Da dieses Effekt Precision und Recall beeinträchtigt, wird er von Fischer et al. für deduktionsbasierte Komponentensuche gesondert untersucht [18].

In Übereinstimmung mit der Untersuchung von Frakes et al. zur Repräsentation von Softwarekomponenten in Bibliotheken [24] spiegelt Tabelle 4.1 wider, daß die Wahl der Methode nur untergeordneten Einfluß auf Precision und Recall besitzt. Diese werden vor allen Dingen durch die konkrete Implementierung und das Zusammenspiel von Komponenten und ihre Indexierung bestimmt.

4.3.2 Benutzungsschnittstelle

Die Benutzungsschnittstelle erlaubt einem Endanwender auf die Komponenten einer Softwarekomponentenbibliothek zuzugreifen. Neben der Qualität der Softwarekomponenten bestimmt sie maßgeblich die Akzeptanz einer Komponentensammlung. Außer der Benutzungsoberfläche für den Anwender von Softwarekomponenten existiert eine weitere für den Aufbau, den Betrieb und die Pflege einer Komponentensammlung. Sie ist an dieser Stelle nicht gemeint; da sie seltener und nur von einer kleinen Gruppe von Spezialisten benutzt wird, kommt ihr eine geringere Bedeutung zu.

Welche Operationen ein Benutzer mit Hilfe der Benutzungsschnittstelle einer Komponentensammlung ausführen kann, wird durch ihre interne Organisation begrenzt. Da diese einen relativ geringen Einfluß auf die Qualität der Suchergebnisse besitzt (siehe vorhergehender Abschnitt), sollte die Methode zur Organisation nach den Bedürfnissen an ihrer Benutzungschnittstelle ausgewählt werden. Die für Benutzer wichtigsten Operationen sind häufig eine direkte Folge des Softwareentwicklungsprozesses, an dem sie teilnehmen. Softwareentwicklungsprozesse können sich beispielweise in den folgenden Punkten unterscheiden:

- Die typische Zeitdauer, die ein Entwickler an einem Projekt teilnimmt.
- Größe der Entwicklergruppe und ihre geographische Verteilung.
- Verhältnis von Wiederverwendung und Neuentwicklung von Software im Projekt.
- Entwicklungsmethodik: formal, semi-formal, informal.
- Anzahl der verwendeten Implementierungssprachen.
- Erfahrung der beteiligten Entwickler.

Entwickler in kurzlebigen Projekten und hoher Entwicklerfluktuation werden Wert auf kurze Einarbeitungszeit und Übersichtlichkeit legen, während in lange laufenden Projekten mit formalen Methoden Wünsche nach Genauigkeit, Adaptierbarkeit an eigene Bedürfnisse und großer Funktionsumfang wahrscheinlicher sind.

Anwendungsszenarien

Für einen Anwender ist die gezielte Suche nach Softwarekomponenten nicht die einzige Motivation für die Arbeit mit einer Softwarekomponentensammlung. Mindestens drei verschiedene Szenerien für ihre Verwendung existieren:

Referenz Der Benutzer kennt eine bestimmte Softwarekomponente bereits und möchte mehr über ihre Details erfahren. Er wünscht die Komponente durch eine gezielte Auswahl oder Suche präsentiert zu bekommen.

Suche Der Benutzer sucht die Implementierung zu einer Problemlösung, ist sich über ihre Verfügbarkeit aber nicht sicher. Dies ist der Ausgangspunkt einer häufig mehrstufigen Suche, in deren Verlauf Komponenten inspiziert und neue Anfragen gestellt oder alte Anfragen umformuliert werden.

Browsing Um die Implementierung eines Projektes zu vereinfachen, möchte ein Benutzer möglichst auf wiederverwendbare Komponenten zurückgreifen. Deshalb möchte er sich in einer Planungsphase über die Struktur und den Inhalt einer vorhandenen Komponentensammlung informieren. Er sucht nicht, wie in den vorhergehenden beiden Szenarien, gezielt nach Komponenten sondern nach übergeordneten Strukturen, denen er folgen kann. Dabei möchte er auch einzelne Komponenten im Detail ansehen.

Jedes Szenario stellt eigene Anforderungen an die Benutzungsschnittstelle. Die *Suche* muß den Benutzer bei der Formulierung seines anfänglich unspezifischen Zieles unterstützen und im Verlauf der Suche seine Reformulierung erlauben. Dies soll gleichzeitig nicht den schnellen und gezielten Zugang zu Komponenten im Szenario *Referenz* verstellen. Das dritte Szenario *Browsing* verlangt eine thematische Strukturierung der Komponenten, die dem Benutzer offengelegt wird, so daß er sich an ihr entlangbewegen, und dabei einzelnen Komponenten inspizieren kann.

Um die verschiedenen Szenarien und einen reibungslosen Wechsel zwischen ihnen zu unterstützen, kann eine Benutzungsoberfläche gleichzeitig verschiedene Zugänge zu Softwarekomponenten bieten. Die Interaktionselemente der Benutzungsoberfläche (wie Menüs, Textfelder, Knöpfe, Hypertext, Schieberegler) unterscheiden sich in dem Grad, in dem sie die Form (Syntax) einer Eingabe vorbestimmen. Die freiste Form der Eingabe bieten Textfelder: sie erheben keinerlei syntaktische Einschränkung. Eine syntaktische und semantische Überprüfung findet in einem separaten Schritt nach Abschluß der Eingabe statt. Suchbegriffe, natürlichsprachliche Eingaben und komplexe Terme formaler Spezifikationen lassen sich textuell gut erfassen. Der Preis für die maximale Freiheit der Eingabe ist, daß ein Benutzer die Eingabesyntax kennen muß, da er sie nicht ablesen kann. Das andere Extrem bildet die Auswahl aus einem Menü: die Form der Auswahl ist vorgegeben (durch Tastendruck, Selektion mit einem Cursor) – der Benutzer

kann sich auf die inhaltliche Frage konzentrieren, *was* er möchte. Gleichzeitig ist nur eine begrenzte Auswahl von Möglichkeiten vorgegeben, die möglicherweise den direkten Weg zu einer Komponente verstellt und deshalb mehrere Selektionsschritte erfordert. Eine Mischform bilden berechneten Auswahlmöglichkeiten: sie stehen dem Benutzer nur in bestimmten Kontexten zur Verfügung, legen die Syntax der Auswahl aber fest. Sie sind häufig eine Systemantwort auf vorangegangene Aktionen des Benutzers und bieten ihm weitere Schritte an.

Schnittstellen mit starker syntaktischer Unterstützung erleichtern die Einarbeitung in ein System und sind deswegen für Gelegenheitsanwender geeignet. Textuelle Schnittstellen sind leicht an eigene Bedürfnisse anzupassen, erlauben komplexe Eingaben und sind deshalb für geübte Anwender tendenziell besser geeignet. Der Grad der syntaktischen Unterstützung wird zusätzlich durch das aktuelle Verwendungsszenario bestimmt: die Organisationsstruktur einer Komponentensammlung muß dem Benutzer im *Browsing*-Szenario sichtbar sein, so daß sie sich zur Ausführung von Navigationsaktionen anbietet. Das gezielte Aufsuchen einer Komponente (*Referenz*) kann zum Beispiel durch die freie Eingabe von Text direkt, ohne Umwege über Menüs erfolgen.

Iterative Suche

Die Suche nach einer Softwarekomponente entstammt häufig einer nur ungefähren Vorstellung nach der benötigten Funktionalität. Bei der Umsetzung dieses Wunsches in eine Anfrage an eine Softwarekomponentensammlung kann es zu weiteren Differenzen zwischen dem Wunsch des Benutzers und der Semantik seiner tatsächlichen Eingabe kommen. Gründe dafür können die ungenügende Ausdruckskraft der Anfragesprache, mangelnde Übung mit ihrem Umgang, oder die Fehlinterpretation von verwendeten Begriffen sein. Man kann also nicht davon ausgehen, daß eine Anfrage den Wunsch eines Benutzers vollständig und exakt beschreibt und ihm die Bedeutung seiner Anfrage vollständig bewußt ist. Gerade Letzteres ist nur selten möglich, da eine Reihe von Systemen eine nur operational definierte Semantik besitzen oder Heuristiken verwenden (vergleiche Abschnitt 4.2, Seite 56). Mili et al. beurteilen in Tabelle 4.2 dementsprechend die Durchschaubarkeit (*Transparency*) der verschiedenen Methoden sehr unterschiedlich. Systemantworten mit nur exakt zu der Anfrage passenden Komponenten führen wegen diesen Schwierigkeiten zur Enttäuschung der Anwender. Besser sind Systeme, die Komponenten in Abhängigkeit von ihrer Ähnlichkeit zu der Anfrage präsentieren; der Anfrage am ähnlichste Komponenten werden dabei bevorzugt präsentiert.

Die angesprochene Unschärfe bei der Formulierung einer Anfrage zusammen mit der ebenfalls vorhandenen unscharfen Indexierung von Komponenten führt häufig nicht zu den gesuchten Komponenten. Trotzdem ist die Systemantwort für den Benutzer hilfreich, da sie ihm die Bedeutung seiner Anfrage explizit macht. Er kann sie zur Re-Formulierung seiner Anfrage verwenden, um so seinem Ziel

Methode	Durchschaubarkeit	Benutzungsschwierigkeit
Information Retrieval	○ ○ ○ ○	○ ○ ○
Deskriptive Methoden	○ ○ ○ ○ ○	○
Operationale Methoden	○ ○ ○ ○ ○	○ ○
Formal-Logische Methoden	○ ○ ○	○ ○ ○
Wissensbasierte Methoden	○ ○ ○ ○ ○	○ ○ ○ ○ ○
Strukturelle Methode	○	○

Tabelle 4.2: Durchschaubarkeit und Schwierigkeit der Benutzung nach Mili et al. [71]. Bei der Durchschaubarkeit zeigen mehr Punkte (○) eine bessere Beurteilung an, bei den Schwierigkeiten weniger.

näher zu kommen. Eine Suche besteht also nicht aus einem einzelnen Schritt von der Suche zum Ergebnis, sondern ist ein iterativer Prozeß aus Formulierung, Komponenteninspektion und Reformulierung der Anfrage, den eine Benutzungsoberfläche unterstützen sollte.

Zur Unterstützung der Umformulierung und Verfeinerung von Anfragen existieren eine Reihe von Techniken. Die bekannteste ist *Relevance-Feedback* [94], bei der ein Benutzer nach einer initialen Anfrage unter den präsentierten Komponenten relevante und nicht relevante Dokumente markiert. Die nächste Anfrage wird mit Informationen aus der Indexierung der als relevant markierten Komponenten angereichert. Dieser Prozeß kann auch automatisiert werden, indem automatisch die vom System als besonders relevant beurteilten Komponenten zur Verfeinerung der Anfrage herangezogen werden. Eine weitere Informationsquelle bilden in der Vergangenheit abgegebene ähnliche Anfragen [21].

Wenn die Komponenten einer Sammlung semantisch sinnvoll (partiell) geordnet sind, bietet diese Ordnung ebenfalls natürliche Wege zur Verfeinerung und Aufweitung einer Anfrage an. Unabhängig von dem aktuellen Verfahren ist eine Hilfe zur Umformulierung für den Benutzer in den beiden Extremfällen wichtig, wenn das System zu viele oder zu wenig Komponenten präsentiert. Da die Unterstützung zur Umformulierung von dem System ausgeht, kann sie durch den Benutzer in der Regel durch die Auswahl einer oder mehrerer angebotener Optionen wahrgenommen werden. Dies führt dazu, daß nach der möglicherweise textuell formulierten initialen Anfrage, alle weiteren Schritte syntaktisch vorgegeben sind, und damit das Risiko einer Fehleingabe minimiert wird.

Mili et al. haben in ihrer Studie [71] die Benutzungsschwierigkeit (*difficulty of use*) verschiedener Komponentenbibliotheken untersucht; die Ergebnisse sind in Tabelle 4.2 dargestellt. Dabei haben sie nicht die verschiedenen Situationen und Intentionen unterschieden, in und mit denen Softwarekomponentenbibliotheken verwendet werden; die Ergebnisse geben deswegen nur eine Tendenz ab. Nichtsdestotrotz ist ihre Untersuchung die einzige in der Literatur, die mehrere Methoden zur Komponentenablage unter diesem Gesichtspunkt untersucht.

4.3.3 Kosten und Nutzen

Die Einführung von Softwarewiederverwendung in einen Softwareprozeß zielt auf eine Qualitäts- und Produktivitätssteigerung und damit letztlich auf eine Kostenersparnis bei der Erstellung von Software. Softwarewiederverwendung selbst verursacht bei ihrer Einführung und im laufenden Betrieb einen zusätzlichen Aufwand – sowohl finanziell als auch intellektuell. Dieser Aufwand unterscheidet sich bei den einzelnen Methoden so sehr, daß er ein weiteres Kriterium für ihre Auswahl wird.

Die Einführung einer komponentenbasierten Wiederverwendung verlangt zunächst die Auswahl einer Methode, ihre Implementierung und den Entwurf der Komponentenorganisation. Natürlich müssen die Komponenten selbst bereitgestellt und indexiert werden, und ihre Anwender müssen sich mit ihnen, ihrer Organisation und ihrem Ablagesystem vertraut machen. Der laufende Betrieb schließlich umfaßt die Pflege und Ergänzung von Komponenten; dabei kann auch eine Reorganisation durch sich ändernde Anforderungen nötig werden.

Methode	Anfangs-investitionen	laufende Betriebskosten
Information Retrieval	○	○ ○
Deskriptive Methoden	○ ○ ○ ○	○ ○ ○ ○
Operationale Methoden	○ ○	○ ○ ○
Formal-Logische Methoden	○ ○ ○ ○	○ ○ ○ ○
Wissensbasierte Methoden	○ ○ ○ ○ ○	○ ○ ○ ○ ○
Strukturelle Methode	○ ○ ○	○ ○

Tabelle 4.3: Investitions- und Betriebskosten nach Mili et al. [71]. Größere Kosten werden durch mehr Punkte repräsentiert.

Mili et al. haben die Kosten zur Einrichtung und zum Betrieb verschiedener komponentenbasierter Ablagesysteme verglichen [71]; ihre Ergebnisse sind in Tabelle 4.3 zusammengefaßt. Henninger begründet in *An Evolutionary Approach to Constructing Effective Software Reuse Repositories* [45], warum der Aufwand zur Etablierung von Softwarewiederverwendung in jedem Schritt einen unmittelbaren Vorteil versprechen muß. Insbesondere fordert er die Evolutionsfähigkeit der gewählten Lösung, um, analog zu Software, spätere und teure Umstrukturierungen zu vermeiden. Auch der Aufwand für einen Benutzer, um an gesuchte Informationen und Komponenten zu gelangen, ist für ihn letztlich ein ökonomischer Aspekt. Er setzt sich deshalb für leicht erlernbare und fehlertolerante Anfragemechanismen ein, die dem Benutzer durch Rückkopplung Hinweise zu seine Aktionen geben.

Den ökonomischen Forderungen wurde bisher in der Literatur kaum Aufmerksamkeit geschenkt. Und so mag es kein Zufall sein, daß nur die von Mili et al. als besonders kostengünstig beurteilte Methode des *Information Retrieval*

eine weite praktische Verwendung hat. Die anderen (nicht komponentenbasierten) Methoden mit größerer Verwendung sind *Design Patterns* und *Generatoren* (vergleiche Abschnitte 4.1.1, 4.1.4), die zumindest in ihrer Anwendung ebenfalls als kostengünstig gelten können.

Niedrige Investitions- und Betriebskosten erfordern einfache Konzepte oder eine durchgehende Automatisierung. Beides wird von Information-Retrieval- (Abschnitt 4.2.1) und operationalen Methoden (Abschnitt 4.2.3) erfüllt. Dabei beschreiben Komponenten sich durch ihre Dokumentation und ihre Verhalten selbst, so daß eine zusätzliche Indexierung oder Organisation ihrer Ablage unnötig ist. Das Wirkungsprinzip beider Methoden ist leicht einsichtig; es erfordert keine besondere Ausbildung seiner Anwender. Außerdem erfüllen beide Methoden Henningers Forderung nach Evolutionsfähigkeit. Gerade weil nur wenige zusätzlichen Strukturen beim Aufbau einer Komponentensammlung eingeführt werden, passen sie sich neuen Anforderungen an.

Auf der anderen Seite des Aufwand-Spektrums stehen formal-logische, und nach Mili et al. auch wissensbasierte, Methoden (Abschnitte 4.2.5, 4.2.4). Formal-logische Methoden verlangen eine durchgehende Formalisierung von Komponenten-Indexierung, Bibliotheksorganisation und Abfragen. Da Komponenten normalerweise ihre Formalisierung nicht enthalten, muß sie manuell beim Aufbau einer Bibliothek ergänzt werden. Gleiches gilt für die Prinzipien, nach denen eine Bibliothek organisiert wird. Im Vergleich zu den anderen Ansätzen werden formal-logische Methoden durch komplexe, und zum Teil sogar unentscheidbare, Algorithmen implementiert, deren Einsatz nennenswerte Hardware-Resourcen erfordert. Der Umgang mit formal-logischen Methoden erfordert eine spezielle Ausbildung, da sich ihre Wirkungsweise und die Präzision ihrer Ergebnisse sonst nicht erschließt. Leider garantieren der hohe Initialaufwand in Ausbildung, Indexierung und Implementierung nicht notwendig die Evolutionsfähigkeit. Im Falle einer Umstrukturierung fällt ein beträchtlicher Teil des Aufwandes für Organisation und Indexierung nochmals an.

Die ungünstigere Beurteilung wissensbasierter Methoden im Vergleich zu formal-logischen durch Mili et al. ist nur bedingt nachvollziehbar. Das System von Henninger [45], der die ökonomischen Aspekte besonders betont, ist wissensbasiert und verwendet komplexe Datenstrukturen und Algorithmen, ähnlich wie formal-logische Methoden. Dafür bietet es weitreichende Automatismen und Hilfestellungen beim Erfassen und Suchen von Komponenten an und eine automatische Adaptierung der Organisationsstruktur durch die Anfragen soll die Evolutionsfähigkeit der Sammlung sichern. Da die Qualität der Benutzungsschnittstelle entscheidenden Einfluß auf die Akzeptanz eines Systemes ausübt (Abschnitt 4.3.2), sind die auf den ersten Blick günstigsten Methoden mit sehr einfachen Schnittstellen nicht zwangsläufig die langfristig besten.

Kostengünstige Methoden zur Ablage von Softwarekomponenten verzichten weitgehend auf eine manuelle Indexierung und nutzen möglichst die inhärenten Eigenschaften wie Dokumentation, Verhalten oder Typen von Komponenten.

Nicht nur bei der Einrichtung, sondern auch im Falle einer Reorganisation zahlt sich der geringere manuelle Aufwand aus. Ob sich der erhöhte Aufwand von deskriptiven, wissensbasierten oder formal-logischen Methoden im Vergleich zu den anderen Methoden lohnt, ist offen. Bislang existieren nur Erkenntnisse über weitgehend vergleichbare Werte von Precision und Recall [24], aber nicht über Wiederverwendungsraten insgesamt. Die Skepsis vieler Praktiker gegenüber „zu komplizierten“ Methoden mag im Allgemeinen also gerechtfertigt sein. Dies gilt selbstverständlich nicht für Spezialanwendungen wie sicherheitskritischer Software, in denen Aufgrund der Folgekosten von Fehlern eine andere Ökonomie herrscht.

Kapitel 5

Begriffsbasierte Komponentensammlungen

Das vorangegangene Kapitel hat eine Vielzahl von Möglichkeiten und Werkzeugen vorgestellt, die die Wiederverwendung von Software fördern. Die Mehrzahl der Ansätze stützt sich auf die Ablage von (Quelltext-) Komponenten, um so ihre spätere Wiederverwendung zu ermöglichen. Eine solche Ablage kann mittels formaler Begriffe organisiert werden; die Begriffsanalyse dient dabei gleichzeitig als unterliegendes theoretisches Modell, das eine abstrakte und genaue Beschreibung aller Vorgänge ermöglicht, und, in Form von begrifflichen Algorithmen und Datenstrukturen, als Implementierung [64, 65, 66].

Eine Sammlung von Komponenten mit assoziierten Schlüsselwörtern (oder anderen Eigenschaften) bildet einen formalen Kontext. Darin können Komponenten durch Anfragen gesucht werden, deren Ergebnis Komponentenmengen sind. Eine geeignete Definition von Anfragen vorausgesetzt, beschreiben die Begriffe des Kontextes Paare aus Anfrage und Ergebnis. Diese Eigenschaft erlaubt den Entwurf einer Komponentenablage mit bemerkenswerten Eigenschaften:

1. Der Begriffsverband einer Sammlung verhält sich zu seinem Kontext wie Objektcode zu seinem ursprünglichen Quelltext: die Vorberechnung von Ergebnissen gestattet, Abfragen durch Rückgriff auf vorberechnete Daten effizient zu bearbeiten.

2. Der Begriffsverband ist eine automatisch gewonnene, semantisch fundierte, Organisations- und Navigationsstruktur der Komponentensammlung.

3. Da der Begriffsverband die wesentlichen Eigenschaften aller möglichen Anfrage- und Ergebniskombinationen für eine Komponentensammlung enthält, kann sie mit Hilfe ihres Verbandes global statisch analysiert werden. So können statistische Aussagen über das Anfrage/Ergebnisverhalten der Sammlung genauso gewonnen werden, wie Hinweise auf eine ungenügende oder zu feine Indexierung.

4. Die durchgehend formale Beschreibung der Operationen auf der Komponentensammlung durch Begriffsanalyse gestattet die Verifikation einer Implementierung.

Dieses Kapitel stellt die Organisation einer kleinen Komponentenbibliothek durch Begriffe zunächst als Beispiel aus Anwendersicht vor. Danach werden die bereichsspezifischen Objekte und Operationen formalisiert, und auf Begriffe zurück geführt. Die Verbindung zur Begriffsanalyse steht dabei im Vordergrund, da die Implementierung begrifflicher Operationen bereits in Abschnitt 3 beschrieben wurde. In dem letzten Abschnitt dieses Kapitels werden zwei praxisnahe Komponentensammlungen mit mehreren hundert Komponenten an Hand ihrer Begriffsverbände global statistisch analysiert.

5.1 Beispiel

Als Inhalt einer kleinen Komponentensammlung sollen 12 Unix Systemaufrufe (*System Calls*) dienen. Für jedem Systemaufruf kann man den Handbüchern des SunOS-Unix eine einzeilige Beschreibung entnehmen; sie ist für jeden der 12 Systemaufrufe in Tabelle 5.1 aufgeführt. Die Indexierung der Komponenten besteht aus der Zuordnung einer Menge von Schlüsselwörtern zu jedem Systemaufruf, wobei die Schlüsselwörter aus der einzeiligen Beschreibung stammen. Dies dient lediglich der Demonstration des Prinzips: Schlüsselwörter können genauso frei vergeben werden oder statt der Schlüsselwörter können andere Eigenschaften wie die Datentypen von Parametern den Aufrufen zugeordnet werden. Systemaufrufe und Schlüsselwörter bilden offensichtlich eine binäre Relation, und damit einen formalen Kontext.

Call	Kurzbeschreibung aus SunOS Handbuch	Indexierung
`chmod`	*change mode of file*	change mode permission file
`chown`	*change owner and group of a file*	change owner group file
`stat`	*get file status*	get file status
`fork`	*create a new process*	create new process
`chdir`	*change current working directory*	change directory
`mkdir`	*make a directory file*	create new directory
`open`	*open or create a file for reading or writing*	open create file read write
`read`	*read input*	read file input
`rmdir`	*remove a directory file*	remove directory file
`write`	*write output*	write file output
`creat`	*create a new file*	create new file
`access`	*determine accessibility of file*	check access file

Tabelle 5.1: Unix-System-Calls mit Kurzbeschreibung und Indexierung

Eine Anfrage ist eine Menge von Schlüsselwörtern (oder Attributen). Sie wählt die Komponenten aus der Menge aller Komponenten aus, die mindestens mit den Schlüsselwörtern der Anfrage indexiert sind. Eine Anfrage wirkt also einschränkend: je mehr Elemente sie enthält, desto mehr Anforderungen muß eine Komponente erfüllen, die zum Ergebnis dieser Anfrage gehört.

In Abbildung 5.1 ist die prototypische Implementierung[1] einer Benutzungsoberfläche gezeigt, in der Komponenten durch die Auswahl von Schlüsselwörtern selektiert werden. Die Schlüsselwörter zur Ergänzung der aktuellen Anfrage werden mit dem Maus-Cursor in der linken oberen Liste ausgewählt und dadurch in die rechte Liste übernommen, die alle Schlüsselwörter der momentanen Anfrage enthält. Die Abbildung zeigt den Übergang von einer leeren Anfrage zu der Anfrage mit dem Schlüsselwort *change*. Das Ergebnis der aktuellen Anfrage ist die untere Liste von Komponenten.

Die Selektion einer Komponente in der Ergebnisliste aktiviert die Anzeige ihrer Dokumentation. Auf die gleiche Weise könnte auch ihr Quellcode, ihre Deklaration oder eine andere mit ihr verknüpfte Information zugänglich gemacht werden. Abbildung 5.1 zeigt als Beispiel auf der rechten Seite die Dokumentation für den Systemaufruf *chown*, der in dem aktuellen Ergebnis enthalten ist. Die anschließende Auswahl des Schlüsselwortes *file* verkleinert das Ergebnis entsprechend, da sie mehr Anforderungen an die Ergebniskomponenten stellt. Das Ergebnis enthält zwei Komponenten und ist in Abbildung 5.2 zu sehen.

Für die Verfeinerung einer Anfrage ist nicht jedes Schlüsselwort geeignet oder sinnvoll: nachdem *change* als ersten Schlüsselwort gewählt wurde, ist es nicht sinnvoll nach Komponenten zu fragen, die zusätzlich mit *access* indexiert sind: es existieren keine. Deshalb bietet die (linke) Liste nur Attribute zur Auswahl an, die ein nicht leeres Ergebnis *garantieren*. Der Anwender wird also an Anfragen vorbeigeleitet, die zu einem leeren oder unveränderten Ergebnis führen.

Die Spezialisierung einer Anfrage durch die Hinzunahme eines Attributes hat eine Untermenge des bisherigen Ergebnis als neues Ergebnis, da das hinzugekommene Attribut zusätzliche Anforderungen an die Ergebniskomponenten stellt. Gleichzeitig mit der Verkleinerung der Ergebnismenge verkleinert sich auch die Menge der zur Verfeinerung angebotenen Attribute in jedem Schritt. Dies führt in Abbildung 5.2 nach der Aufnahme von *file* und anschließend *mode* zu einer Situation, in der keine Schlüsselwörter für eine weitere Verfeinerung zur Verfügung stehen. Der in den Abbildungen 5.1 und 5.2 gezeigte Ablauf einer schrittweise Suche ist zusammen mit einem weiteren Beispiel in Tabelle 5.2 zusammengefaßt.

Die Berechnung des Ergebnis einer Anfrage und der für weitere Schritte noch sinnvollen Schlüsselwörter kann zum Zeitpunkt der Anfrage, oder mit Hilfe des Begriffsverbandes vorab erfolgen. In dem zweiten Fall können Anfragen effizienter bearbeitet werden, da die zuvor abgelegten Ergebnisse nur noch wiedergefunden,

[1] die gezeigte Implementierung entstand in Zusammenarbeit mit Olaf Püschel und basiert auf einer Erweiterung von Tcl [81], ähnlich TkConcept [67].

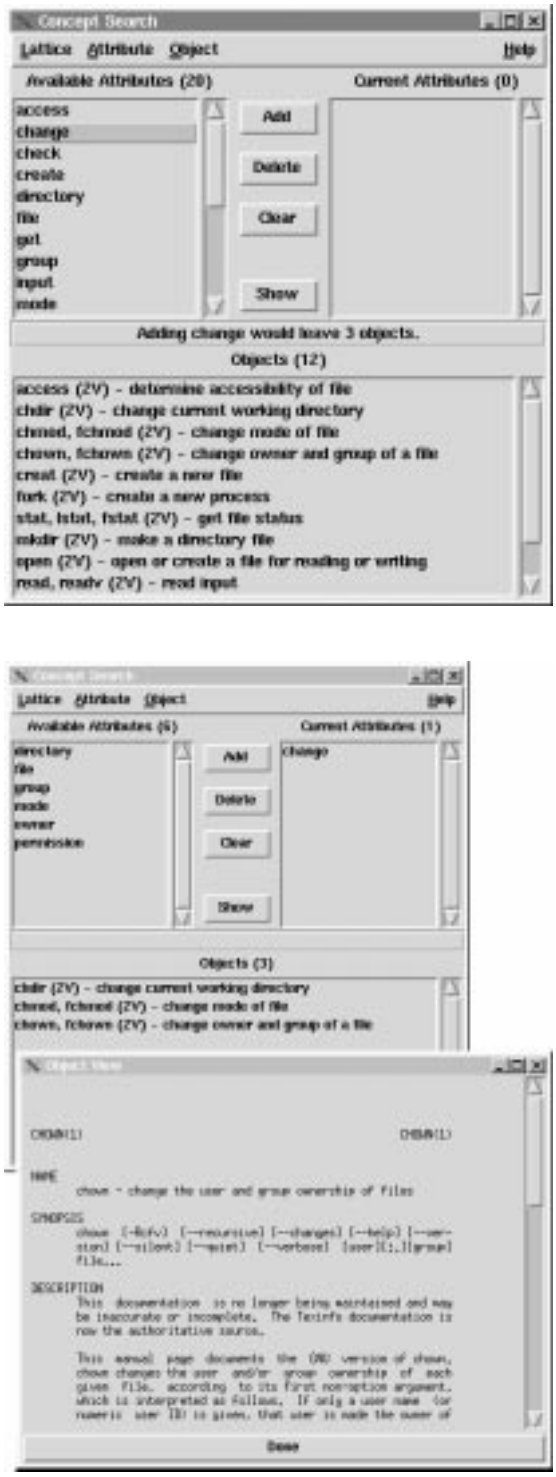

Abbildung 5.1: Selektion von Komponenten durch Auswahl von Attributen.

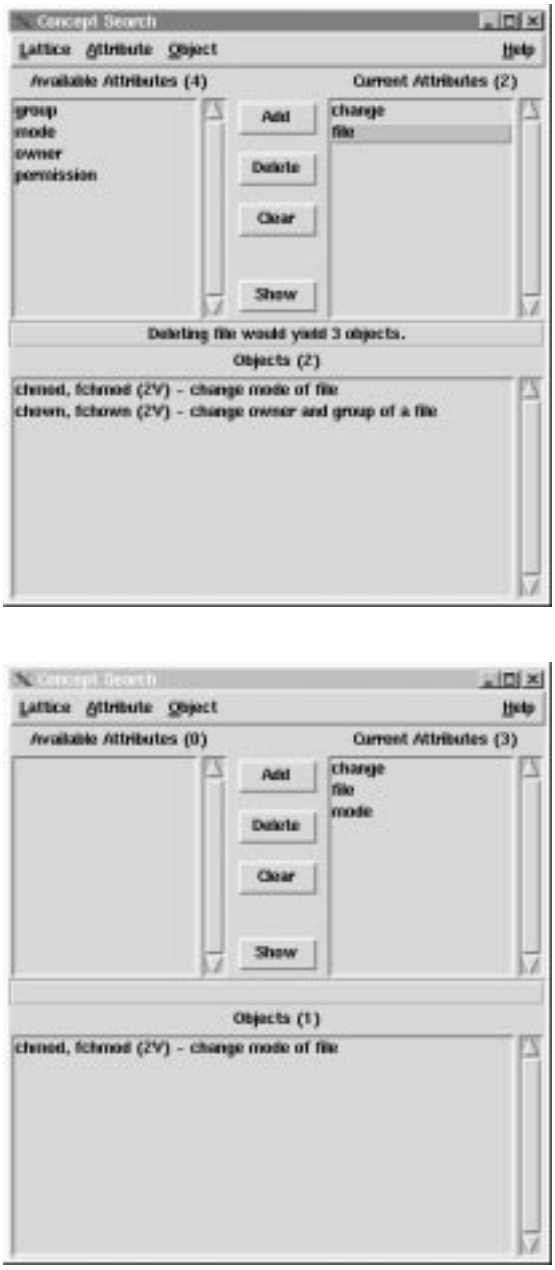

Abbildung 5.2: Die Auswahl von Schlüsselwörtern verkleinert das Ergebnis und die Anzahl wählbarer Schlüsselwörter für weitere Schritte.

aber nicht selbst berechnet werden müssen.

Weil der Begriffsverband einer Sammlung sämtliche Anfragesituationen repräsentiert, können diese vollständig analysiert werde – zum Beispiel mit statistischen Methoden. Als ein Beispiel zeigt Tabelle 5.3 die Verteilung von Anfragen über der Größe von Ergebnismengen für die Systemaufrufe aus Tabelle 5.1: insge-

Schritt	Schlüsselw.	Ergebnis	noch mögliche Schlüsselw.
1	–	alle	alle
2	*change*	`chdir chmod chown`	*directory file group mode owner permission*
3	*change file*	`chmod chown`	*group mode owner permission*
4	*change file mode*	`chmod`	–
1	–	alle	alle
2	*create*	`creat fork mkdir open`	*directory file new open process read write*
3	*create file*	`creat open`	*new read open write*
4	*create file read*	`open`	–

Tabelle 5.2: Zwei Beispiele für schrittweise Suche

Ergebnisgröße	Begriffe mit dieser Ergebnisgröße
1	12
2	4
3	3
4	1
9	1
12	1
Summe	22

Tabelle 5.3: Verteilung der Ergebnisgröße für die Beispielsammlung.

samt existieren 22 verschiedene Anfrage-Situationen, in 4 von ihnen umfaßt das Ergebnis genau 2 Komponenten.

Die nachfolgenden Abschnitte gehen formal auf den Zusammenhang zwischen Anfrage und Ergebnis ein. Darüberhinaus stellen sie die Möglichkeiten einer interaktiven und iterativen Verfeinerung von Anfragen genauer vor.

5.2 Anfrage und Ergebnis

In einer Komponentensammlung, in der jede Komponente mit einer Menge von Attributen indexiert ist, bilden Komponenten und Attribute eine binäre Relation, die als formaler Kontext aufgefaßt werden kann. Zu den Unix-Systemaufrufen des vorangegangenen Abschnitts und ihren Schlüsselwörtern gehört zum Beispiel der Kontext in Tabelle 5.4.

Eine Anfrage $A \subseteq \mathcal{A}$ an einen Kontext $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ selektiert alle Komponenten (oder Objekte) $O \subseteq \mathcal{O}$, deren Indexierung *mindestens* A umfaßt. Die Indexierung einer Komponente $o \in \mathcal{O}$ ist die Zeile o in der Kontexttabelle und

	access	*change*	*check*	*create*	*directory*	*file*	*get*	*group*	*input*	*mode*	*new*	*open*	*output*	*owner*	*permission*	*process*	*read*	*remove*	*status*	*write*
access	×		×			×														
chdir		×			×															
chmod		×				×				×					×					
chown		×				×		×						×						
creat				×		×					×									
fork				×							×					×				
fstat						×	×												×	
mkdir				×	×						×									
open				×		×						×					×			×
read						×			×								×			
rmdir					×	×												×		
write						×							×							×

Tabelle 5.4: Kontexttabelle *Unix-System-Calls* für Systemaufrufe des Betriebssystems Unix

die Menge $\{o\}'$. Dementsprechend wird eine Anfrage und ihr Ergebnis definiert:

Definition 16 (Anfrage und Ergebnis) *In einem Begriffsverband* $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ *ist eine* Anfrage *eine Menge* $A \subseteq \mathcal{A}$ *von Attributen. Ein Objekt* $o \in \mathcal{O}$ *erfüllt eine Anfrage, wenn es mindestens mit den Attributen der Anfrage indexiert ist, also wenn* $\{o\}' \supseteq A$ *gilt. Alle Komponente, die eine Anfrage erfüllen, bilden das* Ergebnis $[\![A]\!]$ *einer Anfrage:*

$$[\![A]\!] := \{o \mid o \in \mathcal{O}, \{o\}' \supseteq A\}$$

Das Ergebnis einer Anfrage kann mit Hilfe des Begriffsverbandes $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ einer Sammlung bestimmt werden. Zu jedem Attribut $a \in A$ der Anfrage existiert ein Attributbegriff $\mu(a)$, der alle Objekte enthält, die mit mindestens a indexiert sind. Das Infimum aller Attributbegriffe der Attribute aus A ist folglich der größte Begriff, der alle Objekte enthält, die mit *allen* Attributen aus A indexiert sind.

Abbildung 5.3 illustriert den Zusammenhang zwischen Attributbegriffen und Ergebnis: die Attributbegriffe a_1 und a_2 einer Anfrage $A = \{a_1, a_2\}$ sind in dem Begriffsverband verteilt. Ihr Infimum (Pfeil) markiert einen Unterverband und die darin enthaltenen Objektbegriffe bilden das Ergebnis der Anfrage. Wird eine Anfrage um ein Attribut a_3 erweitert, muß ein neues Infimum aus dem alten Infimum und $\mu(a_3)$ bestimmt werden, das dann einen kleineren Unterverband bestimmt.

Theorem 12 *Sei* $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ *ein Begriffsverband und sei* $A \subseteq \mathcal{A}$ *eine Anfrage, dann ist* $[\![A]\!] = O_1 = A'$*, wobei* $(O_1, A_1) = \bigwedge_{a \in A} \mu(a)$*. Der Begriff* $\bigwedge_{a \in A} \mu(a)$ *wird als der* Ergebnisbegriff *von* A *bezeichnet.*

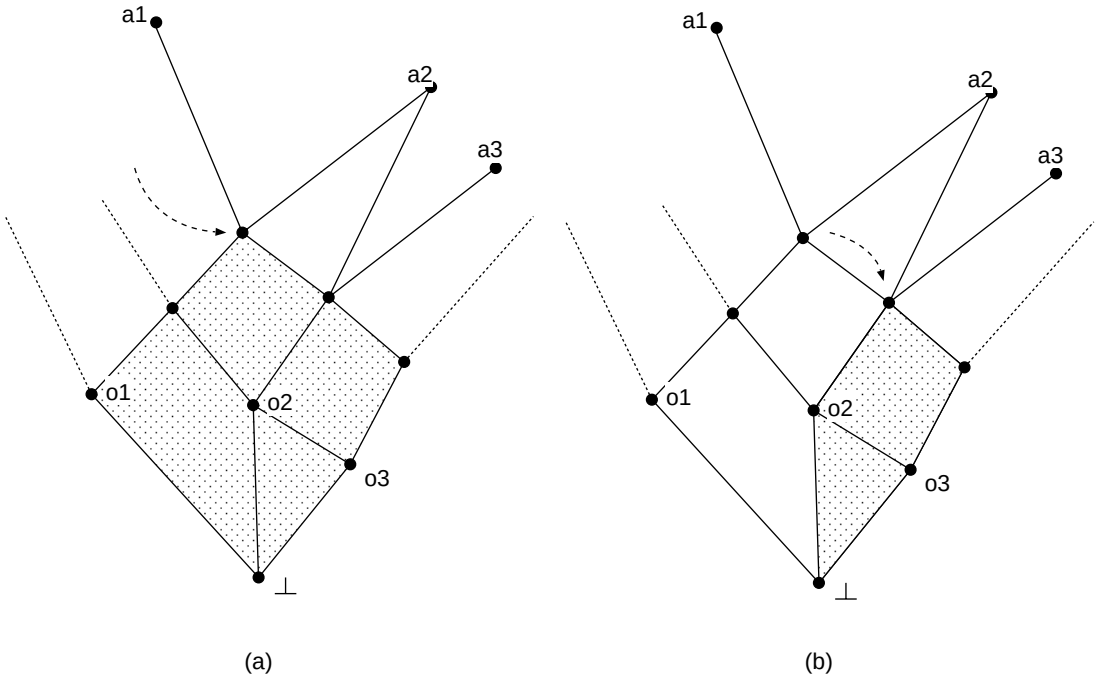

Abbildung 5.3: Eine Anfrage $\{a_1, a_2\}$ definiert einen Unterverband, der alle Objekte des Ergebnis enthält (a). Die Spezialisierung einer Anfrage durch die Hinzunahme von a_3 bestimmt ein neues Infimum, das einen kleineren Unterverband definiert (b).

Beweis. $(O_1, A_1) = \bigwedge_{a \in A} \mu(a) = (\bigcap \{a\}', (\bigcup \{a\})'') = (\bigcap \{a\}', A'')$. Mit Theorem 2 und 1 gilt $(\bigcap \{a\}', A'') = ((\bigcup \{a\})', A'') = (A', A'') = (\{o \mid o \in A'\}, A'') = (\{o \mid \{o\} \subseteq A'\}, A'') = (\{o \mid \{o\}' \supseteq A\}, A'') = (\llbracket A \rrbracket, A'')$.

Weil das Ergebnis jeder Anfrage durch einen Begriff beschrieben wird ist es gerechtfertigt, von einem *Ergebnisbegriff* zu sprechen – analog zu Objekt- und Attributbegriffen (vergleiche Theorem 4).

Das einleitende Beispiel im vorhergehenden Abschnitt 5.1 ist eine Folge von Anfragen, in der jede Anfrage spezieller als die vorhergehende ist. Die Anfragefolge spiegelt sich in dem Begriffsverband als ein Pfad von Ergebnisbegriffen wider. Tabelle 5.5 zeigt nochmals die beiden Anfragen aus Abschnitt 5.1 und in Abbildung 5.4 den Begriffsverband der Sammlung und darin hervorgehoben die Pfade der beiden Anfragen.

Beide Anfragen beginnen an dem größten Element $\top$ des Begriffsverbandes: er umfaßt als allgemeinster Begriff alle Komponenten. Von dort aus steigen die Pfade ab zu spezialisierten Unterbegriffen und enden unmittelbar über dem kleinsten Begriff $\bot$ des Verbandes. Dieser enthält keine Objekte und repräsentiert eine nicht erfüllbare Anfrage. Die im Beispiel vorgestellte Implementierung in Abbildung 5.2 bietet dem Anwender allerdings nur solche Attribute zur Auswahl an, die zu einem nicht leeren Ergebnis führen und damit den Ergebnisbegriff $\bot$ ausschließen.

Schritt	Schlüsselw.	Ergebnis	noch mögliche Schlüsselw.
1	–	alle	alle
2	*change*	`chdir chmod chown`	*directory file group mode owner permission*
3	*change file*	`chmod chown`	*group mode owner permission*
4	*change file mode*	`chmod`	–
1	–	alle	alle
2	*create*	`creat fork mkdir open`	*directory file new open process read write*
3	*create file*	`creat open`	*new read open write*
4	*create file read*	`open`	–

Tabelle 5.5: Zwei Beispiele für schrittweise Suche

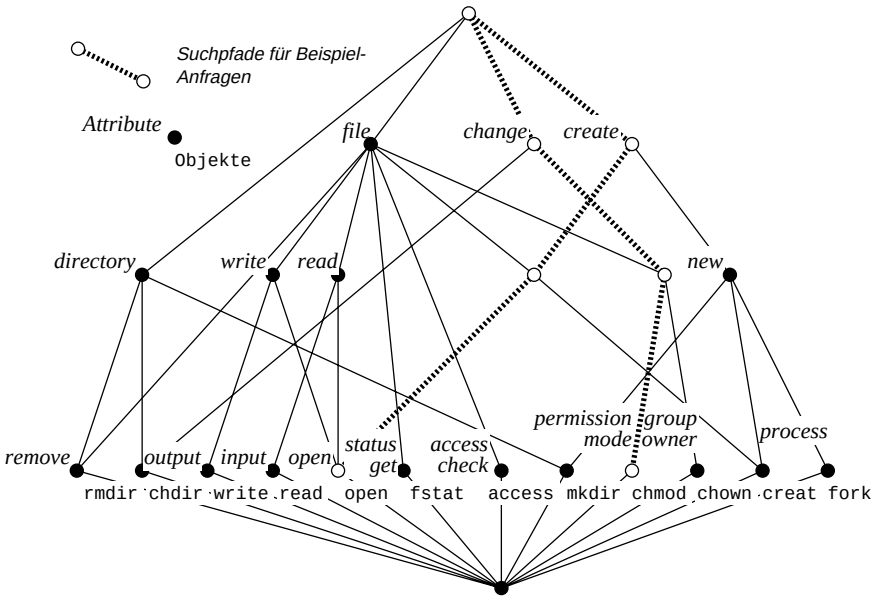

Abbildung 5.4: Begriffsverband für Unix-System-Calls mit hervorgehobenen Pfaden für die Anfragen aus Tabelle 5.5

Im allgemeinen Fall, also ohne diese Vorkehrungen, kann eine Anfrage zu einem beliebigen Ergebnisbegriff führen – auch dem kleinsten Element $\bot$.

Die Berechnung der Ergebnisbegriffe für die erste Anfrage in Tabelle 5.5 kann in Abbildung 5.4 leicht nachvollzogen werden: der Ausgangsbegriff ist $\top$, der nächste $\mu(\mathit{change}) \wedge \top = \mu(\mathit{change})$. Der dritte (unmarkierte) Ergebnisbegriff auf dem Pfad ist $\mu(\mathit{change}) \wedge \mu(\mathit{file})$ und der vierte ergibt sich analog.

5.3 Äquivalenz von Anfragen und Begriffen

Die Beispiele in Tabelle 5.5 und Abbildung 5.4 können den Eindruck erwecken, daß verschiedene Anfragen auch verschiedene Ergebnisse zur Folge haben, oder daß zumindest die Erweiterung einer Anfrage um ein Attribut zu einem neuen Ergebnisbegriff führt. Dies ist im Allgemeinen aber nicht der Fall: die Anfragen $\{directory, remove\}$ und $\{directory, file\}$ führen zu dem selben Ergebnis und Ergebnisbegriff $\gamma(rmdir)$. Die Anfrage $\{file, change\}$ ist eine Untermenge der Anfrage $\{file, change, access\}$. Trotzdem besitzen beide das gleiche Ergebnis, weil $\mu(change) \wedge \mu(access) = \mu(change)$ gilt.

Definition 17 *In einem Begriffsverband $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ heißen zwei Anfragen A_1, $A_2 \subseteq \mathcal{A}$ äquivalent genau dann, wenn sie das gleiche Ergebnis besitzen: $[\![A_1]\!] = [\![A_2]\!]$. Die zu $A \subseteq \mathcal{A}$ äquivalenten Anfragen werden mit $[A]$ bezeichnet: $[A_1] = \{A_2 \subseteq \mathcal{A} | [\![A_1]\!] = [\![A_2]\!]\}$.*

Da die Gleichheit von Mengen bekanntlich eine Äquivalenzrelation ist, führt die obige Definition eine Äquivalenzrelation auf Anfragen ein. Diese Definition hilft, die Beziehung zwischen Anfragen und dem Begriffsverband zu präzisieren. Theorem 12 besagt, daß zu jeder Anfrage ein Begriff existiert, der das Ergebnis der Anfrage als Objektmenge enthält. Tatsächlich besteht zwischen den Begriffen einer Sammlung und den Äquivalenzklassen der Anfragen eine eineindeutige Beziehung. Insbesondere existiert für jeden Begriff auch eine Anfrage, deren Ergebnis die Objektmenge des Begriffes ist.

Theorem 13 *Sei $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Begriffsverband. Die Abbildung φ zwischen den Äquivalenzklassen $\mathcal{A}_{/[\cdot]} = \{[\mathcal{A}] | A \subseteq \mathcal{A}\}$ und $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ist eine Bijektion.*

$$\begin{aligned} \varphi : \mathcal{A}_{/[\cdot]} &\rightarrow B(\mathcal{O}, \mathcal{A}, \mathcal{R}) \\ \varphi : [A] &\mapsto ([\![A]\!], A'') \end{aligned}$$

Beweis: φ ist (1) injektiv und (2) surjektiv. Seien $A_1, A_2 \subseteq \mathcal{A}$ und gelte $\varphi[A_1] = \varphi[A_2]$. Dann gilt $([\![A_1]\!], A_1'') = ([\![A_2]\!], A_2'') \Rightarrow [\![A_1]\!] = [\![A_2]\!] \Rightarrow [A_1] = [A_2]$; also ist φ injektiv. Sei $(O, A) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ und betrachte $\varphi[A] = ([\![A]\!], A'')$. Mit Theorem 12 und $A'' = (A')' = O' = A$ gilt: $\varphi[A] = (A', A'') = (A', A) = (O, A)$. Also ist φ surjektiv und damit bijektiv.

Der Begriffsverband einer Komponentensammlung repräsentiert alle möglichen Ergebnisse, die durch Anfragen an ihn entstehen. Anfragen mit dem gleichen Ergebnis werden unter einem Begriff zusammengefaßt und zu jedem Begriff existiert mindestens eine Anfrage. Der Begriffsverband einer Sammlung gestattet deswegen, die Anfragen und ihre Ergebnisse systematisch zu analysieren.

5.4 Berechnung des Ergebnisses

Das Ergebnis $[\![A]\!]$ einer Anfrage $A \subseteq \mathcal{A}$ in einem Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ist nach Theorem 12 die Menge A' von Komponenten. Sie wird durch den Schnitt der Attributspalten in der Kontexttabelle $A' = (\bigcup_{a \in A} \{a\})' = \bigcap_{a \in A} \{a\}'$ berechnet. Dazu ist eine vorherige Berechnung des Begriffsverbandes unnötig; der algorithmische Aufwand beträgt $O(|\mathcal{O}| \times |\mathcal{A}|)$.

Eine weitere Möglichkeit ist, das Ergebnis einer Anfrage A an dem zugehörigen Ergebnisbegriff $(O_1, A_1) = ([\![A]\!], A'') = (A', A'')$ abzulesen. Er ist das Infimum der Attributbegriffe $\mu(a_i)$ mit $a_i \in A$. Wenn die in Abschnitt 3.7 vorgeschlagenen Datenstrukturen und Algorithmen verwendet werden, bedeutet dieses: zunächst wird die Objektmenge des Infimums berechnet – dies ist wiederum A' und der zugehörige Aufwand ist $O(|\mathcal{O}| \times |\mathcal{A}|)$. Die Objektmenge A' wird dann als Schlüssel benutzt, um in einem geordneten Array aller Begriffe den Ergebnisbegriff durch binäre Suche zu ermitteln. Der zusätzliche Aufwand beträgt $O(|\mathcal{O}| \times \log n)$ mit $n = |B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$. Die Berechnung des Ergebnis einer Anfrage nur mit Hilfe des Kontextes ist also effizienter als mit Hilfe des Begriffsverbandes, da dabei zusätzlicher Aufwand durch eine binäre Suche entsteht.

Der Ergebnisbegriff enthält neben dem Ergebnis A' selbst die Attributmenge A''. Sie ist unter allen Anfragen mit A' als Ergebnis maximal, und deswegen kann A'' als kanonischer Repräsentant der Äquivalenzklasse $[A]$ betrachtet werden. Dieser kanonische Repräsentant wird für die Berechnung aller sinnvollen Verfeinerungsmöglichkeiten einer Anfrage benötigt (vergleiche Abschnitt 5.5) und ist wie diese selbst ein Beispiel für Informationen, die von dem Ergebnis einer Anfrage abhängen. Mit Hilfe des Begriffsverbandes, der jedes mögliche Ergebnis enthält, können diese Informationen vorberechnet und zusammen mit den Begriffen abgelegt werden. Im Falle von A'' ist sie sogar selbst ein Teil des Begriffes. Der Zugriff auf diese Informationen zum Zeitpunkt einer Anfrage ist weiterhin nur durch den Aufwand zur Ermittlung des Begriffes bestimmt: $O(|\mathcal{O}| \times |\mathcal{A}| + |\mathcal{O}| \times \log n)$. Eine Implementierung, die nur den Kontext zur Berechnung des Ergebnis verwendet, muß jede vom Ergebnis abhängende Information zum Zeitpunkt der Abfrage berechnen und verschlechtert damit ihre Effizienz, je mehr Informationen dieser Art bestimmt werden müssen.

Bereits bei der Verwendung des kanonischen Repräsentanten A'' von $[A]$ lohnt sich die Berechnung des Begriffsverbandes. Die allein auf dem Kontext basierenden Implementierung kann A'' mit einem Aufwand von $O(|\mathcal{O}| \times |\mathcal{A}|)$ bestimmen. Bei Komponentensammlungen mit moderat großen Begriffsverbänden ist die Implementierung der binären Suche mit $O(|\mathcal{O}| \times \log n)$ effizienter, da $\log n \leq |\mathcal{A}|$, weil $n \leq 2^{|\mathcal{A}|}$ gilt.

Der Effizienzvorsprung einer Bibliotheksorganisation über ihre Begriffe beruht darauf, daß jedes Ergebnis einer Anfrage in dem Begriffsverband enthalten ist. Damit kann jede vom Ergebnis einer Anfrage abhängende Information ebenfalls (vor-) berechnet werden. Die kanonischen Repräsentanten jeder Anfrageäquiva-

lenzklasse und ihre Verfeinerungsmöglichkeiten sind typische und sinnvolle Beispiele für diese Art von Information.

5.5 Anfragespezialisierung

Die Suche nach einer Softwarekomponente erfolgt in der Regel *nicht* durch die Formulierung einer präzisen Anfrage, deren Ergebnis dann die gesuchte Komponente ist. Bereits in dem Abschnitt 4.3.2 wurde erläutert, daß ein Benutzer das gesuchte Ziel häufig nur unpräzise und unvollständig formulieren kann und deswegen eine schrittweise Suche wünschenswert ist. Eine solche Suche wurde in dem Beispiel in Abschnitt 5.1 vorgeführt: ausgehend von der allgemeinsten Anfrage $A = \emptyset$ mit einem maximalen Ergebnis $[\![A]\!] = \mathcal{A}$ verfeinert ein Anwender seine Anfrage schrittweise und gelangt so (hoffentlich) zu dem gewünschten Ergebnis.

Eine Anfrage $A \subseteq \mathcal{A}$ wird durch die Hinzunahme eines Attributes $a \in \mathcal{A}$ verfeinert. Das neue Ergebnis ist $[\![A \cup \{a\}]\!]$; dabei ist nicht ausgeschlossen, daß das neue Ergebnis $[\![A \cup \{a\}]\!]$ nicht spezieller oder sogar leer ist, wenn das hinzugenommene Attribut im Widerspruch zur bisherigen Anfrage steht. Um diese Situation zu vermeiden, können in einer interaktiven Suche dem Benutzer nur *zur Verfeinerung geeignete Attribute* angeboten werden, die ein spezielleres und nicht leeres Ergebnis garantieren.

Definition 18 (Sinnvolle Attribute) *Sei $A \subseteq \mathcal{A}$ eine Anfrage für den Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$. Dann bezeichnet $\langle\langle A \rangle\rangle \subseteq \mathcal{A}$ die Menge der sinnvollen Attribute zur Spezialisierung von A:*

$$a \in \langle\langle A \rangle\rangle \iff \emptyset \subset [\![A \cup \{a\}]\!] \subset [\![A]\!]$$

Die für eine Verfeinerung geeigneten Attribute $\langle\langle A \rangle\rangle$ einer Anfrage A sind für alle Anfragen in der Äquivalenzklasse $[A]$ gleich. Da der Begriffsverband gerade die Ergebnisse für alle Äquivalenzklassen enthält (Theorem 13), können sie im Voraus berechnet werden.

Theorem 14 *Sei $A \subseteq \mathcal{A}$ eine Anfrage in einem Begriffsverband $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$. Dann gilt für die sinnvollen Attribute für die Verfeinerung der Anfrage A:*

$$a \in \langle\langle A \rangle\rangle \iff \exists o \in A' : (o, a) \in \mathcal{R} \wedge a \notin A''$$

Der Beweis zeigt die Äquivalenz der Prädikate aus Definition 18 und dem Theorem zur Charakterisierung von $\langle\langle A \rangle\rangle$:

1. Sei $a \notin A''$ und existiere $o \in A'$ mit $(o, a) \in \mathcal{R}$. Nach Theorem 1 gilt: $A'' \supseteq A$, also $a \notin A$ und damit $A \cup \{a\} \supset A$. Mit Theorem 1 und Theorem 12 gilt $(A \cup \{a\})' \subset A' \iff [\![A \cup \{a\}]\!] \subset [\![A]\!]$. Es bleibt zu zeigen, daß $[\![A \cup \{a\}]\!]$ nicht leer ist; betrachte o: $o \in \{\bar{o} \mid (\bar{o}, a)\} = \{a\}'$. Aus $o \in A'$ und $o \in \{a\}'$ folgt $o \in A' \cap \{a\}' = (A \cup \{a\})' = [\![A \cup \{a\}]\!]$.

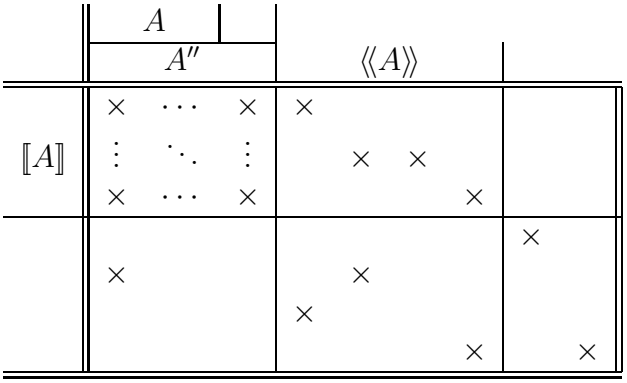

Abbildung 5.5: Schematischer Zusammenhang zwischen einer Anfrage A, ihrem Ergebnis $[\![A]\!]$ und ihrer Verfeinerung $\langle\langle A\rangle\rangle$ in der Kontexttabelle.

2. Gelte $\emptyset \subset [\![A \cup \{a\}]\!] \subset [\![A]\!]$ für beliebige $\{a\}, A \subseteq \mathcal{A}$. Wenn man annimmt, daß $a \in A'' \iff \{a\} \subseteq (A')' \iff A' \subseteq \{a\}'$ gilt, dann ergibt sich der folgende Widerspruch: $[\![A \cup \{a\}]\!] = (A \cup \{a\})' = A' \cap \{a\}' = A' = [\![A]\!]$. Also muß $a \notin A''$ gelten. Betrachte nun ein beliebiges $o \in [\![A \cup \{a\}]\!] \subset [\![A]\!] = A'$. Also existiert $o \in A'$. Weiter folgt aus $o \in [\![A \cup \{a\}]\!] \iff \{o\} \subseteq (A \cup \{a\})' \iff \{o\}' \supseteq A \cup \{a\}$, daß $a \in \{o\}'$ gilt, und damit $(o, a) \in R$.

Der Zusammenhang zwischen einer Anfrage A, ihrem Ergebnisbegriff (A', A'') und $\langle\langle A\rangle\rangle$ ist in Abbildung 5.5 skizziert. Für jede Anfrage läßt sich die Tabelle des zugehörigen Kontextes wie in Abbildung 5.5 sortieren. Die Attribute A einer Anfrage und ihr Ergebnis $[\![A]\!]$ bilden ein ausgefülltes Rechteck in der Kontexttabelle. Sowohl $[\![A]\!]$ als auch A'' können aus dem Ergebnisbegriff abgelesen werden (Theorem 12). Jedes zur Verfeinerung von A sinnvolle Attribut aus $\langle\langle A\rangle\rangle$ steht in Relation zu einem Objekt aus $[\![A]\!]$, ist aber nicht bereits in A'' enthalten. Attribute, die nicht zu einem Objekt $o \in [\![A]\!]$ in Relation stehen, sind auch nicht zur Verfeinerung von A geeignet.

Sowohl in Theorem 14, als auch in Abbildung 5.5 ist bei genauerer Betrachtung ein bislang ignoriertes Problem zu erkennen: ein zur Vereinfachung einer Anfrage A geeignetes Attribut muß in Relation zum bisherigen Ergebnis stehen und darf nicht in A'' enthalten sein. Da $A \subseteq A''$ (Theorem 1) gilt, existieren für bestimmte Anfragen Attribute $A'' \setminus A$, die nicht in der Anfrage enthalten sind, nicht im Widerspruch zu A stehen, und dennoch keine Verfeinerung des Ergebnis bewirken. Wenn ein solches Attribut $a \in A'' \setminus A, \exists o \in A' : (o, a) \in \mathcal{R}$ eine existierenden Anfrage A ergänzt, ist das Ergebnis unverändert $[\![A]\!] = [\![A \cup \{a\}]\!]$ und damit nicht die gewünschte Spezialisierung. Ein Attribut kann also aus zwei Gründen für die Spezialisierung einer Anfrage ungeeignet sein: weil es zu einem leeren Ergebnis führt, oder weil es zu einem unveränderten Ergebnis führt.

Attribute, deren Aufnahme in eine Anfrage zu einem leeren Ergebnis führen, stehen im Widerspruch zu der bisherigen Anfrage: keine Komponente existiert

mit der geforderten Kombination von Attributen. Attribute, deren Aufnahme zu einem unveränderten Ergebnis führen, werden bereits durch die aktuelle Anfrage impliziert. Sie entstehen zum Beispiel durch Begriffe, die gleichzeitig Attributbegriff von zwei Attributen sind: wenn $\mu(a_1) = \mu(a_2)$ gilt und $a_1 \in A$ Teil der aktuellen Anfrage ist, enthält jede Komponente in $[\![A]\!]$ auch a_2. Die Hinzunahme von a_2 bewirkt also keine Verkleinerung von $[\![A \cup \{a_2\}]\!]$.

5.5.1 Praktische Aspekte

Bei der Implementierung einer schrittweisen Spezialisierung durch die Auswahl von sinnvollen Attributen im Sinne der Definition 18 bietet sich die folgende Lösung an: dem Benutzer werden zu einer Anfrage A die Attribute $\langle\langle A \rangle\rangle$ zur Verfeinerung angeboten. Durch die Auswahl eines Attributes $a \in \langle\langle A \rangle\rangle$ entsteht eine neue Anfrage $A \cup \{a\}$, mit einem verkleinerten, aber nicht leeren Ergebnis. Die Präsentation der aktuellen Anfrage enthält nicht nur die unmittelbar durch den Benutzer ausgewählten Attribute $A \cup \{a\}$, sondern wird zusätzlich um die von der Anfrage implizierten (und möglicherweise besonders gekennzeichneten) Attribute $(A \cup \{a\})'' \setminus (A \cup \{a\})$ ergänzt. Die neue Anfrage enthält also insgesamt die Attribute $(A \cup \{a\})''$; dieses ist die maximale Anfrage mit dem selben Ergebnis wie $A \cup \{a\}$ und kann dem Ergebnisbegriff entnommen werden.

Würden die durch eine Anfrage implizierten Attribute nicht automatisch ergänzt, müßte entweder der Effekt eines unveränderten Ergebnis, oder eines plötzlichen Verschwindens von Attributen in Kauf genommen werden: Attribute, deren Auswahl zu einem unveränderten Ergebnis führen würden, würden dem Benutzer nicht mehr zur Auswahl angeboten.

5.5.2 Eigenschaften von $\langle\langle A \rangle\rangle$

Die Attribute zur Verfeinerung einer Anfrage A hängen von A ab, aber zwei verschiedene Anfragen A_1, A_2 bedingen nicht zwei verschiedene Mengen $\langle\langle A_1 \rangle\rangle, \langle\langle A_2 \rangle\rangle$ zu ihrer Verfeinerung. Entscheidend für $\langle\langle A \rangle\rangle$ ist die Menge A'', die dem Ergebnisbegriff entnommen werden kann. Da der Begriffsverband einer Sammlung genau die Ergebnisbegriffe enthält (Theoreme 12 und 13) können sämtliche Mengen zur Verfeinerung vorberechnet, und zusammen mit den Ergebnisbegriffen abgelegt werden. Unter diesem Aspekt ist es hilfreich, die Mengen zur Verfeinerung einer Anfrage als eine Eigenschaft der Begriffe einer Sammlung zu betrachten:

Definition 19 *Sei $B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Begriffsverband und $c = (O, A) \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Begriff. Zu c wird die Menge der zur Verfeinerung (der Anfragen $[A]$) sinnvollen Attribute definiert als $\langle\langle c \rangle\rangle = \langle\langle A \rangle\rangle$.*

Die obige Definition schafft eine neue Notation, die es erleichtert, den Zusammenhang zwischen den Operationen des Begriffsverbandes $\wedge$ und $\vee$ und $\langle\langle \cdot \rangle\rangle$

zu beobachten. Schon in dem einführenden Beispiel ist zu erkennen, daß mit der Verfeinerung einer Anfrage die Möglichkeiten für eine weitere Verfeinerung abnehmen. Durch Anwendung von Theorem 14 kann man diese Eigenschaft leicht zeigen:

Theorem 15 *Seien $c_1, c_2 \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ Begriffe. Dann gilt:*

1. $c_1 \leq c_2 \Rightarrow \langle\langle c_1 \rangle\rangle \subseteq \langle\langle c_2 \rangle\rangle$
2. $\langle\langle c_1 \wedge c_2 \rangle\rangle \subseteq \langle\langle c_1 \rangle\rangle \cap \langle\langle c_2 \rangle\rangle$
3. $\langle\langle c_1 \vee c_2 \rangle\rangle \supseteq \langle\langle c_1 \rangle\rangle \cup \langle\langle c_2 \rangle\rangle$

Für die praktische Vorberechnung aller Verfeinerungen mit Hilfe des Begriffsverbandes bedeutet dies, daß $\langle\langle c_1 \wedge c_2 \rangle\rangle$ nicht einfach aus $\langle\langle c_1 \rangle\rangle \cap \langle\langle c_2 \rangle\rangle$ bestimmt werden kann, da $\langle\langle c_1 \rangle\rangle \cap \langle\langle c_2 \rangle\rangle$ zu viele Elemente enthält: eine Auswahl eines dieser überzähligen Elemente würde zu einem unveränderten Ergebnis führen, aber niemals zu einem leeren Ergebnis. Durch einen Vergleich von $\langle\langle c_1 \wedge c_2 \rangle\rangle$ und $\langle\langle c_1 \rangle\rangle \cap \langle\langle c_2 \rangle\rangle$ ist dies leicht einzusehen; dabei ist $c_i = (O_i, A_i)$:

$$\begin{aligned} \langle\langle c_1 \wedge c_2 \rangle\rangle &= \{a \mid (o,a) \in \mathcal{R}, o \in O_1 \cap O_2, a \notin (A_1 \cup A_2)''\} \\ \langle\langle c_1 \rangle\rangle \cap \langle\langle c_2 \rangle\rangle &= \{a \mid (o,a) \in \mathcal{R}, o \in O_1 \cap O_2, a \notin (A_1 \cup A_2)\} \end{aligned}$$

Die durch die zweite Gleichung bestimmten Attribute stehen in Relation zu $O_1 \cap O_2$, dem Ergebnis, das durch $c_1 \wedge c_2$ bestimmt wird. Deshalb ist ausgeschlossen, daß durch die Wahl eines dieser Attribute ein leeres Ergebnis entsteht. Allerdings werden gegenüber der korrekter Verfeinerung in der oberen Gleichung nicht genügend Attribute ausgeschlossen, da $A_1 \cup A_2 \subseteq (A_1 \cup A_2)''$ gilt.

Wenn die Verfeinerungsmenge eines Begriffes mit Hilfe seiner Unterbegriffe bestimmt werden soll, ist dies ebenfalls nicht einfach durch die Vereinigung der entsprechenden Mengen möglich. In diesem Fall enthält die Vereinigung zu wenig Attribute, wodurch sogar Ergebnisse verlorengehen würden. Die Berechnung von $\langle\langle c \rangle\rangle$ kann also die Werte $\langle\langle c_i \rangle\rangle$ von Ober- und Unterbegriffen nicht ausnutzen und muß für jeden Begriff getrennt erfolgen.

5.6 Verfeinerung durch Auswahl relevanter Beispiele

Die im letzten Abschnitt vorgestellte Verfeinerung einer Anfrage basiert auf der Auswahl von Attributen, um die eine bestehende Anfrage ergänzt wird. Alternativ kann eine Anfrage durch die Markierung einer Teilmenge von Objekten im Ergebnis verfeinert werden. Der Benutzer markiert dazu im aktuellen Ergebnis

einige relevante Objekte; daraus wird eine Anfrage für diese, sowie ihnen ähnliche Objekte abgeleitet, die dann im nächsten Ergebnis enthalten sind. Die Auswahl relevanter Dokumente in einer Ergebnismenge zur Umformulierung einer Anfrage ist bei Information-Retrieval-Systemen als *Relevance-Feedback* bekannt [94].

Sei A_i die aktuelle Anfrage und $[\![A_i]\!]$ das zugehörige Ergebnis: eine Menge von Objekten. Eine Teilmenge O des Ergebnis mit $\emptyset \subset O \subseteq [\![A_i]\!]$ beschreibt Objekte, die ein Anwender als relevant beurteilt. Aus ihnen kann die nächste Anfrage abgeleitet werden:

$$A_{i+1} = \bigcap_{o \in O} \{o\}'$$

Die neue Anfrage selektiert nur Objekte, die mindestens die den Beispielen gemeinsamen Attribute aufweist. A_{i+1} läßt sich mit Hilfe von Theorem 2 wie folgt charakterisieren:

$$A_{i+1} = \bigcap_{o \in O} \{o\}' = (\bigcup_{o \in O} \{o\})' = O'$$

Die neue Anfrage A_{i+1} ist eine Obermenge der alten Anfrage A_i: $O \subseteq [\![A_i]\!] = A_i' \Leftrightarrow A_i \subseteq O' = A_{i+1}$. Sie selektiert deswegen eine kleinere Menge von Objekten: $A_i \subseteq A_{i+1} \Leftrightarrow A_i' \supseteq A_{i+1}' \Leftrightarrow [\![A_i]\!] \supseteq [\![A_{i+1}]\!]$.

Für eine nicht leere Menge von Beispielen ist auch das Ergebnis der verfeinerten Anfrage nicht leer, da mindestens die Beispiele O Teil des neuen Ergebnis sind. Dies ergibt sich mit Theorem 12: $[\![A_{i+1}]\!] = O'' \supseteq O \supset \emptyset$. Das Ergebnis der konstruierten Anfrage $[\![A_{i+1}]\!]$ kann wieder mit Hilfe des Begriffsverbandes bestimmt werden, wie im Abschnitt 5.4 erläutert.

Die Verfeinerung an Hand von Beispielen berechnet die kleinste (im Sinne des $''$-Operators) abgeschlossene Menge von Objekten, die eine vorgegebene Menge von Beispielen enthält. Sie weist ähnliche, aber nicht ganz so starke Eigenschaften wie die Verfeinerung durch Auswahl von Attributen aus der Menge $\langle\langle A_i \rangle\rangle$ von sinnvollen Attributen auf. Das neue Ergebnis ist niemals leer und es stellt eine mögliche Verfeinerung dar. Im Extremfall entspricht das neue Ergebnis dem alten – es tritt keine Verfeinerung ein. Dieser kann sowohl bei der Auswahl aller Objekte in die Beispielmenge ($O = [\![A_i]\!]$) auftreten, als auch bei der Auswahl sehr unterschiedlicher Beispiele. In diesem Fall existieren unter den Beispielobjekten außer A_i keine gemeinsamen Attribute.

Beide Methoden zur Verfeinerung einer Anfrage können gleichzeitig einem Benutzer angeboten werden, da sie sich ergänzen. Im ersten Fall wählt der Benutzer direkt gewünschte Attribute aus und indirekt die damit verknüpften Objekte. Im zweiten Fall wählt er relevante Objekte aus und damit indirekt Attribute, die zur Konstruktion der nächsten Anfrage verwendet werden. Dies ähnelt dem *Relevance-Feedback* bei Information-Retrieval-Systemen.

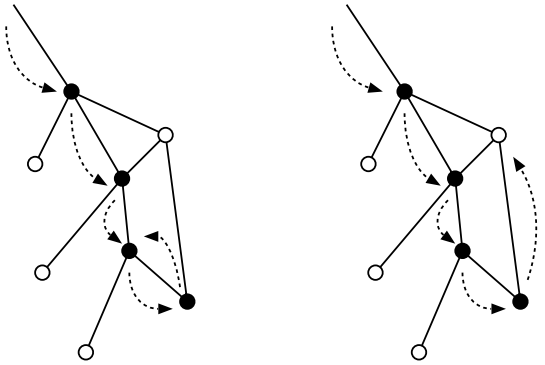

Abbildung 5.6: Das Entfernen von Attribute in einer beliebigen Reihenfolge aus einer Anfrage kann zu einer allgemeineren, aber unbekannten (∘) Anfragesituation führen (b).

5.7 Verallgemeinerung einer Anfrage

Im Verlauf einer inkrementell verfeinerten Anfrage wird jeder Zustand durch einen Begriff im Begriffsverband markiert. Durch eine Verfeinerung wird ein neuer Zustand, beziehungsweise Begriff, erreicht, der ein Unterbegriff des vorhergehenden ist. Dies ergibt sich daraus, daß der neue Ergebnisbegriff c_{i+1} nach Theorem 12 das Infimum (und damit ein Unterbegriff) des alten Ergebnisbegriffes c_i und der Attributbegriffe des neu hinzugenommenen Attributes a ist:

$$c_{i+1} = c_i \wedge \mu(a)$$

Im Verlauf einer Anfrage ergibt sich ein Pfad von besuchten Begriffen durch den Begriffsverband, der am größten Element beginnt. Für die Anfragen des einführenden Beispiels sind die beiden Pfade in Abbildung 5.4 eingezeichnet. Da ein Benutzer sich bei der Auswahl eines oder mehrerer zur Verfeinerung angebotener Attribute durchaus irren kann, entsteht der Wunsch, die getroffene Auswahl rückgängig zu machen. Wenn dies streng in der umgekehrten Reihenfolge der Auswahl geschieht, ist der nächste Zustand der vorherige. Eine Applikation kann die besuchten Begriffe in einer Liste speichern und damit leicht den vorhergehenden Zustand wiederherstellen.

Wird statt des letzten Attributes ein anderes oder werden mehrere Attribute aus A entfernt, entsteht eine unbekannte Situation. Der nächste Begriff ist zwar wieder ein Oberbegriff des aktuellen, aber nicht notwendig einer der zuvor besuchten Begriffe – siehe Abbildung 5.6.

Wenn die Attributmenge des aktuellen Begriffs (oder Zustands) c_4 die Attribute $\{a_1, a_2, a_3, a_4\}$ umfaßt, dann enthalten die vorherigen Zustände das Wis-

sen über die Konstellationen[2] $c_3 \equiv \{a_1, a_2, a_3\}$, $c_2 \equiv \{a_1, a_2\}$, $c_1 \equiv \{a_1\}$ und $c_0 \equiv \{\}$. Wird zum Beispiel das Attribut a_3 entfernt, entsteht eine Anfrage $A = \{a_1, a_2, a_4\}$, die durch keinen der vorherigen Zustände beschrieben wird.

Um den neuen Zustand c_{i+1} zu berechnen, kann auf den maximalen Zustand $c_i = (O_i, A_i)$ mit $A_i \subseteq A$ zurückgegriffen werden. Von dort ausgehend müssen die verbleibenden Attribute $A \setminus A_i$ eingefügt werden:

$$c_{i+1} = c_i \wedge \bigwedge_{a \in A \setminus A_i} \mu(a)$$

Im Falle des Beispiels ist dies $c_5 = c_2 \wedge \mu(a_4)$. Eine Applikation hat bei der Implementierung der Verallgemeinerung einer Anfrage also die Wahl, die Rücknahme nur des unmittelbar letzten Schrittes, oder eines beliebigen vorherigen Schrittes zu implementieren. Während ersteres zu einem auch dem Benutzer bekannten vorherigen Zustand führt, muß dieser im zweiten Fall berechnet werden. Dabei kann das Wissen vorheriger Zustände verwendet werden. Statt nur eines beliebigen Schrittes können auch gleich mehrere vorhergehende Schritte (durch Entfernung entsprechender Attribute aus A) auf die beschriebene Weise zurückgenommen werden.

5.8 Analyse von Komponentensammlungen

Der Wert einer Komponentensammlung wird nicht alleine durch die Qualität ihrer Komponenten bestimmt, sondern auch durch die Zugriffsmöglichkeiten auf diese Komponenten. Der Zugriff auf eine Komponente wird einerseits durch die Benutzungsoberfläche der Komponentensammlung vermittelt, und andererseits durch die Indexierung der Komponenten (vergleiche Abschnitt 4.3.2). Sie bestimmt die logische Gruppierung, aber auch Abgrenzung von Komponenten untereinander. Deshalb bestimmt nicht die Indexierung einer einzelnen Komponente ihre Wirksamkeit, sondern das Zusammenspiel der Indexierung aller Komponenten. Dieses Zusammenspiel kann bei den meisten Konzepten zur Komponentenablage nur durch Versuche oder Beobachtungen des laufenden Betriebes überprüft werden. Der Begriffsverband einer Komponentensammlung bietet eine globale Sicht auf die Komponenten und gestattet so, die Qualität einer Indexierung ohne Testfälle zu beurteilen.

Jeder Begriff (O, A) repräsentiert die Äquivalenzklasse $[A]$ von Anfragen; jede Anfrage aus diese Äquivalenzklasse besitzt das selbe Ergebnis O. Im Rahmen einer iterativen Suche kann eine Anfrage aus $[A]$ verfeinert werden, um so zu einem spezielleren Ergebnis zu gelangen. Die neue Anfrage wird wieder durch einen

[2] unter der Annahme, daß bei der Auswahl eines Attribut nicht zusätzliche Attribute mit aufgenommen werden mußten wie im Abschnitt 5.5 erläutert; die Erklärungen hier bleiben aber prinzipiell gültig

	System-aufrufe	Bibliotheks-funktionen
Komponenten ($\|\mathcal{O}\|$)	223	407
Schlüsselwörter ($\|\mathcal{A}\|$)	256	350
Relationengröße ($\|\mathcal{R}\|$)	833	1715
Rel. Füllgrad der Kontext-Tabelle	0.0146	0.0120
Eindeutige Indexierungen	148	194
Begriffe	278	349
erwartete Begriffe nach Tab. 6.3	484	935

Tabelle 5.6: Eckdaten für zwei experimentelle Komponentenbibliotheken

Begriff repräsentiert, der ein Unterbegriff von (O, A) ist. Die Halbordnung des Begriffsverbandes ordnet auch die Anfragesituationen, weil zwischen Anfragen und Begriffen eine Bijektion besteht (vergleiche Abschnitt 5.3). Dies bedeutet, daß der Begriffsverband die Anfrage/Ergebnissituationen einer Komponentensammlung vollständig wiedergibt. Statt (unvollständige) Versuche mit eine Komponentensammlung durchzuführen ist es also möglich, ihren Begriffsverband (vollständig) zu analysieren.

Die Aufgabe einer Indexierung ist es, Komponenten sinnvoll zu gruppieren, aber auch zu unterscheiden. Ein Hinweis darauf, wie gut eine Indexierung dieses leistet, ist die Größe von Ergebnis- und Verfeinerungsmengen. Sollte sich zum Beispiel herausstellen, daß eine Anfrage ein Ergebnis mit über 20 Komponenten liefert und gleichzeitig keine Verfeinerungsmöglichkeiten mehr bietet, wäre dies auf eine ungenügend feine Indexierung der betroffenen Komponenten zurückzuführen.

5.8.1 Zwei Beispielbibliotheken

Zur Demonstration einiger Analysen wurden zwei Komponentensammlungen aus Betriebssystem- und Bibliotheksaufrufen des Betriebssystems Unix und der Programmiersprache C aufgebaut. Dazu wurde die Online-Dokumentation der Funktionen durch Schlüsselwörter indexiert, die automatisch der Dokumentation entnommen wurden. Konkret bildeten die Wörter der einzeilige Beschreibung der Funktionen aus den Sektionen 2 (System Calls) und 3 (Library Functions) des Manuals eines Linux-2.0-Systems die Attribute, die den Objekten zugeordnet wurden. Zuvor wurden aus den Attributmengen häufig auftretende Konjunktionen und Artikel wie *of*, *and* und *the* mit Hilfe einer Stop-Wörterliste entfernt. Dadurch entstanden zwei Komponentensammlungen mit 223 System- und 407 Bibliotheksaufrufen, deren Kennzahlen in Tabelle 5.6 zusammengefaßt sind.

Sowohl die Unix-Betriebssystemschnittstelle, als auch die C-Bibliothek enthält Gruppen von Funktionen mit ähnlichen Aufgaben. Ein Beispiel sind die Funktionen *execl*, *execlp*, *execle*, *exect*, *execv* und *execvp* der C-Bibliothek, die alle einen

neuen Prozeß starten. In Gruppen ähnlicher Funktionen reicht die einzeilige Beschreibung oft nicht aus, sie eindeutig zu unterscheiden. Als Folge davon existieren in beiden Komponentensammlungen deutlich weniger eindeutige Indexierungen als Komponenten. Bei 223 Systemaufrufen und 148 eindeutigen Indexierungen können zum Beispiel maximal 147 ($147 \times 1 + 1 \times 76 = 223$) und minimal 73 ($73 \times 1 + 75 \times 2 = 223$) eindeutig indexierte Komponenten existieren. Dies ist eine Folge der automatischen und sehr einfachen Indexierung und könnte durch größeren technischen oder manuellen Aufwand vermieden werden.

Die Begriffsverbänden der beiden Komponentensammlungen enthalten 278 und 349 Elemente. Dies sind relativ wenig im Vergleich zu zufällig aufgebauten Kontexten mit vergleichbaren Kontexttabellen. Das nachfolgende Kapitel 6 geht genauer auf die zu erwartende Größe von Begriffsverbänden ein. Als ein gutes Maß zu ihrer Vorhersage hat sich dabei die Relationengröße $|\mathcal{R}|$ und der relative Füllgrad von Kontexttabellen erwiesen. Er ist der Quotient aus $|\mathcal{R}|$ und $|\mathcal{O}| \times |\mathcal{A}|$ und beschreibt anschaulich den mit Kreuzen ausgefüllten Anteil einer Kontexttabelle. Zufällig generierte Kontexte mit den gleichen Maßzahlen wie die beiden Komponentensammlungen haben nach den in Kapitel 6 beschriebenen Versuchen Begriffsverbände mit 484 und 935 Elementen erwarten lassen. Die vergleichsweise geringe Anzahl von Begriffen in beiden Sammlungen ist vermutlich eine Folge der im Sinne der Indexierung zahlreichen Duplikate.

5.8.2 Verteilung von $[\![A]\!]$ und $\langle\langle A \rangle\rangle$

Die beiden einfachsten Möglichkeiten für die Analyse einer Komponentensammlung bestehen darin, die empirische Verteilung ihrer Ergebnis- und Verfeinerungsmengen zu betrachten. Nach Theorem 12 ist die Objektmenge O eines Begriffes (O, A) das Ergebnis aller Anfragen aus der Äquivalenzklasse $[A]$. Die empirische Verteilung der Ergebnisgrößen stellt die Häufigkeit jeder auftretenden Ergebnisgröße $|O|$ im Begriffsverband fest. Dabei wird allerdings der kleinste Begriff $\perp$ des Verbandes unberücksichtigt gelassen, da die durch ihn repräsentierte Anfragesituation mit den in den Abschnitten 5.1 und 5.5 vorgestellten Verfahren nicht erreicht werden kann: dem Anwender werden nur Attribute zur Verfeinerung seiner Anfragen angeboten, die ein leeres Ergebnis, und damit den Ergebnisbegriff $\perp$, vermeiden.

Die empirische Verteilung der Ergebnisgrößen für die beiden Komponentenbibliotheken zeigt Abbildung 5.7. Zu jeder Ergebnisgröße x auf der logarithmischen horizontalen Achse zeigt die Abbildung den prozentualen Anteil der Begriffe (Anfragesituationen), mit $|O| = |[\![A]\!]| \leq x$; diese akkumulierende Darstellung ist eine streng monoton ansteigende Funktion in Abhängigkeit der Ergebnisgröße. Zusätzlich ist die Häufigkeit jede Ergebnisgröße, ihre Dichte, mit einer Achse am rechten Rand der Abbildung eingetragen. Die Dichte der Ergebnisgröße ist der Ausgangspunkt für die Berechnung der Verteilung und nicht monoton, aber anschaulicher.

Die Statistik berücksichtigt Äquivalenzklassen von Anfragen, und nicht ein-

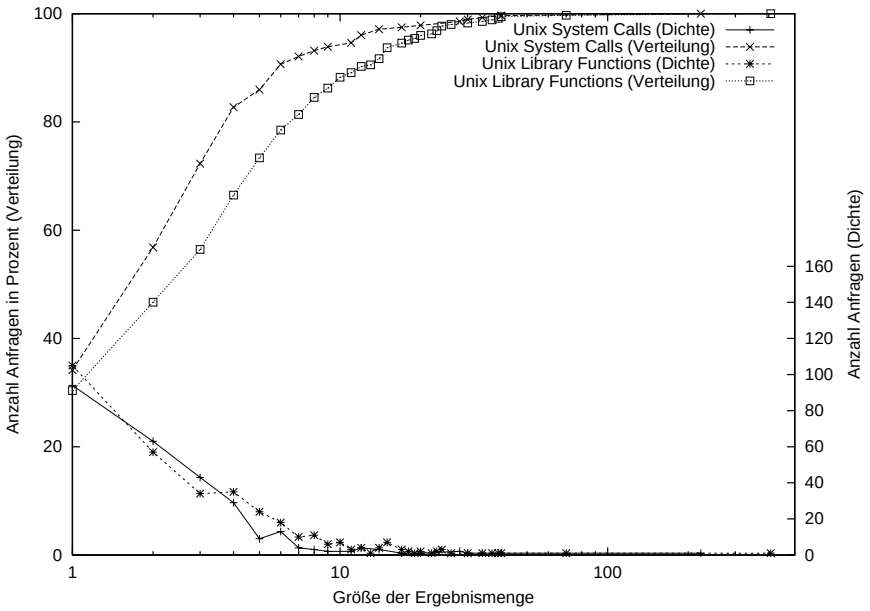

Abbildung 5.7: Verteilung von Anfragen über die Größe von Ergebnismengen $|[\![A]\!]|$ von zwei Komponentenbibliotheken

zelne Anfragen, da jede Klasse einer markanten Anfragesituation entspricht. Für eine genauere Analyse müßte die unterschiedliche Mächtigkeit $|[A]|$ der einzelnen Äquivalenzklassen $\{[A]|A \subseteq \mathcal{A}\}$ berücksichtigt werden. Es ist allerdings nicht offensichtlich, wie diese effizient bestimmt werden können.

In beiden Bibliotheken besitzen mehr als 50% aller Anfrageäquivalenzklassen Ergebnismengen mit weniger als 3 Elementen, und 80% aller Anfrageklassen Ergebnismengen mit weniger als 10 Elementen. Damit sind in mindestens 80% aller Anfragesituationen die Ergebnismengen gut überschaubar, was auf eine im Allgemeinen annehmbar feine Indexierung hinweist.

Allerdings sind weniger als 35% Anfragen mit einelementigem Ergebnis. Diese kennzeichnen Objekte, die wegen ihrer eindeutigen Indexierung gezielt gesucht werden können. Umgerechnet auf den Anteil der eindeutig indexierten Komponenten bedeutet dies, daß unter den Systemaufrufen 42% ($34\% \times 271 = 94 = 42\% \times 223$) und unter der Funktionsaufrufen der C-Bibliothek sogar nur 26% ($30\% \times 349 = 105 = 26\% \times 407$) aller Komponenten eindeutig indexiert sind. Obwohl die bisherige Indexierung überwiegend zu kleinen Ergebnismengen führt, kann eine eindeutigere Indexierung die gezielte Suche nach einzelnen Komponenten noch verbessern.

Bei circa 5% allen Anfragen entstehen große und sehr große Ergebnisse. Wegen der globalen Analyse ist allerdings nicht zu erkennen, ob diese Ergebnisse auch zu entsprechend allgemeinen Anfragen gehören, oder zu speziellen. Im letzteren Fall wäre dies ein Hinweis, daß einzelne Indexierungen verfeinert werden müßten.

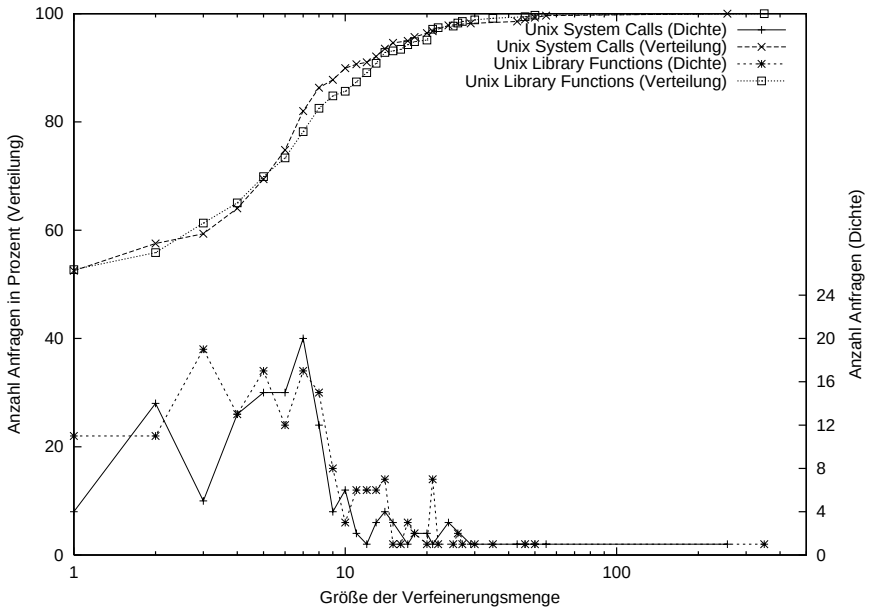

Abbildung 5.8: Verteilung von Anfragen über die Größe $|\langle\langle A \rangle\rangle|$ von Verfeinerungsmengen von zwei Komponentenbibliotheken

Abbildung 5.8 zeigt die Größe der Verfeinerungsmengen als empirische Verteilung (linke Achse) und Dichte (rechte Achse). Wegen der logarithmischen Darstellung auf der horizontalen Achse können die Werte für leere Mengen (*System Calls*: $142 \equiv 51\%$, *Library Functions*: $173 \equiv 50\%$) nicht dargestellt werden. Sie sind in der akkumulierenden Darstellung der Verteilung enthalten und Verzerren deshalb nur die Darstellung der Dichte.

Die Verteilung der Größe der Verfeinerungsmengen bietet ein ähnliches Bild wie die Ergebnisgrößen: in mehr als 85% aller Anfragesituationen stehen weniger als 10 Attribute zur weiteren Verfeinerung zur Verfügung. Und in circa 50% aller Situationen ist eine weitere Verfeinerung überhaupt nicht möglich. Die Indexierung der Komponenten erscheint damit ausreichend fein, aber nicht unübersichtlich, zu sein. Eine Verbesserung der Indexierung sollte zunächst der Anteil der eindeutig indexierten Komponenten erhöhen.

5.8.3 Berücksichtigung der Anfragehalbordnung

Die globale Analyse des vorhergehenden Abschnitts berücksichtigt nicht, daß Anfragen geordnet sind: von der allgemeinsten Anfrage ausgehend entstehen spezialisierte Anfragen, die selbst weiter spezialisiert werden können. Durch die gleichzeitige Betrachtung aller Begriffe des Verbandes wird der Unterschied zwischen allgemeinen und speziellen Anfragen verwischt. So sind Anfragen mit einer großen Ergebnismenge nur dann kritisch, wenn sie bereits stark spezialisiert sind. Da die

Halbordnung $\leq$ der Begriffe gleichzeitig die Spezialisierung von Anfragen ordnet, können allgemeine und spezielle Begriffe bei der Analyse unterschieden werden.

Eine Folge von einer allgemeinen zu einer spezialisierten Anfrage beginnt am größten Element $\top$ des Verbandes und führt zu einem Begriff c, dessen Verfeinerungsmenge $\langle\langle c \rangle\rangle$ leer ist. Alle möglichen Anfragefolgen können durch eine Tiefensuche [11] durch das Hasse-Diagramm des Begriffsverbandes bestimmt werden – zumindest theoretisch. Zu einer Kontexttabelle der Größe $n \times n$ existiert im ungünstigsten Fall ein Begriffsverband mit 2^n Begriffen. Sein Hasse-Diagramm besitzt die Form eines n-dimensionalen Würfels, der $2^{n-1} \times n$ Kanten und $n!$ Pfade vom größten zum kleinsten Element besitzt. Im Falle der Suche sind sogar noch mehr Pfade möglich, da innerhalb der so gezählten Pfade einzelne Begriffe übersprungen werden können. Diese Komplexität läßt den Versuch, alle Suchpfade auch für weniger extreme Kontexte zu bestimmen, hoffnungslos erscheinen.

Ein Grund für die große Anzahl von möglichen Suchpfaden in einem Verband ist, daß jeder Begriff, beziehungsweise jede Anfragesituation, mehrfach betrachtet wird. Ein Begriff c kann an mehreren Pfaden teilnehmen und in einem Pfad als zweiter, in einem anderen an dritter Stelle auftreten: $\top < c_1^1 < c$ und $\top < c_1^2 < c_2^2 < c_3^2 < c$. Aus diesem Grund kann einem Begriff keine globale gültige Position in allen Suchpfaden zugeordnet werden, die eine mehrfache Betrachtung vermeiden würde.

Statt alle Pfade eines Verbandes zu betrachten, können vereinfachend seine *Ebenen* analysiert werden. Auch wenn einem Begriff keine eindeutige Position auf einem Pfad zugeordnet werden kann, so existiert eine *minimale* Anzahl von Suchschritten, bevor eine gegebene Anfragesituation erreicht werden kann. Diese minimale Anzahl von Suchschritten definiert die *Ebene* eines Begriffes:

Definition 20 (Ebene eines Begriffes) *Sei $c \in B(\mathcal{O}, \mathcal{A}, \mathcal{R})$ ein Begriff; dann ist die* Ebene $l(c)$ *von wie folgt definiert:*

$$l(c) = \begin{cases} 0 & \textit{wenn } c = \top \\ 1 + \min\{l(i) \mid i \in \textit{super}(c)\} & \textit{sonst} \end{cases}$$

Dabei ist $\textit{super}(c)$ die Menge aller direkten Oberbegriffe von c: $\textit{super}(c) = \{i \in B(\mathcal{O}, \mathcal{A}, \mathcal{R}) \mid i > c, \forall j \in \textit{super}(c) : i \not< j\}$.

Die Ebene $l(c)$ eines Begriffes gibt die minimale Entfernung von dem größten Element $\top$ des Verbandes an; dabei gilt $l(\top) = 0$. Ein Begriff c mit $l(c) = n$ tritt in mindestens einem Pfad als Suchschritt n auf, kann in anderen Pfaden aber auch in Schritten $> n$ auftreten. Bei einer vereinfachten Analyse der Komponentensammlung werden die Begriffe einer Komponentensammlung in Ebenen eingeteilt und jeweils die Begriffe einer Ebene gemeinsam analysiert. Zur Interpretation wird vereinfachend angenommen, daß alle Begriffe auf Ebene n als Schritt n einer Suchfolge auftreten. Algorithmisch wird die Ebene jedes Begriffe durch eine Breitensuche [11] in dem Hasse-Diagramm des Verbandes ermittelt. Die Komplexität

	Ebene	n	$[\![\bar{A}]\!]$	$\sigma[\![A]\!]$	$\langle\!\langle \bar{A} \rangle\!\rangle$	$\sigma\langle\!\langle A \rangle\!\rangle$
a	0	1	223.0	0.0	256.0	0.0
	1	79	6.7	66.2	11.5	91.4
	2	142	2.7	27.3	1.8	35.0
	3	54	1.9	10.5	0.3	7.5
	4	2	1.0	0.0	0.0	0.0
b	0	1	407.0	0.0	350.0	0.0
	1	95	10.3	100.3	11.7	86.2
	2	181	3.6	65.4	2.0	44.5
	3	63	2.4	18.8	0.8	12.2
	4	9	1.1	0.9	0.0	0.0

Tabelle 5.7: Analyse einer Komponentensammlung von 223 Unix-System-Calls (a) und 407 Unix-Library-Funktionen (b). Ein Begriff befindet sich auf Ebene l, wenn er nicht durch weniger als l Suchschritte erreicht werden kann; n bezeichnet die Anzahl der Begriffe auf Ebene l. Für jede Ebene wurde die durchschnittliche Größe der Ergebnis- und Verfeinerungsmengen bestimmt, sowie ihre Standardabweichungen.

einer Breitensuche in einem Graphen G mit Knotenmenge V und Kantenmenge E ist $O(|K| + |V|)$; im Falle des oben angesprochenen n-dimensionalen Würfels ist dies zum Vergleich $O(2^n + 2^{n-1+\ln n}) \leq O(2^{n+\ln n})$.

Für die beiden experimentellen Komponentenbibliotheken wurden die Anfrageäquivalenzklassen der verschiedenen Ebenen analysiert[3]. Tabelle 5.7 zeigt für jede Ebene die Zahl n der Anfragen, die durchschnittliche Größe des Ergebnis $[\![\bar{A}]\!]$, seine Standardabweichung $\sigma[\![A]\!]$ sowie die durchschnittliche Größe der Verfeinerungsmengen $\langle\!\langle A \rangle\!\rangle$ mit der zugehörigen Standardabweichung. Insgesamt sind deutlich die im Verlauf einer Anfrage abnehmenden Größen der Ergebnis- und Verfeinerungsmengen zu erkennen. Trotz der Einteilung der Anfragen in Ebenen ist eine hohe Standardabweichung bei den einzelnen Parametern immer noch auffällig. Sie deuten auf möglicherweise problematische Anfragesituationen hin, in denen große Ergebnis- und Verfeinerungsmengen existieren.

Einen genaueren Eindruck vermitteln die empirischen Verteilungen der Anfragesituationen über die Größe von Ergebnissen in den Abbildungen 5.9 und 5.10. Sie unterscheiden die Ebenen der Begriffe und verwenden die absolute Anzahl von Begriffe, statt ihres prozentualen Anteils wie in Abbildung 5.7. Wie in den Abbildungen 5.7 und 5.8 sind die Häufigkeiten der verschiedenen Ergebnisgrößen zur Orientierung ebenfalls eingezeichnet; die zugehörige Achse liegt am rechten Rand der Abbildung.

- Die Verteilung der Ergebnisgrößen ist bei Betriebssystemaufrufen und Bi-

[3]Die Laufzeit dieser Analysen betrug wenige Sekunden auf einem 166 MHz Pentium unter Linux 2.0. Dabei wurde eine Implementierung der Begriffsanalyse in Objective Caml verwendet.

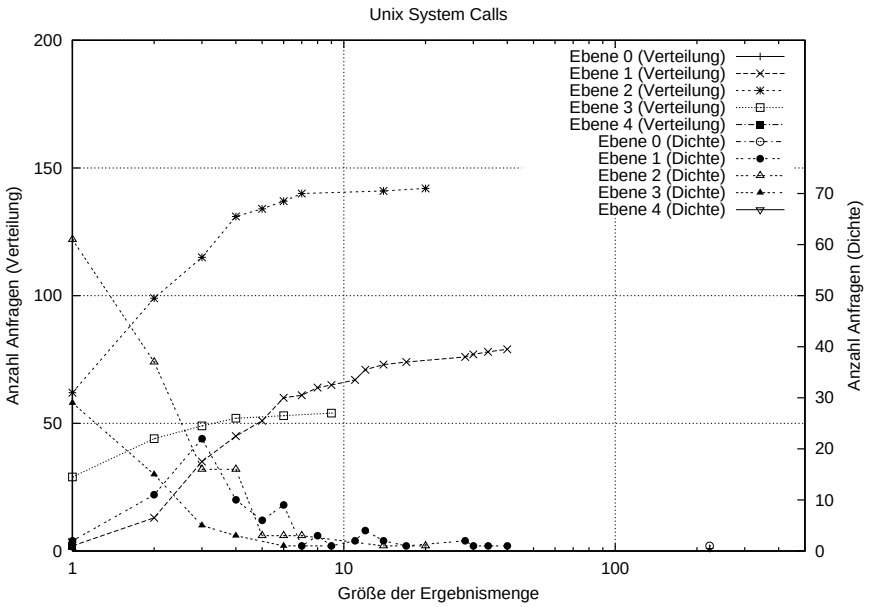

Abbildung 5.9: Verteilung von Anfragen über die Größe $|[\![A]\!]|$ von Ergebnismengen auf verschiedenen Ebenen bei Betriebssystemaufrufen.

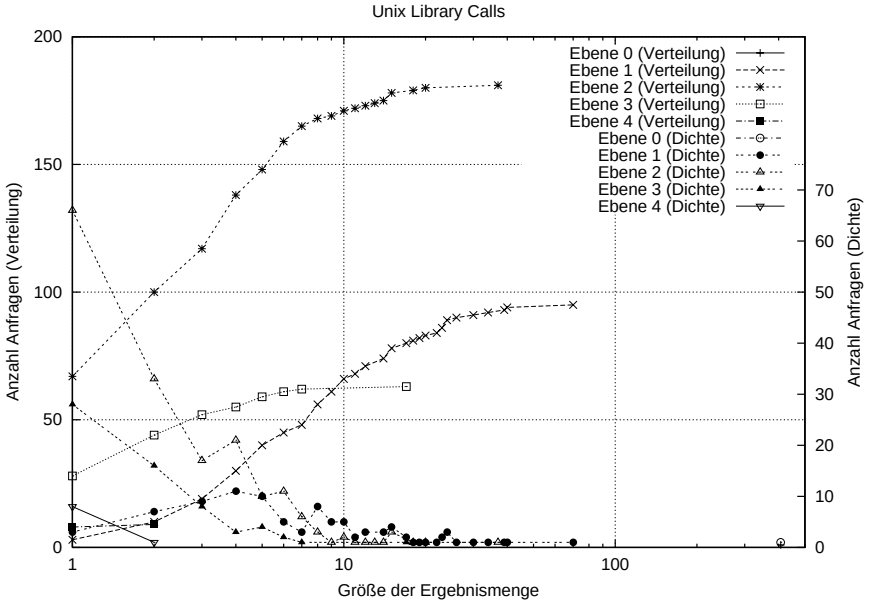

Abbildung 5.10: Verteilung von Anfragen über die Größe $|[\![A]\!]|$ von Ergebnismengen auf verschiedenen Ebenen bei Bibliotheksaufrufen der C-Library.

bliotheksfuntionen prinzipiell ähnlich: die meisten Anfragen liegen auf Ebene 2, die zusammen mit Ebene 1 die größte Bandbreite an Ergebnisgrößen aufweist. Nach spätestens 4 Suchschritten ist eine nicht verfeinerbare An-

frage erreicht.

- Die Verteilungen aller Ebenen sind zu ihrem rechten Ende hin stark abgeflacht. Dies bedeutet, daß die großen Ergebnismengen auf jeder Ebene nur einen kleinen Anteil ausmachen.

- Die wenigen Begriffe auf den Ebenen 1 und 2 mit mehr als ca. 25 Objekten können durch eine feinere Indexierung entfernt werden.

Dieser Abschnitt kann nur einen ersten Eindruck geben, welche Möglichkeiten sich zur Analyse durch die Äquivalenz von Anfragesituationen und Begriffen eröffnen. Ohne die Erfahrung aus einer in einem Produktionsumfeld genutzten Komponentenbibliothek sind die Merkmale einer guten Indexierung kaum abzuschätzen. Die Analyse von Ergebnisgrößen ist sicherlich nur ein erster Schritt in Richtung von Metriken, die helfen, die Indexierung von Komponenten schon bei ihrer Eingabe in ein System zu optimieren.

Kapitel 6

Größe von Begriffsverbänden

Der Begriffsverband einer $n \times n$ großen Kontexttabelle kann bis zu 2^n Elemente enthalten, wie bereits in Abschnitt 3.1 gezeigt wurde. Wegen dieses, zumindest in Einzelfällen, exponentiellen Zusammenhangs sind genaue Aussagen über die Größe von Begriffsverbänden für praktische Applikationen unerläßlich. Nur wenn exponentiell große Begriffsverbände für die geplante Anwendung eine pathologische Ausnahme bleiben, ist formale Begriffsanalyse überhaupt einsetzbar. Neben der Größe der Begriffsverbände als Maß für die Speicherkomplexität ist der Zeitaufwand zur Berechnung der Begriffe und ihrer Verbandsstruktur von praktischem Interesse.

Die theoretische Behandlung der Größe von Begriffsverbänden hat sich als schwierig erwiesen. Eines der wenigen Ergebnisse ist die mathematische Diplomarbeit von Schütt, in der obere Schranken für $|B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ angegeben werden. Das zentrale Theorem dieser Arbeit gibt die folgende obere Schranke für die maximale Größe beliebiger Begriffsverbände an:

Theorem 16 *Sei* $\alpha = \frac{3}{2}2^{\sqrt{3}} - 4$*, also* $\alpha \approx 0.98$*, dann gilt für alle Kontexte* $|B(\mathcal{O}, \mathcal{A}, \mathcal{R})| \leq \frac{3}{2}2^{\sqrt{|\mathcal{R}|+1}} - \alpha$.

Obwohl die durch das Theorem angegebene Schranke tatsächlich durch Begriffsverbände erreicht wird, bilden diese eine Ausnahme. Die Größe der bei praktischen Anwendungen entstehenden Verbände liegt typischerweise weit unter der durch Theorem 16 beschriebenen Obergrenze. Für die meisten Anwendungen ist deswegen auch nicht diese obere Schranke, sondern ein Erwartungswert für die Größe der entstehenden Begriffsverbände wichtiger. Dazu muß ein Kontext zunächst durch möglichst einfache Kennzahlen charakterisiert werden, aus denen dann die zu erwartende Größe abgeleitet werden kann. Da praktisch keine Aussagen über die zu erwartende Größe von Begriffsverbänden existieren, wurde eine Versuchsreihe durchgeführt; sie soll

- helfen, Kennzahlen zur Charakterisierung von Kontexten zu finden,

- einen funktionalen Zusammenhang zwischen einem Kontext und seiner zu erwartenden Größe nachweisen,
- und Aussagen über den Aufwand zur Berechnung von Begriffen und ihrer Verbandsstruktur ermöglichen.

In der Versuchsreihe wurden zufällige Kontexte und verschiedene Maßzahlen von ihnen festgehalten. Anschließend wurden die zugehörigen Begriffe und ihre Verbandsstruktur bestimmt, und nach Korrelationen zwischen den Maßzahlen und der Größe der entstandenen Verbände gesucht.

6.1 Versuchsaufbau und -ablauf

In jedem Versuch der Versuchsreihe wurde zufällig ein Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ konstruiert, der bis zu 200 Objekte und Attribute aufwies: $1 \leq |\mathcal{O}| \leq 200$ und $1 \leq |\mathcal{A}| \leq 200$. Jedes Objekt und Attribut stand mit mindestens einem Attribut, beziehungsweise Objekt, in Beziehung. Damit enthielt die Kontexttabelle der Größe $|\mathcal{O}| \times |\mathcal{A}|$ keine leeren Zeilen oder Spalten. Alle generierten Kontexte erfüllten zusätzlich die Bedingung $|\mathcal{R}| \leq 1/10 \times |\mathcal{O}| \times |\mathcal{A}|$: maximal 10% aller Plätze in der zugehörigen Kontexttabelle waren besetzt. Diese Bedingung schloß sehr kleine Kontexte aus, da diese zwangsläufig einen größeren Anteil ihrer Kontexttabelle belegen müssen, wenn jedes Objekt und jedes Attribut mit mindestens einem anderen Attribut oder Objekt in Relation steht. Für jeden generierten Kontext wurden die Parameter in Tabelle 6.1 festgehalten.

Symbol	Beschreibung
$\lvert\mathcal{O}\rvert$	Größe der Objektmenge
$\lvert\mathcal{A}\rvert$	Größe der Attributmenge
ratio	$\min(\lvert\mathcal{O}\rvert/\lvert\mathcal{A}\rvert, \lvert\mathcal{A}\rvert/\lvert\mathcal{O}\rvert)$, Seitenverhältnis Kontexttabelle
$\lvert\mathcal{R}\rvert$	Größe der Relation
fill	$\lvert\mathcal{R}\rvert/(\lvert\mathcal{O}\rvert \times \lvert\mathcal{A}\rvert)$, Belegungsgrad der Kontexttabelle
$\bar{o}$	mittlere Anzahl von Attributen pro Objekt
$\bar{a}$	mittlere Anzahl von Objekten pro Attribut

Tabelle 6.1: Während des Versuchs für Kontexte erfaßte Parameter

Die Beschränkung auf Relationen mit maximal 1/10 der Größe der Kontexttabelle stammt aus der Annahme, daß sie für die Verwendung bei Softwarekomponentenbibliotheken charakteristisch ist. Andernfalls müßte jede Softwarekomponente im Durchschnitt mehr als 10% aller zur Verfügung stehenden Attribute aufweisen. In Abschnitt 5.8 wurden zwei Bibliotheken untersucht, deren Belegungsgrad sogar kleiner als 0.02 (siehe Tabelle 5.6, Seite 95) war; sie stützen damit die getroffene Annahme. Natürlich kann sie nur durch die Untersuchung

einer Vielzahl von realen Anwendungen a posteriori gerechtfertigt oder widerlegt werden.

Symbol	Beschreibung
n	$\lvert B(\mathcal{O}, \mathcal{A}, \mathcal{R})\rvert$, Anzahl der Begriffe im Begriffsverband
t_1	CPU-Sekunden zur Berechnung aller Begriffe
t_2	CPU-Sekunden zur Berechnung der Verbandsstruktur

Tabelle 6.2: Erfaßte Parameter bei der Berechnung von Begriffen

Für jeden Kontext wurde die Menge aller Begriffe mit Ganters Algorithmus (Abschnitt 3.1.2, Seite 16) berechnet. Anschließend wurde die Verbandsstruktur der Begriffe mit Algorithmus-I (Abschnitt 3.6.1, Seite 31) bestimmt. Die Algorithmen waren in Objective Caml 2.01 [58] implementiert und wurden mit dem Native-Code-Compiler übersetzt. Sie liefen auf einem 200 MHz AMD K6 Prozessor unter Linux 2.0 ab; dabei wurden für jeden Kontext die Parameter in Tabelle 6.2 festgehalten. Die Zeiten zur Berechnung der Begriffe und ihrer Verbandsstruktur wurden separat erfaßt; ihre Summe ist die Zeit zur Berechnung des Begriffsverbandes im herkömmlichen Sinne, da diese eine Berechnung der Begriffe einschließt. Die Laufzeiten wurden mit Hilfe der Objective Caml Standard-Funktion `Sys.time` ermittelt.

6.2 Die Kontexte

Im Rahmen des Versuches wurden 3187 Kontexte generiert, die die zuvor genannten Kriterien erfüllten. Ihre Zusammensetzung kann an den vier empirischen Verteilungen in den Abbildungen 6.1 bis 6.4 abgelesen werden. Jede Abbildung zeigt die Verteilung aller Kontexte über einen Parameter aus Tabelle 6.1. Eine empirische Verteilung beschreibt vertikal den prozentualen Anteil y der Kontexte, deren Parameter *nicht größer* als ein horizontaler Wert x ist. Dabei entsprechen 100% auf der vertikalen Achse 3187 Kontexten. Diese Darstellung ergibt für jeden der 4 beobachteten Parameter eine streng monoton ansteigende Funktion; ein gleichverteilter Parameter spiegelt sich darin als eine Gerade wider. Zusätzlich enthalten die Abbildungen zur Anschauung die Häufigkeiten (Dichten) der Kontexte für die einzelnen Parameter. Dazu wurde der beobachtete Wertebereich eines Parameters in 30 gleichgroße Klassen geteilt, und für jede Klasse die Häufigkeit der darin liegenden Kontexte bestimmt. Die zugehörige Achse befindet sich in den Abbildungen 6.1 bis 6.4 auf der rechten Seite.

Die 3187 generierten Kontexte wiesen für die Parameter $|\mathcal{O}|$, *ratio* und *fill* eine annähernde Gleichverteilung auf, wie an den Abbildungen 6.1, 6.2 und 6.3 zu erkennen ist: die Verteilungsfunktionen sind nahezu linear ansteigende Kurven. Dies bedeutet, daß unter den generierten Kontexten keine Häufungen in Bezug

auf die Kantenlängen der Kontexte, das Längenverhältnis der beiden Kanten einer Kontexttabelle und ihres Füllgrades zu beobachten war. Wie bereits erläutert, wurden nur Kontexte bis zu einem maximalem Füllgrad von 0.1 und einer maximalen Kantenlänge von 200 betrachtet.

Die absolute Größe der Relationen, im Gegensatz zum Füllgrad als relatives Maß für die Größe, ist dagegen nicht gleichverteilt: die Verteilungsfunktion in Abbildung 6.4 zeigt die größere Häufigkeit kleiner Relationen (steiler Anstieg der Kurve) und nur eine geringe Häufigkeit großer Relationen. Schlußfolgerungen über Beobachtungen in Relation zu der Größe von Relationen müssen diese Ungleichverteilung berücksichtigen.

6.3 Charakterisierung von Kontexten

Die Zahl der Begriffe eines Kontextes und der Aufwand zu ihrer Berechnung sind die zentralen Punkte bei der praktischen Untersuchung der Komplexität der Begriffsanalyse. Gesucht sind Parameter, die die Eingabe (den Kontext) möglichst einfach charakterisieren und aus denen eine Aussage über die Zahl der Begriffe dieses Kontextes abgeleitet werden kann. Um diese Parameter zu ermitteln, wurde die Größe von Begriffsverbänden über verschiedene Parameter aufgetragen. Bei einem idealen Parameter besitzen alle Kontexte mit dem gleichen Wert des Parameters die gleiche Größe. Die Kontextgröße wurde über der folgenden Liste von Parametern aufgetragen, zusammen mit den dabei festgestellten Standardabweichungen:

1. Die Anzahl der Begriffe über dem relativen Belegungsgrad der Kontexttabelle in Abbildung 6.5,

2. die Anzahl der Begriffe über der durchschnittlichen Anzahl von Attributen pro Begriff in Abbildung 6.6,

3. die Anzahl der Begriffe über dem Seitenverhältnis der Kontexttabelle in Abbildung 6.7,

4. und die Anzahl der Begriffe über der absoluten Größe der Relation in Abbildung 6.8.

Die Größe der beobachteten Begriffsverbände ist in den Abbildungen 6.5 bis 6.8 vertikal aufgetragen, der zugehörige Parameter horizontal. Der Wertebereich des Parameters wurde jeweils in 20 gleich große Klassen geteilt; die Größen der darin beobachteten Begriffsverbände wurden gemittelt, sowie ihre Standardabweichung bestimmt. Der Mittelwert jeder Klasse ist als Punkt, die (einfache) Standardabweichung als vertikale Gerade eingetragen. Die Klasseneinteilung erlaubt durch die Berechnung von Mittelwert und Standardabweichung eine Beurteilung,

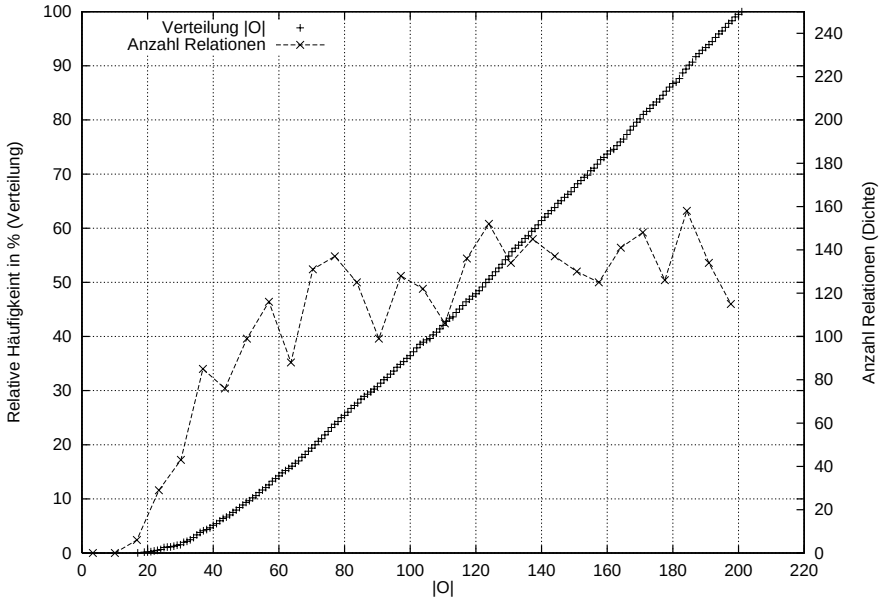

Abbildung 6.1: Verteilungsfunktion $f(|\mathcal{O}|)$ der generierten Kontexte. $f(x) = y$ bedeutet: für y Kontexte gilt $|\mathcal{O}| \leq x$.

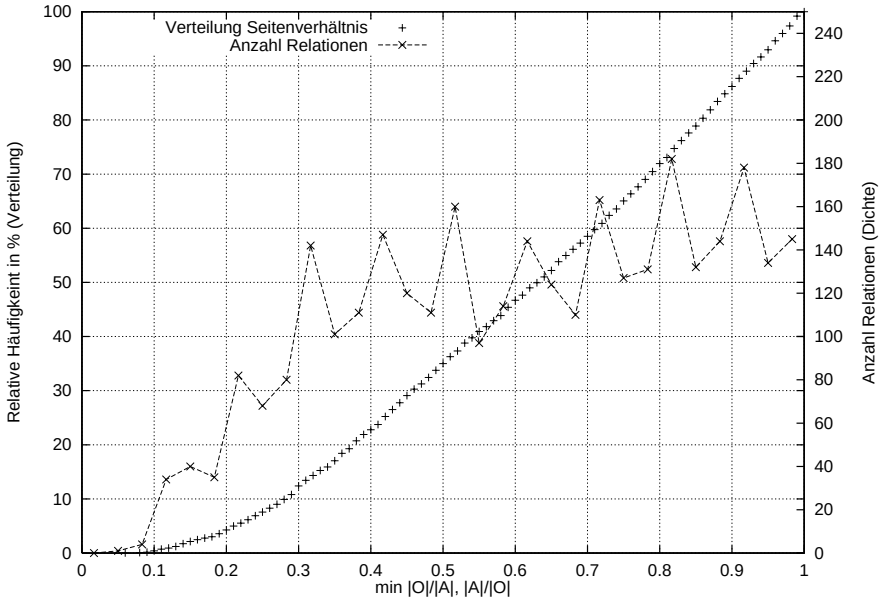

Abbildung 6.2: Verteilungsfunktion $f(ratio)$ (Seitenverhältnis der Kontexttabelle) der generierten Kontexte. $f(x) = y$ bedeutet: für y Kontexte gilt $ratio \leq x$.

wie funktional der Zusammenhang zwischen dem Parameter und der Größe von Begriffsverbänden ist. Im Falle der Relationengröße enthalten Klassen mit einem ansteigenden Wert des Parameters eine abnehmende Anzahl von Kontexten, da

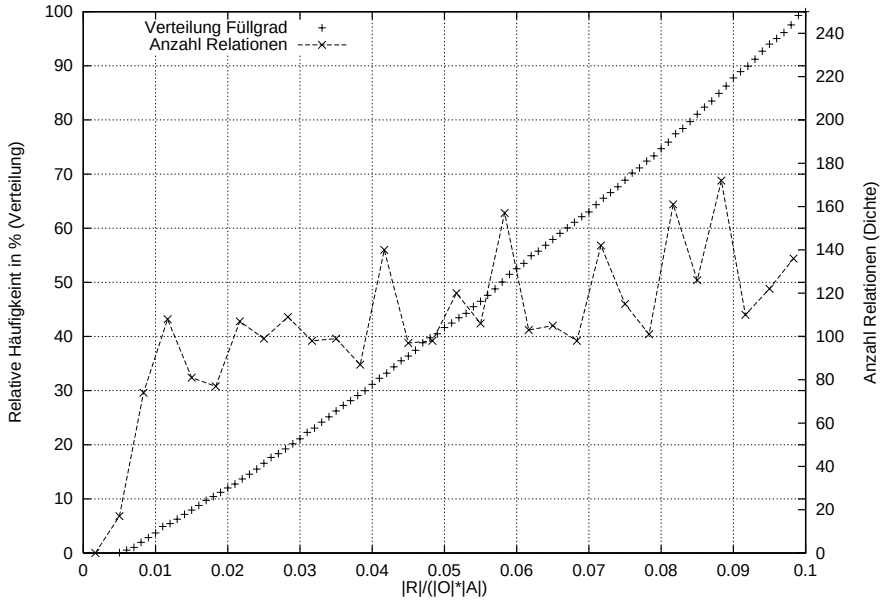

Abbildung 6.3: Verteilungsfunktion $f(\mathit{fill})$ (Füllgrad) der generierten Kontexte. $f(x) = y$ bedeutet: für y Kontexte gilt $\mathit{fill} \leq x$.

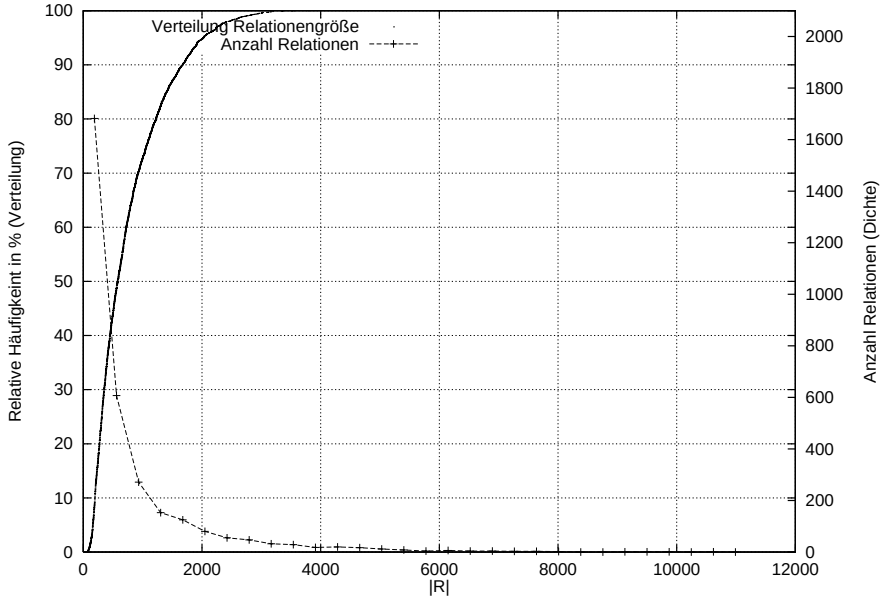

Abbildung 6.4: Verteilungsfunktion $f(|\mathcal{R}|)$ (Relationengröße) der generierten Kontexte. $f(x) = y$ bedeutet: für y Kontexte gilt $|\mathcal{R}| \leq x$.

Kontexte über diesem Parameter nicht gleichverteilt sind (vergleiche Abschnitt 6.2).

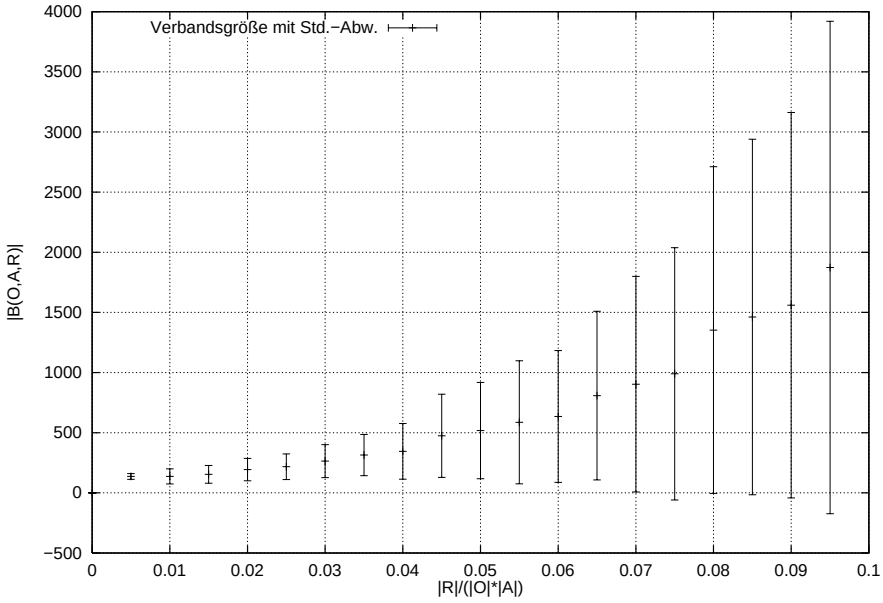

Abbildung 6.5: Beobachtete Größe von Begriffsverbänden in Abhängigkeit der relativen Dichte der Kontexttabelle.

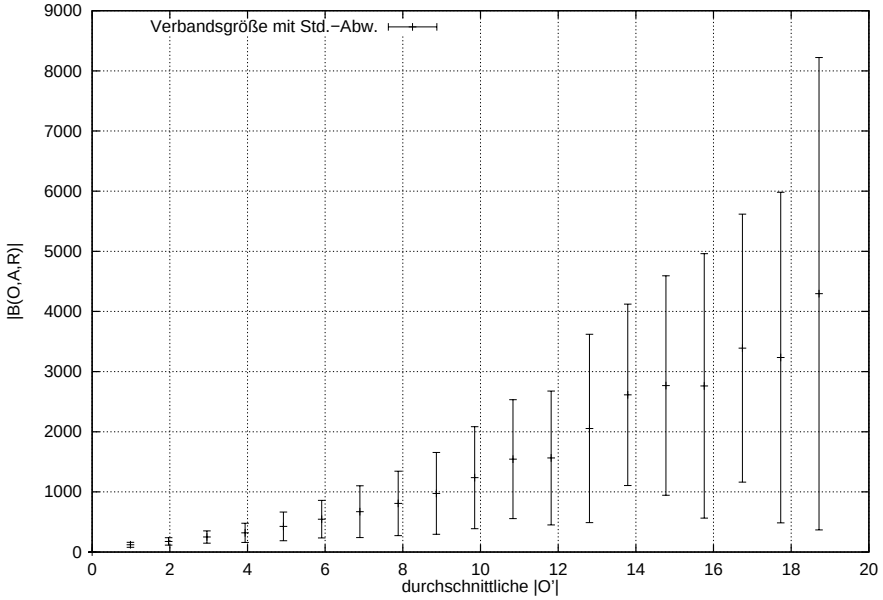

Abbildung 6.6: Beobachtete Größe von Begriffsverbänden in Abhängigkeit der durchschnittlichen Anzahl von Attributen, die in Relation zu einem Objekt stehen.

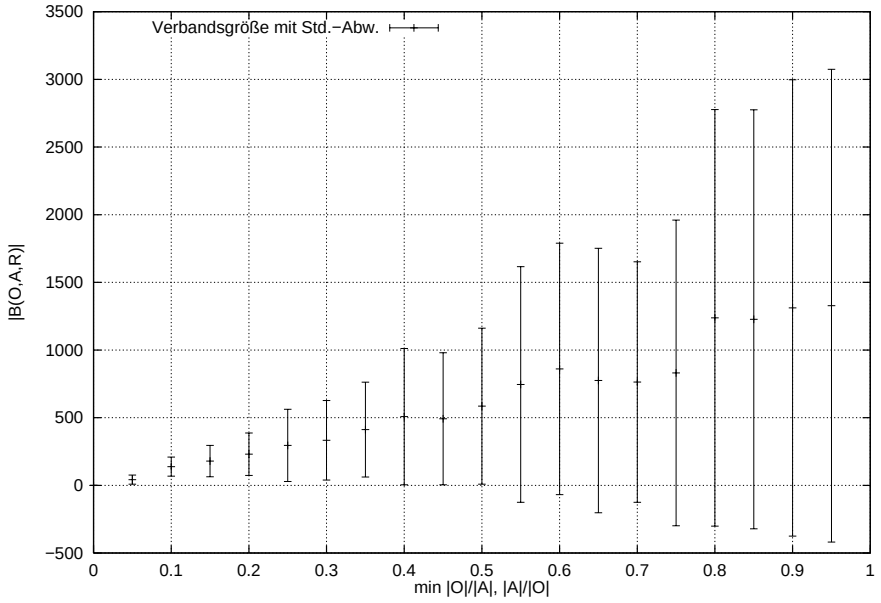

Abbildung 6.7: Beobachtete Größe von Begriffsverbänden in Abhängigkeit der Seitenverhältnisse der Kontexttabelle.

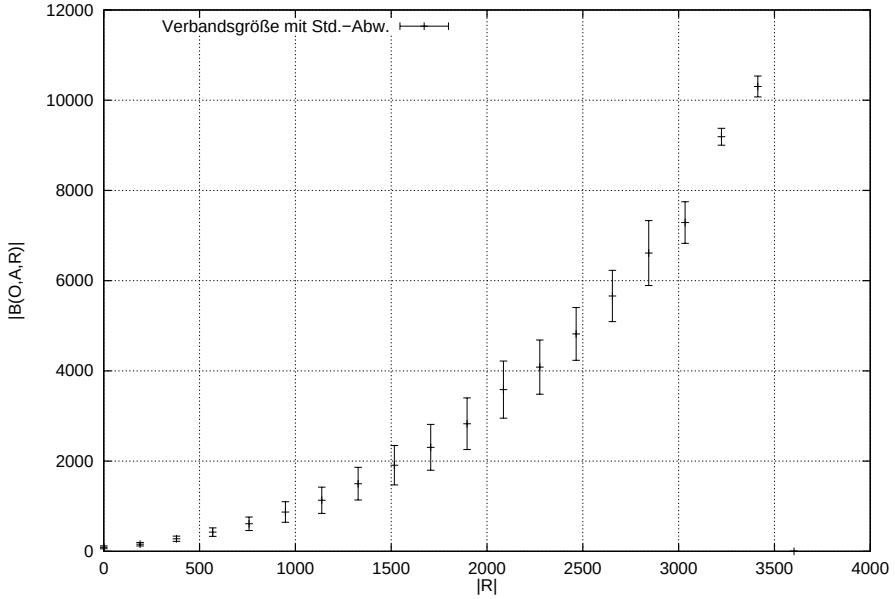

Abbildung 6.8: Beobachtete Größe von Begriffsverbänden in Abhängigkeit der absoluten Größe der Ausgangsrelation.

Bei allen beobachteten Parametern ist ein Anstieg der durchschnittlichen Verbandsgröße mit dem Anstieg des Parameters in den Abbildungen 6.5 bis 6.8 zu beobachten. Innerhalb einer Klasse von Kontexten ist die Größe von Begriffs-

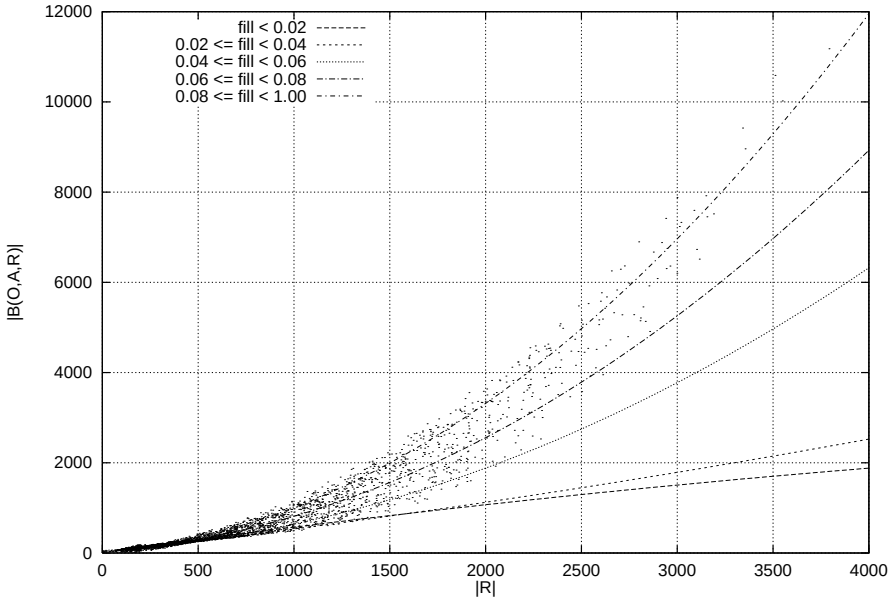

Abbildung 6.9: Größe von Begriffsverbänden in Abhängigkeit der absoluten Größe der Ausgangsrelation

verbänden stark schwankend, zu erkennen an den Standardabweichungen, wenn die Klasseneinteilung an Hand der relativen Dichte der Kontexttabelle, der durchschnittlichen Anzahl von Attributen und dem Seitenverhältnis der Kontexttabelle vorgenommen wurde. Dies läßt auf einen schwachen funktionalen Zusammenhang zwischen den Parametern des Kontextes und der Größe von Begriffsverbänden schließen. Deshalb sind diese Parameter zur Charakterisierung der Kontexte ungeeignet. Die absolute Größe $|\mathcal{R}|$ eines Kontextes dagegen scheint ein guter Parameter zu seiner Beschreibung zu sein: bei einer nach diesem Parameter vorgenommenen Klasseneinteilung ist die Streuung der Ergebnisse innerhalb einer Klasse gering (Abbildung 6.8). Dies kann trotz der Ungleichverteilung der Begriffsverbandsgrößen über diesem Parameter festgestellt werden: 16 von 20 Klassen enthalten mehr als 10 Kontexte, 11 Klassen mehr als 60 Kontexte. Damit sind Aussagen über Mittelwerte und Standardabweichungen in diesen Klassen aussagekräftig.

6.4 Die Größe von Begriffsverbänden

Auf Grund der Ergebnisse des letzten Abschnitts wurde für die weitere Auswertung des Versuchs die absolute Größe $|\mathcal{R}|$ der Ausgangsrelation $\mathcal{R}$ zur Charakterisierung von Kontexten verwendet. Interessanterweise nutzt auch Theorem 16 diesen Parameter; in der zugehörigen Arbeit begründet Schütt [96] seine Wahl aber nicht näher.

Abbildung 6.9 zeigt nochmals die beobachteten Verbandsgrößen in Abhängigkeit von der absoluten Größe der Ausgangsrelation; diesmal ist jede beobachtete Kontext/Begriffsverbandsbeziehung durch einen Punkt dargestellt. Obwohl sich die Größe $|\mathcal{R}|$ des Ausgangskontextes im letzten Abschnitt als das Maß mit der geringsten Standardabweichung bezüglich der Größe der entstehenden Verbände gezeigt hat, ist die absolute Differenz zwischen den beobachteten Verbandsgrößen *einer* Relationengröße $|\mathcal{R}|$ immer noch groß.

Um präzisere Aussagen zu erhalten, wurde nach einem zweiten Parameter gesucht, der Kontexte mit identischer Relationengröße $|\mathcal{R}|$ differenziert. Dieser Parameter soll Vorhersagen erlauben, ob bei einer gegebenen Relationengröße die zu erwartende Größe des Begriffsverbandes eher am oberen oder unteren Rand des Spektrums liegt, das in Abbildung 6.9 zwischen der obersten und untersten Kurve aufgespannt wird.

Für die Suche nach einem solchen Parameter wurden Kontexte an Hand drei verschiedener Parameter in je fünf Klassen eingeteilt. Anschließend wurde für jede Klasse die Beziehung zwischen $|R|$ und $|B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ durch ein Polynom zweiten Grades angenähert:

$$|B(\mathcal{O}, \mathcal{A}, \mathcal{R})| \approx a_2|\mathcal{R}|^2 + a_1|\mathcal{R}| + a_0$$

Die Koeffizienten des Polynoms wurden so bestimmt, daß sie die Summe der quadratischen Fehler zwischen dem Polynom und den Meßwerten minimieren. Dazu wurde die Implementierung des Marquardt-Levenberg Algorithmus in dem Programm *Gnuplot* [52] verwendet.

Ein Parameter weist die gesuchte Eigenschaft auf, wenn die in seinen Klassen enthaltenen Kontexte bei einer Darstellung über $|\mathcal{R}|$ klar zu unterscheiden sind. Würde man die Punkte in Abbildung 6.9 entsprechend ihrer Klassenzugehörigkeit einfärben, müßten sich bei einem geeigneten Parameter farblich klar abgegrenzte Punktewolken ergeben. Indirekt äußert sich dies darin, daß sich die Polynome zu Annährung der $|\mathcal{R}|$–$|B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ Relation in den einzelnen Klassen deutlich unterscheiden.

Abbildung 6.9 zeigt die fünf Polynome für eine Klasseneinteilung an Hand des relativen Füllgrades der Kontexttabelle. Die Koeffizienten, sowie weitere Merkmale der jeweiligen Klassen sind in Tabelle 6.3 zusammengefaßt; Die Spalte n gibt die Anzahl der Kontexte in der entsprechenden Klasse an.

Nicht in jeder Klasse traten alle Größen $|\mathcal{R}|$ von Relationen auf, wie in Tabelle 6.3 an den beobachteten maximalen und minimalen Größen abzulesen ist. Die in Abbildung 6.9 eingezeichneten Kurven können fälschlicherweise diesen Eindruck hervorrufen. Tatsächlich war unter den generierten Kontexten zum Beispiel kein Kontext mit $|\mathcal{R}| > 3000$ und $\mathit{fill} \leq 0.02$.

Die unterschiedlichen Näherungen für die Beziehung zwischen $|\mathcal{R}|$ und $|B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ in den einzelnen Klassen zeigen, daß der relative Füllgrad einer Kontexttabelle einen Einfluß auf die Größe der entstehenden Verbände besitzt. Dich-

fill	a_2	a_1	a_0	n	$\min \lvert\mathcal{R}\rvert$	$\max \lvert\mathcal{R}\rvert$
0.00 ... 0.02	-3.0688×10^{-5}	0.58955	13.776	357	70	592
0.02 ... 0.04	3.6527×10^{-5}	0.48174	14.153	599	38	1282
0.04 ... 0.06	0.00032621	0.26237	52.876	685	51	2286
0.08 ... 0.08	0.00048146	0.29659	36.275	697	74	2856
0.08 ... 1.00	0.00067034	0.29714	40.730	849	105	3792

Tabelle 6.3: Koeffizienten zur Approximation der Größe von Begriffsverbänden durch Polynome zweiten Grades. Klasseneinteilung von Kontexten an Hand des Parameters *fill*.

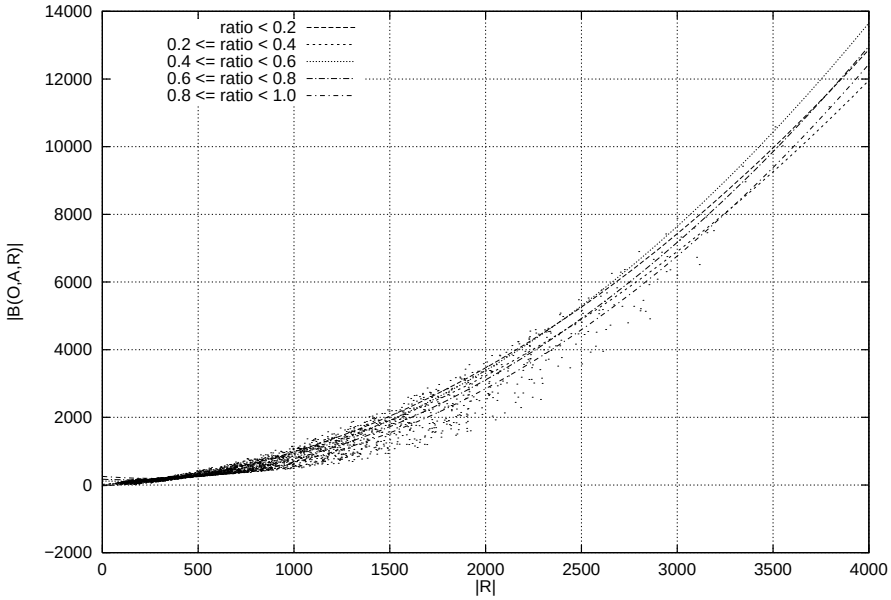

Abbildung 6.10: Beobachtete Größe von Begriffsverbänden in Abhängigkeit des Seitenverhältnis der Kontexttabelle

ter gefüllte Kontexttabellen führten in der Versuchsreihe zu größeren Begriffsverbänden.

Das Seitenverhältnis $\min(|\mathcal{O}|/|\mathcal{A}|, |\mathcal{A}|/|\mathcal{O}|)$ von Kontexttabellen scheint gegenüber der Relationengröße $|\mathcal{R}|$ nur geringen Einfluß zu besitzen. Bei einer Einteilung der generierten Kontexte in fünf Klassen an Hand ihrer Seitenverhältnisse und anschließender Annäherung ihrer Beziehung zwischen $|\mathcal{R}|$ und $|B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ ergeben sich sehr ähnliche Kurven (Abbildung 6.10, Tabelle 6.4). Das Seitenverhältnis von Kontexttabellen ist zusätzlich zur Größe einer Relation also kein geeigneter Parameter, um die Größe der entstehenden Begriffsverbände zu charakterisieren.

Als dritter Parameter zu einer Klasseneinteilung wurde die durchschnittliche Anzahl von Attributen verwendet, zu der ein Objekt eines Kontextes in Relation steht. Da im Rahmen der Versuchsreihe $|\mathcal{A}| \leq 200$ galt und gleichzeitig

ratio	a_2	a_1	a_0	n	$\min \lvert\mathcal{R}\rvert$	$\max \lvert\mathcal{R}\rvert$
0.0 ... 0.2	0.00074005	0.26319	-24.224	114	74	650
0.2 ... 0.4	0.00069038	0.22337	18.760	584	97	1341
0.4 ... 0.6	0.00087551	-0.11696	115.370	749	51	2325
0.6 ... 0.8	0.00086528	-0.26253	170.330	799	54	2748
0.8 ... 1.0	0.00087459	-0.45275	254.150	941	38	3792

Tabelle 6.4: Koeffizienten zur Approximation der Größe von Begriffsverbänden durch Polynome zweiten Grades. Klasseneinteilung an Hand des Parameters *ratio*.

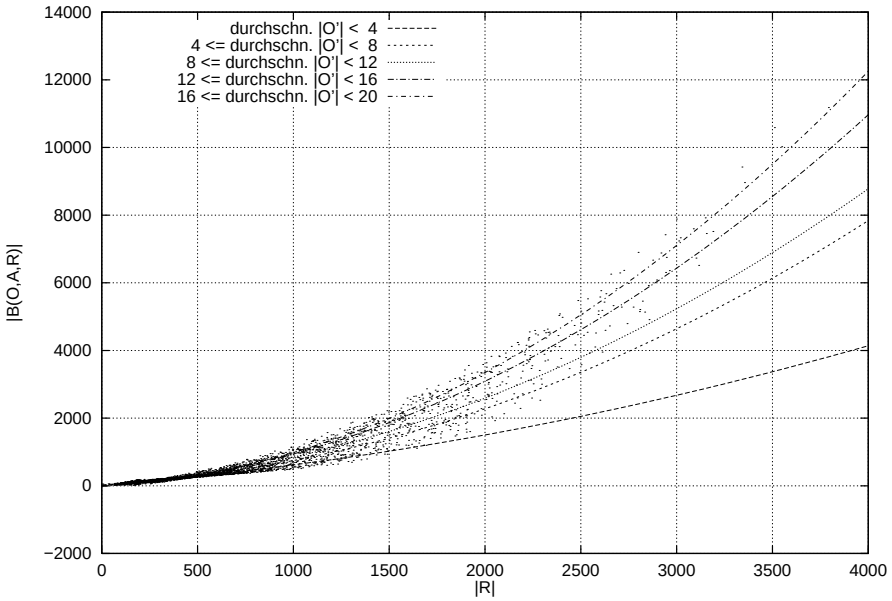

Abbildung 6.11: Beobachtete Größe von Begriffsverbänden in Abhängigkeit der absoluten Größe der Ausgangsrelation

fill ≤ 0.1, konnten durchschnittlich zwischen 1 und 20 Attribute pro Objekt beobachtet werden. Die Grenzen zur Klasseneinteilung lagen dementsprechend bei $0, 4, 8, 12, 16, 20$. Wiederum wurde für jede Klasse ein Polynom zweiten Grades zur Annährung der Relation zwischen $|\mathcal{R}|$ und $|B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ ermittelt (Tabelle 6.5), dessen Kurve in Abbildung 6.11 auch eingezeichnet ist.

Die durchschnittliche Anzahl von Attributen eines Objektes unterscheidet Kontexte mit ähnlicher Relationengröße $|\mathcal{R}|$ deutlich, wie an den verschiedenen Approximationen in Abbildung 6.11 abzulesen ist. Die unterscheidende Wirkung ist stärker als bei dem Seitenverhältnis (Abbildung 6.10), aber schwächer als bei dem Füllgrad des Kontextes (Abbildung 6.9). Im Vergleich mit der Partitionierung an Hand des Füllgrades sind die Partitionen ungleichmäßiger und die Annährung durch die Polynome ungenauer.

Unter den drei untersuchten Parametern Füllgrad, Seitenverhältnis und durch-

$\|O'\|$	a_2	a_1	a_0	n	$\min \|\mathcal{R}\|$	$\max \|\mathcal{R}\|$
0 ... 4	0.00014455	0.45034	23.711	1100	38	792
4 ... 8	0.00041637	0.28219	47.492	1019	74	1531
8 ... 12	0.00045114	0.38796	8.211	637	117	2286
12 ... 16	0.00059555	0.36328	-16.735	315	174	3149
16 ... 20	0.00069486	0.28172	5.521	116	258	3792

Tabelle 6.5: Koeffizienten zur Approximation der Größe von Begriffsverbänden durch Polynome zweiten Grades. Klasseneinteilung an Hand des Parameters $|O'|$.

schnittliche Anzahl von Attributen weist der relative Füllgrad die größte Selektivität auf: er unterscheidet im Versuch Kontexte mit gleich großen Ausgangsrelationen, so daß in der Kombination aus den beiden Kontext-Parametern Relationengröße und relativem Füllgrad genaue Aussagen über die zu erwartende Verbandsgröße möglich sind.

6.5 Komplexität der Begriffsanalyse

Während des Versuchablaufs wurden außer den Verbandsgrößen die Zeiten festgehalten, die zur Berechnung aller Begriffe und ihrer Verbandsstruktur benötigt wurden (siehe Tabelle 6.2, Seite 105). Die Berechnungen wurden auf einem 200 MHz AMD K6 Prozessor unter Linux ausgeführt. Zur Berechnung der Menge aller Begriffe wurde der Algorithmus von Ganter verwendet, wie er in den Abschnitten 3.1.2 und 3.2 (Seiten 16, 21) beschrieben ist. Die in der Menge aller Begriffe implizit enthaltene Verbandsstruktur wurde anschließend mit dem Algorithmus-I aus Abschnitt 3.6.1 bestimmt.

Der Zeitbedarf zur Berechnung der Begriffe und ihrer Verbandsstruktur wird im Wesentlichen durch die Anzahl der Begriffe im Verband bestimmt. Abbildung 6.12 zeigt den Zeitbedarf für Begriffe und Verbandsstruktur in CPU-Sekunden in Abhängigkeit von der Größe des Begriffsverbandes. Beide Meßreihen wurden durch Polynome zweiten Grades approximiert, die ebenfalls in Abbildung 6.12 eingezeichnet sind; wie im vorhergehenden Abschnitt wurde das Polynom mit der minimalen Summe der Fehlerquadrate zur Approximation verwendet.

$$t = a_2|B(\mathcal{O},\mathcal{A},\mathcal{R})|^2 + a_1|B(\mathcal{O},\mathcal{A},\mathcal{R})| + a_0 \quad \text{in CPU-Sekunden}$$

Zusätzlich ist die Summe der beiden Polynome eingezeichnet; diesen Zeitbedarf besitzt die Berechnung der Verbandsstruktur insgesamt, da die Berechnung der Verbandsstruktur die vorherige Bestimmung aller Begriffe voraussetzt. Die genauen Koeffizienten der Polynome sind in Tabelle 6.6 aufgeführt.

Abbildung 6.12 zeigt, daß bei größeren Begriffsverbänden nicht die Berechnung der Begriffe, sondern die der zugehörigen Verbandsstruktur den Gesamtaufwand bestimmte. Der Faktor, um den die Berechnung der Verbandsstruktur

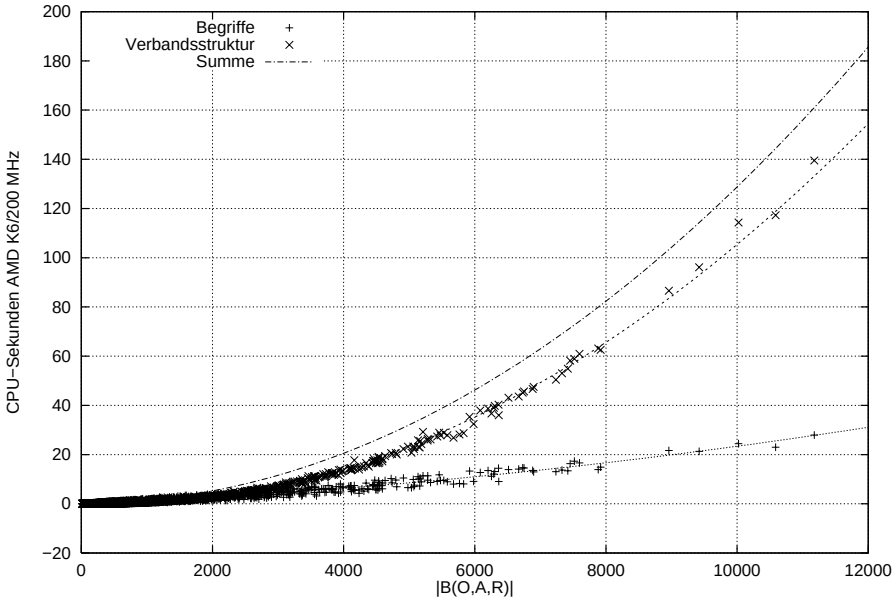

Abbildung 6.12: Zeitbedarf zur Berechnung aller Begriffe mit Ganters Algorithmus und ihrer Verbandsstruktur mit Algorithmus I.

Operation	a_2	a_1	a_0
Verbandsstruktur	1.1721×10^{-6}	-0.0012147	0.338980
Begriffe	1.2663×10^{-7}	0.0010740	-0.011212

Tabelle 6.6: Koeffizienten für Polynome zur Annäherung des Zeitbedarfs zur Berechnung aller Begriffe und ihrer Verbandsstruktur.

länger mit Algorithmus-I dauerte als die Berechnung der Begriffe, ist in Abbildung 6.13 aufgetragen. Jedes Kreuz in der Abbildung kennzeichnet einen Versuchsausgang und die Kurve ist ein Annährung durch ein Polynom zweiten Grades.

6.5.1 Komplexität von Algorithmus-II

Bei der Durchführung der Experimente zur Größe von Begriffsverbänden war Algorithmus-II (basierend auf [30], Abschnitt 3.6.2) zur gleichzeitigen Berechnung von Begriffen und ihrer Verbandsstruktur noch nicht bekannt. Sein Laufzeitverhalten konnte deswegen nicht mit den Kontexten gemessen werden, die in Abschnitt 6.5 zur Bewertung von Ganters Algorithmus und Algorithmus-I verwendet wurden. Für einen Vergleich von Ganters Algorithmus, Algorithmus-I und -II wurden deswegen zwei Experimente mit zufällig generierten Kontexten durchgeführt.

Die theoretischen Komplexitätsbetrachtungen in den Abschnitten 3.1.2 und 3.6.2 und die Versuche im Abschnitt 6.5 zeigen, daß die Laufzeit der Algorithmen

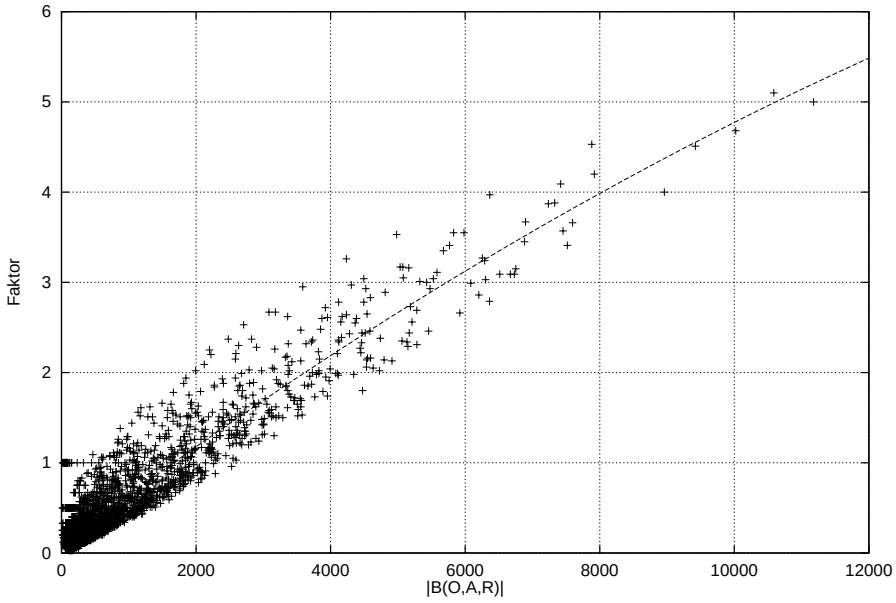

Abbildung 6.13: Zeitbedarfsfaktor zwischen Berechnung aller Begriffe und ihrer Verbandsstruktur in Abhängigkeit von der Verbandsgröße.

primär durch die Anzahl der Begriffe des analysierten Kontextes bestimmt wird. Deswegen konzentrieren sich beide Experimente auf den Zusammenhang zwischen der Größe der Verbände und der Laufzeit der Algorithmen und diskutieren den Einfluß anderer Parameter nicht.

Das erste Experiment vergleicht die Laufzeit der drei Algorithmen (Ganter, I, II) an Hand von 1 000 zufällig generierten Kontexten. Die Kontexte enthielten zwischen 2 und 1572 Elemente und erzeugten Verbände mit bis zu 11 148 Begriffen. Der Füllgrad der Kontexttabellen variierte zwischen 0.01 und 1.00 und war damit wesentlich weiter als bei den Experimenten in Abschnitt 6.2. Alle Algorithmen waren in Objective Caml 2.02 [58] implementiert und liefen als *native code* auf einem AMD K6/200 MHz Prozessor unter Linux 2.0 ab.

Abbildung 6.14 zeigt die gemessenen Laufzeiten in CPU-Sekunden zusammen mit einer Approximation des gefundenen Zusammenhanges. Die Approximation ist das Polynom zweiten Grades mit der minimalen Summe der Fehlerquadrate; seine Koeffizienten sind in Tabelle 6.7 unter *Experiment 1* zusammengefaßt. Dabei ist zu beachten, daß die hier verglichenen Algorithmen verschiedene Daten verwenden und berechnen:

1. Ganters Algorithmus berechnet die Menge aller Begriffe eines Kontextes.

2. Algorithmus-I berechnet für die Menge aller Begriffe ihre Verbandsstruktur.

3. Algorithmus-II berechnet für einen Kontext gleichzeitig die Menge seiner

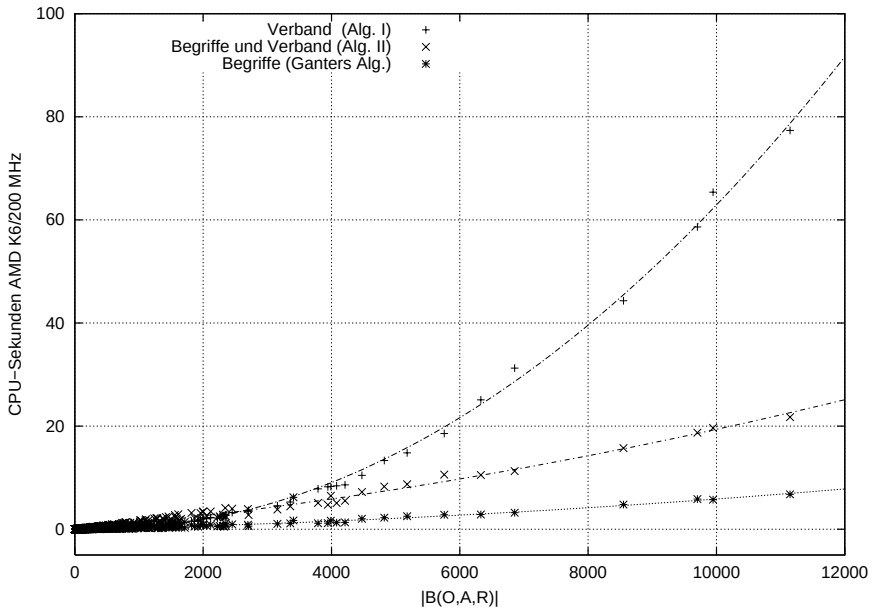

Abbildung 6.14: Zeitbedarf zur Berechnung aller Begriffe mit verschiedenen Algorithmen in Abhängigkeit der Verbandsgröße.

Algorithmus	a_2	a_1	a_0
	Experiment 1		
Ganters Algorithmus	3.255×10^{-8}	0.00026	0.00101
Algorithmus-I	6.721×10^{-7}	-0.00043	0.04699
Algorithmus-II	7.764×10^{-8}	0.00116	-0.03158
	Experiment 2		
Ganters Algorithmus	1.858×10^{-8}	0.00024	-0.00332
Algorithmus-II	3.542×10^{-8}	0.00117	-0.06581

Tabelle 6.7: Koeffizienten für Polynome zweiten Grades $t = a_2|B(\mathcal{O},\mathcal{A},\mathcal{R})|^2 + a_1|B(\mathcal{O},\mathcal{A},\mathcal{R})| + a_0$ zur Annäherung des Zeitbedarfs in CPU-Sekunden für die Berechnung aller Begriffe und ihrer Verbandsstruktur.

> Begriffe und ihre Verbandsstruktur (in Form direkter Ober- und Unterbegriffe für jeden Begriff).

Bereits bei der Vorstellung von Algorithmus-II in Abschnitt 3.6.2 wurde auf Grund seiner asymptotischen Komplexität von $O(|B(\mathcal{O},\mathcal{A},\mathcal{R})| \times |\mathcal{O}| \times |\mathcal{A}|)$ eine Laufzeit in der Größenordnung von Ganters Algorithmus vorausgesagt: tatsächlich ist die Laufzeit von Algorithmus-II etwa doppelt so hoch wie von Ganters Algorithmus und quasi linear vor der Größe der Begriffsverbände abhängig. Damit ist Algorithmus-II bei der Berechnung der Verbandsstruktur Algorithmus-I klar überlegen, der eine quadratische Abhängigkeit von $|B(\mathcal{O},\mathcal{A},\mathcal{R})|$ aufweist.

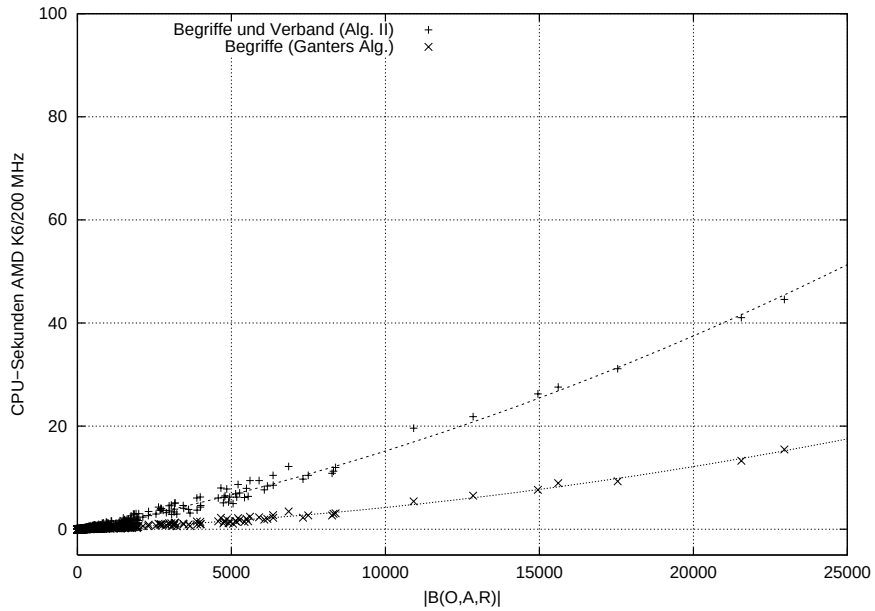

Abbildung 6.15: Vergleich von Ganters Algorithmus mit Algorithmus-II bei großen Verbänden. Laufzeit in CPU-Sekunden auf einem AMD K6/200 MHz Prozessor.

Bei relativ kleinen Verbänden ($|B(\mathcal{O}, \mathcal{A}, \mathcal{R})| < 2\,000$) ist Algorithmus-I dennoch effizienter, da Begriffe in einem Array statt in einem Binärbaum abgelegt werden (vergleiche Abschnitt 3.6.1 und 3.6.2).

In einem zweiten Experiment wurden Ganters Algorithmus und Algorithmus-II nochmals speziell für große Begriffsverbände verglichen. Die Implementierung von Algorithmus-I nahm wegen ihres Speicherbedarfs als Folge naiver Datenstrukturen (vergleiche Abschnitt 3.6.1) nicht teil. Es wurden unter den gleichen Umständen wie im ersten Versuch 1 000 Kontexte zufällig generiert, und anschließend die Laufzeiten der Algorithmen gemessen. Abbildung 6.15 faßt die Beobachtungen zusammen; die Koeffizienten der ebenfalls eingezeichneten Approximationen durch Polynome zweiten Grades stehen unter *Experiment 2* in Tabelle 6.7. Die Unterschiede zwischen den in Experiment 1 und 2 gewonnenen Approximationen sind vermutlich auf die Abhängigkeit der Algorithmen von nicht erfaßten Faktoren (wie Anzahl der direkten Vorgänger pro Begriff) zurückzuführen. Die hier gewonnenen Ergebnisse können auch deshalb nur eine Tendenz aufzeigen, weil die Implementierung aller Algorithmen noch Raum für Optimierungen läßt.

Das Verhältnis der Laufzeiten von 1 zu 2 der beiden Algorithmen bleibt auch bei großen Verbänden gewahrt. Die Begriffe und ihre Verbandsstruktur von Kontexten mit mehr als 25 000 Begriffen kann Algorithmus-II in weniger als 60 Sekunden berechnen. Die Approximation für Algorithmus-I aus Tabelle 6.7 sagt dafür eine Laufzeit von über 400 Sekunden voraus und zeigt nochmals deutlich die Überlegenheit von Algorithmus-II.

6.6 Ergebnisse

In einer Versuchsreihe wurden über 3000 Kontexte zufällig generiert und die zugehörigen Begriffsverbände bestimmt. Die generierten Kontexttabellen wiesen eine maximale Größe von 200 Objekten und 200 Attributen auf, die entstandenen Begriffsverbände enthielten zwischen 13 und 11 177 Begriffe. Alle betrachteten Kontexttabellen waren bis zu 10% ihrer maximalen Auslastung gefüllt. Zur Beschreibung von Kontexten im Hinblick auf die Größe ihrer Begriffsverbände ist die Größe $|\mathcal{R}|$ der Ausgangsrelation geeignet. Zwischen ihr und der Größe $|B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ der zugehörigen Begriffsverbände konnte ein weitgehend funktionaler Zusammenhang beobachtet werden, der sich durch ein Polynom zweiten Grades annähern läßt. Die Annährung durch das Polynom läßt sich verbessern, wenn zusätzlich der Füllgrad der Kontexttabelle berücksichtigt wird. Er unterscheidet Kontexte mit gleich großen Relationen aber verschiedenen Füllgraden, da diese signifikant verschieden große Begriffsverbände erwarten lassen. Von zwei Kontexten gleicher Größe aber verschiedener Füllgrade besitzt der Kontext mit höherem Füllgrad typischerweise die größere Anzahl von Begriffen. Die Koeffizienten der Polynome in Abhängigkeit vom Füllgrad des Kontextes sind in Tabelle 6.3 zusammengefaßt.

Die in der Versuchsreihe verwendete Implementierung der Begriffsanalyse berechnet die Menge aller Begriffe durch Ganters Algorithmus. Die implizit in dieser Menge enthaltene Verbandsstruktur wurde anschließend mit Algorithmus-I (Abschnitt 3.6.1) berechnet. Dabei überstieg der Zeitbedarf zur Berechnung der Verbandsstruktur den Zeitbedarf zur Berechnung der Begriffe bis um den Faktor 5. Der Zeitbedarf für Algorithmus-I bestimmt also die Komplexität der Gesamtanalyse; im Laufe des Versuches konnte eine quadratische Abhängigkeit seiner Laufzeit von der Anzahl der Begriffe im Verband beobachtet werden. Für einen Begriffsverband mit circa 10 000 Begriffen wurde eine Gesamtlaufzeit von circa 130 CPU-Sekunden beobachtet, wobei ungefähr 100 CPU-Sekunden alleine auf die Berechnung der Verbandsstruktur entfielen.

Eine wesentliche Verbesserung bei der Berechnung der Verbandsstruktur stellt die Verwendung von Algorithmus-II aus Abschnitt 3.6.2 dar. Er berechnet gleichzeitig die Begriffe eines Kontextesund ihre Verbandsstruktur; seine Komplexität $O(|B(\mathcal{O}, \mathcal{A}, \mathcal{R})| \times |\mathcal{O}| \times |\mathcal{A}|)$ resultiert auch in der Praxis zu einem quasi linear von $|B(\mathcal{O}, \mathcal{A}, \mathcal{R})|$ abhängigen Laufzeitverhalten. Bei zwei Versuchsreihen war seine gemessene Laufzeit etwa doppelt so hoch wie von Ganters Algorithmus und lag in der Größenordnung von 2 ms pro Begriff auf einem AMD K6/200 MHz Prozessor. Damit lassen sich Verbände von über 25 000 Begriffen in weniger als einer Minute bestimmen.

Die begriffsbasierte Komponentensuche benötigt lediglich die Menge aller Begriffe eines Kontextes, nicht aber ihre explizite Verbandsstruktur. Deshalb ist die ungünstige Komplexität von Algorithmus-I für diese Anwendung ohne Auswirkungen. Betrachtet man lediglich den Zeitaufwand zur Berechnung der Be-

griffe, so können für Komponentensammlungen mit 200 Komponenten und 200 Schlüsselwörtern zur Indexierung alle Begriffe innerhalb von 20 CPU-Sekunden auf einem 200 MHz Prozessor der Pentium-Klasse bestimmt werden. Zur Analyse von Komponentensammlungen (Abschnitt 5.8) muß dagegen auch die Verbandsstruktur berechnet werden. Hier stellen sowohl die Komplexität, als auch der Speicherbedarf der naiven Datenstrukturen von Algorithmus-I einen Engpaß dar (vergleiche Abschnitt 3.6.1). Realistische Anwendungen werden daher Algorithmus-II verwenden; neben der besseren Komplexität ist die Implementierung von Algorithmus-II auch einfacher als die sonst nötige Implementierung von Ganters Algorithmus in Verbindung mit Algorithmus-I. Insgesamt erweist sich die theoretisch exponentielle Komplexität der Begriffsanalyse in der Praxis als erstaunlich gut zu handhaben.

Kapitel 7

Vergleich

Eine Einordnung der in dieser Arbeit vorgeschlagenen Organisation von Softwarekomponenten durch formale Begriffe in die bekannten Methoden der komponentenbasierten Wiederverwendung fällt nicht eindeutig aus. Begriffsbasierte Komponentensammlungen kombinieren Merkmale des Information-Retrieval, von deskriptiven und wissensbasierten Methoden, ohne zugleich ihre Nachteile zu besitzen; einen Vergleich nimmt der erste Teil dieses Abschnitts vor. Der Vorschlag, formale Begriffe zur Strukturierung von Informationen zu verwenden, ist nicht völlig neu. Ganter, Godin, Fischer und Priss haben ähnliche Vorschläge gemacht und zum Teil implementiert. Auf ihre Gemeinsamkeiten mit, und Unterschiede zu, dieser Arbeit geht der zweite Teil ein. Der dritten Teil schließlich betrachtet Verfahren, die formale Begriffsanalyse zur globalen Analyse von Daten verwenden. Sie profitieren von den Ergebnissen dieser Arbeit über die Implementierung und Komplexität der Begriffsanalyse.

7.1 Traditionelle Methoden

7.1.1 Deskriptive Methoden

Die Organisation von Softwarekomponentensammlungen aus Abschnitt 5 kann am ehesten als deskriptive Methode (vergleiche Abschnitt 4.2.2) bezeichnet werden. Bei der facettierten Klassifikation [87], dem bekanntesten Beispiel für deskriptive Methoden, besteht jede Komponentenbeschreibung aus einem Vektor von Schlüsselwörtern. Statt der Komponenten selbst, also ihrem Quell- oder Objektcode, wird als Surrogat ihre Dokumentation indexiert und verwaltet. Die Länge des Vektors, sowie die Bedeutung und der Wertebereich jeder seiner Elemente (*Facetten*) wird bei dem Entwurf einer Komponentenablage festgelegt. Anfragen sind Vektoren, die die Belegung für die gesuchten Komponenten ganz oder teilweise vorgeben. Um synonyme oder unscharfe Anfragen bearbeiten zu können, wird für den Wertebereich jeder Vektorkomponente beim Entwurf (manuell) ein

	Facette 1			Facette 2			...	Facette n		
Komponente	a_1^1	...	a_n^1	a_1^2	...	a_n^2	...	a_1^n	...	a_n^n
k_1	×			×			...		×	
k_2		×				×	...			×
k_3		...			...		...		...	

Abbildung 7.1: Darstellung facettierter Klassifikation als Kontext

semantisches Netzwerk spezifiziert, das die Ähnlichkeit zwischen den Werten erfaßt. Die begriffsbasierte Organisation von Komponenten verwaltet ebenfalls die Dokumentation als Surrogat der eigentlichen Komponente; ihre Indexierung und Anfragen sind Mengen, statt Vektoren, von beschreibenden Schlüsselwörtern.

Die facettierte Klassifikation läßt sich mit der Theorie formaler Begriffe partiell erklären und deshalb mit der vorgeschlagenen begriffsbasierten Organisation von Komponenten vergleichen. Jeder Wertebereich einer Facette bildet eine Menge von Attributen und ihre disjunkte Vereinigung die Menge alle Attribute – siehe Abbildung 7.1. Jede Komponente steht mit genau einem Attribut aus jeder Facette in Relation – im Gegensatz zum Vorschlag in Abschnitt 5, bei dem keine solche Einschränkung existiert. Alternativ kann jede Facette auch als ein mehrwertiges Attribut aufgefaßt werden, das durch eine Nominalskala in einen zweiwertigen Kontext übertragen wird (vergleiche [32]). Der noch folgende Abschnitt 7.2.4 geht genauer auf mehrwertige Kontexte ein.

Das zwischen den Elementen eines Wertebereiches einer Facette aufgespannte semantische Netzwerk wird durch den skizzierten Kontext in Abbildung 7.1 nicht erfaßt. Uta Priss verwendet die angedeuteten mehrwertigen Kontexte zur Organisation von Komponenten zusammen mit einem Thesaurus, der diesem Netzwerk vergleichbar ist; ihr Vorschlag entspricht einer verallgemeinerten facettierten Klassifikation und wird ebenfalls in dem noch folgenden Abschnitt 7.2.1 besprochen.

Der Hauptnachteil der facettierten Klassifikation ist ihre starre Struktur. Nachdem beim Entwurf einer Ablage die Bedeutung und der Wertebereich aller Facetten festgelegt sind, erfordert eine spätere Erweiterung oder Änderung die Neuindexierung sämtlicher Komponenten und die Überarbeitung der semantischen Netze. Zusätzlich wird eine (halb-) automatische Indexierung durch die Analyse von Texten erschwert, da Schlüsselwörter Facetten zugeordnet werden müssen. Die Spezifikation des semantischen Netzwerkes ist überhaupt nur manuell möglich [80]. Als Folge davon beurteilen Mili et al. die initialen und laufenden Kosten dieser Methode als hoch [71] (Abschnitt 4.3.3).

Das Fehlen einer Meta-Struktur in der Indexierung von Komponenten in einer begriffsbasierten Komponentensammlung begünstigt im Vergleich ihre Erweiterbarkeit. Die Aufnahme neuer Komponenten kann eine Verschlechterung der Indexierung zwar nicht ausschließen, aber sie kann durch eine globale Analyse erkannt werden. Eine in diesem Fall nötige Umstrukturierung kann frühere Entwurfsent-

scheidungen nicht gefährden und betrifft in der Regel nur wenige Komponenten. Die Synonymität von Schlüsselwörtern ergibt sich in einem Kontext automatisch und nicht durch eine manuelle Spezifikation. Zwei synonyme Attribute a_1 und a_2 sind durch ihren gemeinsamen Attributbegriff gekennzeichnet: $\mu(a_1) = \mu(a_2)$.

Die einfache Struktur begriffsbasierter Komponentensammlungen ermöglicht zumindest eine teilweise automatische Indexierung von Komponenten, wie sie in Abschnitt 5.8.1 gezeigt wurde. Dies rückt sie in die Nähe von Information-Retrieval-Methoden [95] (Abschnitt 4.2.1), ohne daß sie allerdings die ausgefeilten Text-Analyse-Algorithmen des Information-Retrieval verwenden. In Information-Retrieval-Systemen entscheidet die syntaktische Ähnlichkeit einer Anfrage mit einer Komponente über deren Teilnahme am Ergebnis, während bei begriffsbasierten Sammlungen Anfragen und Indexierung in einer definierten Relation zueinander stehen müssen. Durch umfangreiche Indexierungen für jede Komponente könnte diese scharfe Grenze verwischt werden; durch die dann zahlreichen Synonyme entstehen große Klassen äquivalenter Anfragen.

7.1.2 Wissensbasierte Methoden

Eine möglichst natürliche Präsentation und Organisation von Softwarekomponenten ist der Anspruch von wissensbasierten Methoden (vergleiche Abschnitt 4.2.5). Ihre Grundlagen sind zu gleichen Teilen die Eigenschaften von Software sowie kognitive und ergonomische Aspekte. Die zentrale Annahme ist, daß Anwender ihr Informationsbedürfnis bei einer Suche oft nur ungenau kennen oder formulieren. Deshalb bieten wissensbasierte System intensive Unterstützung bei der Formulierung einer Anfrage, die zudem als Teil eines iterativen Prozesses aufgefaßt wird. Typische Hilfestellungen sind die Möglichkeit der Um-Formulierung einer Anfrage, Eingabe durch Auswahl, Präsentation von Ergebnissen in der Reihenfolge ihrer Relevanz. Da die meisten zur Implementierung der Hilfen nötigen Strukturen manuell angelegt werden müssen, beurteilen Mili et al. die entstehenden Kosten für Aufbau und Pflege eines Systems als *very high* [71] (vergleiche Abschnitt 4.3.3).

Der Begriffsverband einer Komponentensammlung ist eine natürliche hierarchische Struktur. Sie ermöglicht die Implementierung einer den wissensbasierten Systemen ähnlichen Benutzungsschnittstelle, aber ohne deren manuellen Spezifikationsaufwand. Anfragen werden durch die iterative Auswahl kontextsensitiv angebotener Attribute formuliert; alternativ können Anfragen mit Relevance-Feedback durch die Auswahl von Beispielen aus dem aktuellen Ergebnis spezialisiert werden (vergleiche Abschnitte 5.5, 5.6). Die Benutzerführung der prototypischen Implementierung (Abschnitt 5.1) basiert auf der Eingabe durch Auswahl; sie bestimmt die syntaktische Form der Eingabe vollständig und gestattet dem Benutzer, sich auf inhaltliche Aspekte zu konzentrieren. Da der Begriffsverband die Anfragesituationen vollständig abdeckt, können Ergebnisse und angebotene Auswahlen vorberechnet werden, was zu einer effizienten Ausführung aller An-

fragen führt. Der Begriffsverband als semantische Struktur muß nicht manuell spezifiziert werden, sondern wird aus den individuell indexierten Komponenten einmalig berechnet. Dies führt zu einer hohen Änderungsfreundlichkeit und, im Vergleich zu wissensbasierten Methoden, zu geringen Kosten.

Eine Reihe von Systemen zur komponentenbasierten Wiederverwendung erlauben eine interaktive Verallgemeinerung und Spezialisierung von Anfragen mittels einer hierarchischen Ordnung ihrer Komponenten [69, 43, 61, 44]. Diese Hierarchien besitzen eine Baumstruktur, mit genau einem Pfad von der Wurzel zu jeder Komponente. Als Folge davon muß ein Anwender die Entscheidungen über seinen Weg in einer eindeutigen Reihenfolge treffen. In einem (Begriffs-) Verband existieren in der Regel mehrere Wege vom größten Element zu einem kleineren Element. Ein Anwender kann deshalb häufig eine anstehende Richtungsentscheidung zugunsten der Beantwortung einer anderen zurückstellen in der Hoffnung, mit ihrem Ergebnis auch Klarheit über die richtige erste Entscheidung zu erhalten. Im Falle der vorgeschlagenen Benutzungsoberfläche in Abschnitt 5 äußert sich dies darin, daß in einem gewissen Rahmen die Reihenfolge bei der Auswahl von Attributen frei ist. Die größere Flexibilität bei gleicher Genauigkeit macht Verbandsstrukturen gegenüber Bäumen als Organisationsprinzip überlegen.

7.2 Begriffsbasierte Informationssysteme

7.2.1 Thesaurus für Attribute

Bei dem in dieser Arbeit vorgeschlagenen Verfahren zur Organisation von Softwarekomponenten sind die Attribute $\mathcal{A}$ zur Indexierung von einander unabhängig. In [91] schlägt Priss eine Meta-Struktur für Attribute durch Einführung eines Thesaurus vor. Der Thesaurus wird durch einen Kontext definiert, der Attribute aus $\mathcal{A}$ zu sogenannten Klassen in Beziehung setzt. Ein Attribut $a \in \mathcal{A}$ kann ein Element einer oder mehrerer Klassen sein. Die Kombination aus dem Dokumenten-Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ und dem Thesaurus-Kontext erzeugt verschiedene neue Kontexte. Diese erlauben, ja nach Konstruktionsprinzip der neuen Kontexte, die (Meta-) Suche von Dokumenten alleine an Hand ihrer Klassen, oder einer Mischung aus Klassen und Attributen. Im Vergleich mit den klassischen Methoden zur Komponentenorganisation kann der Vorschlag von Priss als eine verallgemeinerte facettierte Klassifikation bezeichnet werden. Dabei entsprechen die Klassen des Thesaurus den Facetten, die die Attribute zur Indexierung gruppieren.

Die Möglichkeiten zur Organisation und Suche durch die Einführung eines Thesaurus werden von Priss nur skizziert, da eine Formalisierung oder Definition von Anfrage und Ergebnis fehlt. Die Ursache könnte die vorgeschlagene, ausschließlich grafische, Navigation mit Hilfe des Hasse-Diagramms des Begriffsverbandes sein; sie verwischt die Begriffe Anfrage und Ergebnis. Da eine übersicht-

liche Darstellung von Begriffsverbänden mit mehr als 100 Elementen schwierig ist (vergleiche [98]), bleibt offen, wie praktikabel die vorgeschlagene Benutzungsoberfläche ist. Die aus Platzgründen wahrscheinlich notwendige Beschränkung der dargestellten Begriffe führt zu einem Verlust an wählbaren Aktionen für den Benutzer. Das Problem der grafischen Darstellung ist einer der Gründe, warum in dieser Arbeit der Begriffsverband nur als eine dem Benutzer verborgene Datenstruktur verwendet wird, und nur die für die Suche relevanten Informationen vollständig präsentiert werden.

Der von Priss vorgeschlagene Thesaurus ist unabhängig von einer verbandsbasierten grafischen Benutzungsoberfläche. Da eine um einen Thesaurus erweiterte Komponentensammlung wiederum ein Kontext ist, bleiben die in dieser Arbeit vorgeschlagenen Möglichkeiten der Suche und Analyse weiterhin gültig, so daß die Integration eines Thesaurus möglich ist. Es bleibt zu untersuchen, welche Aspekte ein Thesaurus adressieren könnte und bei welchen Komponentensammlungen sein Einsatz sinnvoll ist.

7.2.2 Schrittweise Navigation

Ganter und Bittner schlagen in [30] eine schrittweise Navigation innerhalb eines Begriffsverbandes vor, die an die Navigation in einem Dateisystem angelehnt ist. Ausgehend von einem aktuellen Begriff existieren Kommandos um:

- alle direkten Ober- und Unterbegriffe anzuzeigen,
- Objekt- und Attributmengen des aktuellen Begriffs anzuzeigen,
- einen Begriff zum aktuellen Begriff zu machen.

Jeder Begriff besitzt eine (künstliche) Identität, die an Kommandos übergeben werden kann. Analog zur Navigation in einem Dateisystem kann man sich vorstellen, daß die Angabe von Begriffen relativ vom aktuellen Begriff, oder absolut von einem anderen bekannten Begriff aus erfolgen kann. Die Navigation kann, muß aber nicht, in Verbindung mit einer graphischen Präsentation des Verbandes erfolgen. Damit werden die im vorhergehenden Abschnitt beschriebenen Probleme vermieden.

Ganter und Bittner haben den Algorithmus zur Berechnung aller direkten Oberbegriffe (Abschnitt 3.6.2) für diese Anwendung entworfen. Statt den vollständigen Begriffsverband vorab zu berechnen, kann dieser Algorithmus zum Zeitpunkt der Abfrage die benötigten Begriffe bestimmen. Dies ist mit Algorithmus I (Abschnitt 3.6.1) nicht möglich, da er die Begriffe eines Verbandes in einer topologischen Reihenfolge bestimmt, die überflüssige Begriffe enthält.

Die Möglichkeit, auf die Vorberechnung eines Begriffsverbandes für sehr große Kontexte zu verzichten, ist der Hauptvorteil einer schrittweisen Navigation. Gleichzeitig verhindert die Lokalität der einzelnen Operationen den direkten Zugang zu

einem Begriff: die von einem Begriff c erreichbaren Begriffe sind eine Untermenge der durch Auswahl eines Attributes $a \in \langle\langle c \rangle\rangle$ erreichbaren Begriffe. Ein Anwender ist also unter Umständen zu mehreren Schritten gezwungen, um einen bestimmten Begriff zu erreichen. Der Ergebnisbegriff einer verfeinerten Anfrage in Abschnitt 5 ist zwar ein Unterbegriff des letzten Ergebnisbegriffes, aber nicht notwendig ein direkter Unterbegriff. Diese globale Sicht erlaubt eine schnellere Navigation, die zudem durch das Angebot von sinnvollen Attributen zur Verfeinerung unterstützt wird. Sie konzentrieren die Suche des Anwenders auf die Attribute und nicht, wie bei Ganter und Bittner, auf Begriffe, deren Bedeutung manchmal schwer intuitiv zu erfassen ist.

7.2.3 Begriffsbasierte Navigation

Der in Abschnitt 5 gemachte Vorschlag, Software-Komponenten durch ihren Begriffsverband zu organisieren, ist nicht völlig neu: in [33, 34] schlagen Godin et al. ebenfalls vor, zur Organisation und Suche von Dokumenten ihren Begriffsverband zu verwenden. Wie in Abschnitt 5 bilden Dokumente und ihnen zugeordnete Schlüsselwörter einen formalen Kontext; sein Begriffsverband leitet die Suche nach Dokumenten, ohne dabei im implementierten Prototypen als Hasse-Diagramm dem Benutzer sichtbar zu sein. Ähnlich wie in dieser Arbeit bietet der Prototyp eine Benutzungsoberfläche, die eine Navigation durch Auswahl und textuelle Eingabe von Schlüsselwörtern zuläßt. Als Alternative wird eine graphische Präsentation des Verbandes diskutiert, die mit meiner *fisheye*-Sicht lokale Details gegenüber peripheren Strukturen betont.

Die Navigation orientiert sich an den Begriffen des Verbandes: ausgehend von der aktuellen Anfrage, die durch einen Begriff repräsentiert wird, werden in dem Prototypen (ohne graphische Darstellung des Verbandes) dem Benutzer direkte Ober- und Unterbegriffe zur Auswahl angeboten. Ein Unterbegriff (O_1, A_1) des aktuellen Begriffes (O, A) wird durch die (eindeutige) Attributmenge $A_1 \setminus A$ repräsentiert; analog wird ein Oberbegriff (O_2, A_2) durch $A \setminus A_2$ repräsentiert. Sowohl in [33] als auch [34] erwähnen Godin et al. die zusätzliche Möglichkeit, eine Anfrage durch Hinzunahme eines beliebigen Attributes zu verfeinern. Alle im Verlauf einer Folge von Anfragen besuchten Begriffe werden in einer Historie vermerkt, aus der sie durch Auswahl wieder zum aktuellen Begriff gemacht werden können.

Die Navigation durch Auswahl von direkten Ober- und Unterbegriffen entspricht der von Ganter und Bittner [30] vorgeschlagenen schrittweisen Navigation. Somit besteht auch der in Abschnitt 7.2.2 beschriebene Nachteil: die lokale Umgebung eines Begriffes spiegelt nur eine Untermenge der Möglichkeiten zur Verallgemeinerung und Spezialisierung wider. Die im Abschnitt 5.5 definierten *sinnvollen Attribute zur Verfeinerung* einer Anfrage erfassen dagegen die globalen Optionen zur Verfeinerung einer Anfrage. Die von Godin et al. erwähnte Möglichkeit der Erweiterung einer Anfrage um ein beliebiges Attribut ermöglicht

eine Navigation über das lokale Umfeld hinaus. Während eine lokale Navigation ein leeres Ergebnis für eine spezialisierte Anfrage weitgehend ausschließt, ist dieses bei der beschriebenen globalen Navigation jederzeit möglich. Nur eine Beschränkung auf die sinnvollen Attribute bei der Auswahl kann ein leeres Ergebnis sicher verhindern.

Sowohl das Zurückgehen zu einem vorherigen Begriff (mittels der Historie), als auch die Auswahl eines direkten Oberbegriffes stellen eine Verallgemeinerung der aktuellen Anfrage dar. Die Auswahl eines direkten Oberbegriffes führt nicht notwendig zu einer bekannten früheren Anfrage – ein Problem, das in Abschnitt 5.7 diskutiert wurde.

Bei einem direkten Vergleich dieser Arbeit mit der von Godin et al. ergibt sich vor allem eine unterschiedliche Verwendung der Begriffsverbandes: Godin et al. verwenden ihn für eine begriffs-lokale Navigation; in Abschnitt 5 wird eine globale Navigation beschrieben, die den Begriffsverband zur Effizienssteigerung einsetzt. Zusätzlich wird in Abschnitt 5.8 demonstriert, wie der Begriffsverband zur globalen Analyse der Indexierung und des Antwortverhaltens einer Sammlung eingesetzt werden kann.

In [33] haben Godin et al. die Größe von Begriffsverbänden an Hand weniger Beispiele untersucht. Sie vermuten, daß $B(\mathcal{O},\mathcal{A},\mathcal{R})/|\mathcal{O}|$ linear von der durchschnittlichen Anzahl von Attributen pro Objekt, also $|\mathcal{R}|/|\mathcal{O}|$, abhängt. Demnach besteht der folgende Zusammenhang:

$$\begin{aligned} & |B(\mathcal{O},\mathcal{A},\mathcal{R})|/|\mathcal{O}| = x_1 \times (|\mathcal{R}|/|\mathcal{O}|) + x_0 \\ \Longleftrightarrow\ & |B(\mathcal{O},\mathcal{A},\mathcal{R})| \quad\ = x_1 \times |\mathcal{R}| + x_0 \times |\mathcal{O}| \end{aligned}$$

Dieser Zusammenhang wurde durch die Versuche in Abschnitt 6 nicht bestätigt. Dort wurde ein quadratischer Zusammenhang zwischen $|B(\mathcal{O},\mathcal{A},\mathcal{R})|$ und $|\mathcal{R}|$ in über 3000 Versuchen nachgewiesen.

7.2.4 Komplexe Attribute

Viele Anwendungen und Beispiele formaler Begriffsanalyse verwenden informale oder einfach strukturierte Attribute zur Kennzeichnung von Objekten. Dies trifft nicht nur auf diese Arbeit zu, sondern zum Beispiel auch auf [91] und [32]. In *Deduction-Based Software Component Retrieval* [18] verwendet Fischer den Begriffsverband zur Suche und Strukturierung von Softwarekomponenten, die durch logische Formeln, also komplexe Attribute, indexiert sind.

Das Hauptproblem deduktionsbasierter Komponentensuche (Abschnitt 4.2.4) ist die Komplexität des Vergleichs logischer Formeln: Anfragen und Indexierung sind logische Formeln $(P,Q) \in \mathcal{P} \times \mathcal{Q}$, die aus einer Vorbedingung P und einer Nachbedingung Q bestehen. Bei der Suche muß für jede Komponente (P_i,Q_i) ein Theorembeweiser zeigen, ob Anfrage und Komponente in einer bestimmten

Beziehung $(P, Q) \sqsubseteq (P_i, Q_i)$ stehen. Die gewünschte Relation $\sqsubseteq$ zwischen Anfrage und Komponente kann dabei entsprechend der Zielsetzung einer Anfrage noch variieren.

Um diesen Flaschenhals zum Anfragezeitpunkt zu entfernen schlägt Fischer vor, vorab die Beziehung jeder Komponente (P_i, Q_i) gegen eine Reihe ausgewählter Formeln (P_j, Q_j) zu untersuchen, und das Ergebnis in einem Kontext $((\mathcal{P} \times \mathcal{Q}), (\mathcal{P} \times \mathcal{Q}), \mathcal{R})$ festzuhalten. Der zugehörige Begriffsverband dient dann der Suche und Navigation im Sinne von Abschnitt 5. Der Unterschied zur Anfrage durch Eingabe einer Formel bei klassischen deduktiven Verfahren (wie in [18] und [119]) liegt darin, daß eine Komponente nun nur noch durch ihr Verhältnis zu ausgewählten Formeln bestimmt ist, und nicht zu der von dem Benutzer eingegebenen Formel. Dieser Einschränkung steht neben dem Effizienzgewinn auch eine wesentlich vereinfachte Anfrage gegenüber, da der Anwender jetzt nur noch Formeln auswählen muß, statt sie einzugeben.

Fischers Arbeit ist ein Beispiel für die Verwendung komplexer Attribute bei der Organisation und Suche von Softwarekomponenten durch formale Begriffe. Beispiele für komplexe oder formal definierte Attribute sind auch die Arbeiten von Snelting und Tip sowie Snelting und Lindig [101, 100, 63], die allerdings den Begriffsverband nicht zur Suche von Objekten verwenden. Die Idee, deduktive Verfahren durch Klassifikation zu vereinfachen, wird ebenfalls von Penix et al. in [83] aufgegriffen.

Die im Abschnitt 2 eingeführte Begriffsanalyse betrachtet binäre Relationen. In ihrem Rahmen kann nur zwischen der An- oder Abwesenheit eines Attributes unterschieden werden; eine Verwendung mehrwertiger Attribute (Aufzählungen, natürliche Zahlen) ist unmöglich. *Mehrwertige Kontexte* [32] gestatten die Verwendung diskreter mehrwertiger Attribute, indem der Wertebereich eines mehrwertigen Attributes als mehrere, zusammenhängende, binäre Attribute kodiert wird. Die entstehenden binären Kontexte können dann mit den bekannten Algorithmen verarbeitet werden. Die Kodierung für jedes mehrwertige Attribut vermittelt eine binäre Relation. Sie bestimmt, welche binären Attribute zu dem Wert eines (mehrwertigen) Attributes gehören. Die Relation zur Kodierung eines mehrwertigen Attributes heißt *Skala*; für verschiedene Typen von Attributen existieren eine Reihe von Standard-Skalen: zum Beispiel Nominalskalen, Ordinalskalen, Inter-Ordinalskalen.

Die Verwendung mehrwertiger Kontexte führt schnell zu großen und unübersichtlichen Verbänden. *Gestufte Liniendiagramme* [32] erlauben eine übersichtliche Darstellung mehrwertiger Begriffsverbände und ihre Analyse. Das Layout eines Verbandes kann halbautomatisch unter Verwendung seiner Skalen aufgebaut werden. Jede Skala ist ein (kleiner) Kontext mit einem zugehörigen Begriffsverband. Für jede Skala wird manuell ein Layout seines Verbandes angefertigt; die verschiedenen Layouts können dann rekursiv zu einem Gesamtlayout kombiniert werden.

Toscana ist ein Werkzeug zur Analyse mehrwertigen Kontexte und Verwaltung

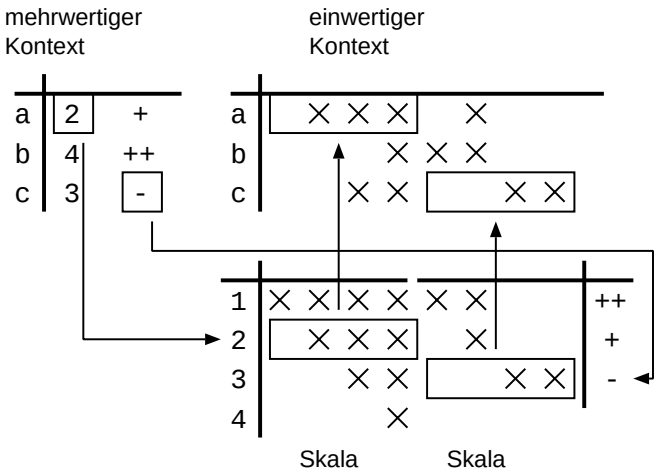

Abbildung 7.2: Das Prinzip der Übersetzung mehrwertiger in einwertige Kontexte: jedes mehrwertige Attribut wird durch eine Skala in mehrere einwertige Attribute übersetzt.

der Skalen und ihrer Layouts [111, 112]. Da es an der graphischen Darstellung gestufter Liniendiagramme orientiert ist, ist es vor allen Dingen für die (manuelle) Analyse kleiner mehrwertiger Kontexte geeignet und wird dafür auch erfolgreich eingesetzt.

Da mehrwertige Attribute durch Umkodierung mittels ihrer Skalen in einwertigen Kontexte integriert werden können, steht ihrer Verwendung bei der Organisation von Softwarekomponenten prinzipiell Nichts entgegen. Ob ihre Verwendung auch nötig ist, ist zweifelhaft, da keine der betrachteten komponentenbasierten Wiederverwendungsmethoden Unterstützung für den Vergleich mehrwertige Indexierungen fordert. Wissensbasierte und formal-logische Methoden ermöglichen konzeptionell die Verwendung mehrwertiger Attribute zur Indexierung und Anfrage.

7.3 Analysen mit formalen Begriffen

Formale Begriffsanalyse transformiert eine binäre Relation verlustfrei in einen Begriffsverband. Dieser stellt globale Eigenschaften der ursprünglichen Relation oft deutlichen dar als die relationale oder tabellarische Ausgangsdarstellung. Begriffsanalyse ähnelt damit prinzipiell einer Fourier-Transformation, die den zeitlichen Verlauf eines elektrischen Signals in ein Frequenzspektrum übersetzt, das ebenfalls neue Einsichten zuläßt. Diese Eigenschaft hat formale Begriffsanalyse zu einem wirksamen Mittel zur Analyse von Daten im Allgemeinen, und von Software im Speziellen werden lassen. In diesem Abschnitt werden exemplarisch begriffsbasierte Anwendungen zur Analyse von Software vorgestellt, die die Weite des

```
#if defined(e1) && defined(e2)
    e1
#endif
    s2
#if defined(e3)
    s3
#if defined(e4)
    s4
#endif /* inner if */
#endif /* outer if */
```

Abbildung 7.3: Variantenerzeugung mit dem C-Präprozessor

Einsatzbereiches der Begriffsanalyse über die in Abschnitt 5 vorgestellte Anwendung hinaus unterstreicht. Alle hier vorgestellten Verfahren haben für praktische Untersuchungen die C-Implementierung der Begriffsanalyse [68] verwendet, die auf Algorithmus-I beruht. Sie können damit von den algorithmischen Ergebnissen dieser Arbeit profitieren.

7.3.1 Re-Engineering von Softwarekonfigurationen

Trotz aller Bemühungen zur Standardisierung von Programmiersprachen und Betriebssystemschnittstellen (zum Beispiel in [86]) enthält Software weiterhin Plattformabhängigkeiten. Diese können nicht immer in eigenen Modulen gekapselt werden, so daß feingranulare Abhängigkeiten in Quelltexten zurückbleiben. Das am häufigsten gewählte Vorgehen setzt einen Quelltext-Präprozessor ein, der einen Quelltext filtert, bevor er an den eigentlichen Compiler übergeben wird. Er entfernt die für die aktuelle Plattform überflüssigen Teile des Quelltextes und ersetzt symbolische Konstanten durch Literale. Der bekannteste Präprozessor ist der zur Definition der Sprache C [3] gehörende C-Präprozessor `cpp`. Sein übermäßiger Einsatz führt zu einer chaotischen Vielzahl von Quelltextvarianten, die aus einem einzigen Quelltext mit seiner Hilfe generiert werden können. Snelting hat in [100] die Beziehung zwischen C-Präprozessor-Anweisungen und von ihnen beeinflußten Codesegmenten mit formalen Begriffen zum Re-Engineering von Software untersucht.

Der C-Präprozessor kontrolliert Codesegemente s_i, aufeinanderfolge Quelltextzeilen, durch logische Ausdrücke e_j, wie in Abbildung 7.3 skizziert. Obwohl die Ausdrücke relationale Vergleiche und einfache Berechnungen zulassen, beschränkt sich die Selektion von Varianten häufig darauf, die Existenz bestimmter Variablen zu prüfen; `defined`(*has_termio_h*) ist logisch erfüllt, wenn die Variable *has_termio_h* definiert ist. Sie zeigt das Vorhandensein einer bestimmten Eigen-

	Ausdrücke			
	e_1	e_2	e_3	e_4
s_1	×	×		
s_2				
s_3			×	
s_4			×	×

Abbildung 7.4: Konfigurationstabelle des Beispiels aus Abbildung 7.3

schaft[1] der aktuellen Plattform an. Ein Codesegment wird nur übersetzt, wenn sein kontrollierender `cpp`-Ausdruck logisch erfüllt ist. Da Präprozessoranweisungen geschachtelt werden dürfen, ist es oft schwer zu erkennen, welche Teile eines Quelltextes tatsächlich übersetzt werden.

Die Varianten eines Softwareproduktes sind Teil seiner makroskopischen Struktur; damit ist ihre Analyse ein Weg, Strukturen in Alt-Software aufzudecken und sie qualitativ zu bewerten. Snelting untersucht dazu die binäre Relation zwischen den kontrollierenden Ausdrücken $\mathcal{E}$ und den von ihnen kontrollierten Codesegmenten $\mathcal{S}$ durch Begriffsanalyse. In einem ersten Schritt werden dazu für jedes Codesegment seine kontrollierenden Ausdrücke bestimmt und in eine konjunktive Normalform gebracht. Anschließend wird in einer *Konfigurationstabelle* die Abhängigkeit zwischen den Codesegmenten und den Faktoren (Disjunktionen und Negationen) der konjunktiven Normalform festgehalten. Abbildung 7.4 enthält die Konfigurationstabelle des Beispiels und Abbildung 7.5 ihren Begriffsverband.

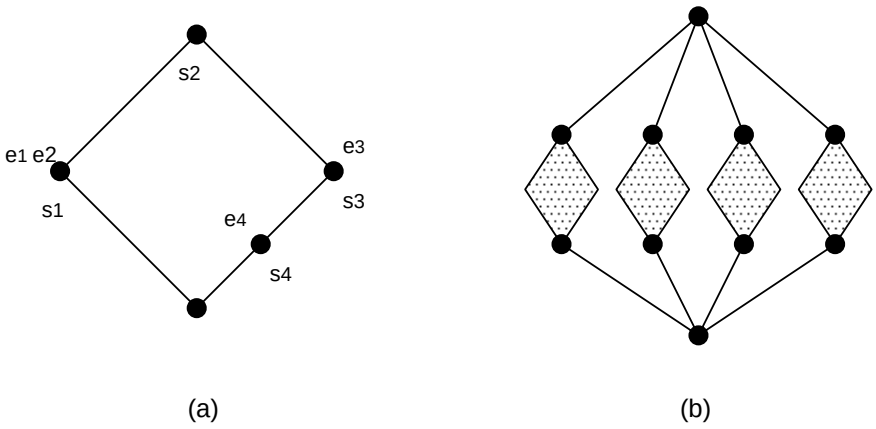

Abbildung 7.5: Begriffsverband zu Kontext in Abbildung 7.4 und Skizze eines horizontal zerlegbaren Verbandes.

Begriffe fassen Ausdrücke und von ihnen kontrollierte Codesegmente zusammen; damit die Codesegment Teil einer Konfiguration werden, müssen mindestens die Ausdrücke des Begriffes logisch erfüllt sein. Begriffe weisen auf in ei-

[1] in diesem Fall zum Beispiel die Existenz einer C-Header-Datei `termio.h`

nem Programm zusammenhängende Einheiten hin. Ihre qualitative Beurteilung kann nur durch einen Experten an Hand semantischer Kriterien erfolgen: die eine Menge von Codesegmente kontrollierenden Ausdrücke sollten einen semantischen Zusammenhang haben. Das Gegenteil weist auf eine Verletzung des Software-Engineering-Prinzipes der Trennung der Belange hin. Darüberhinaus hat Snelting festgestellt, daß die Grobstruktur des Begriffsverbandes bei gut strukturierten System aus einer horizontalen Summe von Unterverbänden besteht (Abbildung 7.5 (b)). Wird dieses Kriterium durch einzelne von $\perp$ verschiedene Infima verletzt, gibt der Begriffsverband Hinweise auf die Verursacher und Möglichkeiten zur Verbesserung der Struktur.

7.3.2 Analyse des Konfigurationsraumes

Wie im vorhergehenden Abschnitt beschrieben, erzeugt der Präprozessor `cpp` aus einem Quelltext mit `cpp`-Anweisungen eine Menge von Varianten. Diese stimmen nicht mit den von Snelting bestimmten Begriffen überein, da ein Begriff der Konfigurationstabelle eine Teilmenge der Ausdrücke beschreibt, die zur Generierung einer Variante (mit mindestens den Segmenten des Begriffes) logisch erfüllt sein müssen. Bei einer veränderten Betrachtung der Konfigurationstabelle können die möglichen Varianten eines Quelltextes direkt mit Begriffsanalyse bestimmt werden (Lindig [62]).

Eine Konfigurationstabelle ist ein Kontext $(\mathcal{S}, \mathcal{E}, \mathcal{T})$ aus Segmenten S und kontrollierenden Ausdrücken E, die in einer Relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{T}$ stehen. Jede Belegung der Ausdrücke in $\mathcal{E}$ mit (Wahrheits-) Werten $\{0, 1\}$ durch eine Konfigurationsfunktion $f : \mathcal{E} \to \{0, 1\}$ bestimmt eine Variante $f(\mathcal{S}, \mathcal{E}, \mathcal{T}) = \{s \in \mathcal{S} \mid f(e) = 1 \text{ für alle } (s, e) \in \mathcal{T}\}$. Ein Segment s wird genau dann in eine Variante aufgenommen, wenn alle seine kontrollierenden Ausdrücke logisch erfüllt (mit 1 belegt) sind. Zwei Konfigurationsfunktionen f_1 und f_2 sind äquivalent, wenn sie die gleiche Variante erzeugen: $f_1(\mathcal{S}, \mathcal{E}, \mathcal{T}) = f_2(\mathcal{S}, \mathcal{E}, \mathcal{T})$.

In einer endlichen Konfigurationstabelle teilt eine Konfigurationsfunktion f die Menge $\mathcal{E}$ in zwei disjunkte Teilmengen E und $\overline{E}$: die Ausdrücke in E werden mit 1 belegt, die in $\overline{E}$ mit 0 (Abbildung 7.6 (a) – oben). Alle Segmente S, die nur von Ausdrücken in E kontrolliert werden, bilden die von f erzeugte Variante. Dies bedeutet insbesondere, daß die Elemente in S nicht in Relation zu Ausdrücken in $\overline{E}$ stehen: $(S, \overline{E})$ bildet ein maximales kreuzfreies Rechteck in der Tabelle $\mathcal{T}$ oder anders gesagt: $(S, \overline{E})$ ist ein Begriff in der invertierten Tabelle $\overline{T}$ (Abbildung 7.6 (a) unten).

Man erhält also alle Varianten eines Quelltextes durch die Begriffe seiner invertierten Konfigurationstabelle; genauer: $[f]$ ist eine Äquivalenzklasse einer Konfigurationsfunktion f mit $f(\mathcal{S}, \mathcal{E}, \mathcal{T}) = S$ genau dann, wenn $(S, \overline{E}) \in B(\mathcal{S}, \mathcal{E}, \overline{\mathcal{T}})$ gilt. Abbildung 7.6(b) zeigt den Begriffsverband des invertierten Kontextes aus Abbildung 7.3 und damit die 6 Varianten des Beispieles aus Abbildung 7.3. Der Begriff $(\{s_2, s_3\}, \{e_1, e_2, e_4\})$ beschreibt die Variante $S = \{s_2, s_3\}$, sie wird durch

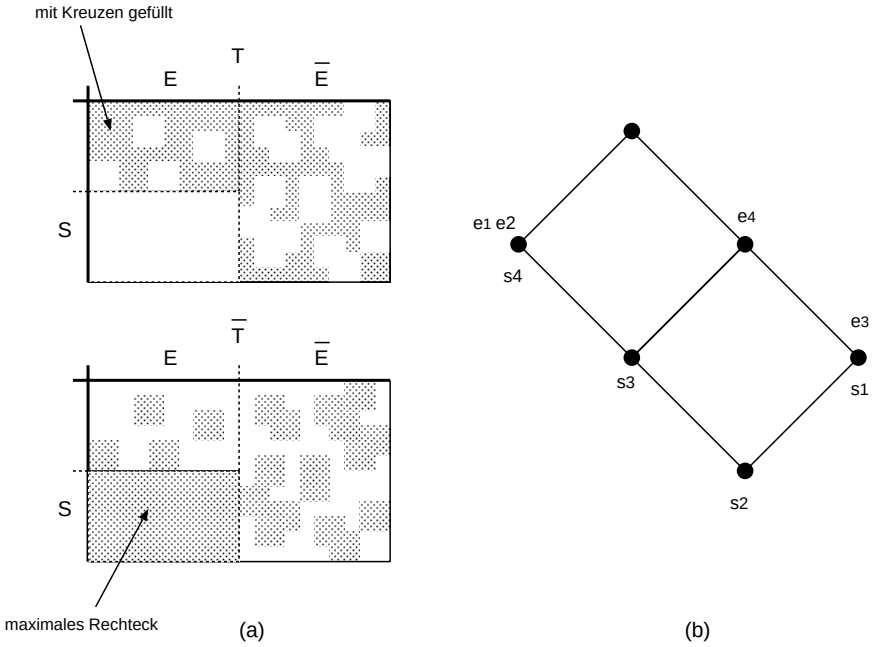

Abbildung 7.6: Varianten entsprechen maximalen Rechtecken in der invertierten Kontexttabelle. Rechts die Varianten des Beispiels in Abbildung 7.3

.

$f(e_3) = 1$ erzeugt, weil e_3 nicht in dem Begriff enthalten ist.

Eine Konfigurationstabelle kann redundante Ausdrücke enthalten; werden sie aus dem Quellcode (und der Konfigurationstabelle) entfernt, verändert sich die Menge aller erzeugbaren Varianten nicht. Redundante Ausdrücke sind im Begriffsverband der invertierten Tabelle Markierungen von $\wedge$-reduziblen Begriffen oder markieren mit anderen Ausdrücken den selben Begriff. Ihre Eliminierung entspricht der Reduktion eines Kontextes (Definition 8), die bekanntlich die Struktur eines Begriffsverbandes erhält. In dem Beispiel ist wahlweise einer der Ausdrücke e_1 oder e_2 redundant, da beide im Verband die Markierung des gleichen Begriffes sind.

Die vorgestellte Methode zur Berechnung der Varianten geht davon aus, daß alle Ausdrücke in $\mathcal{E}$ unabhängig von einander durch eine Konfigurationsfunktion mit 0 oder 1 belegt werden können. Dies stimmt nicht, wenn diese Ausdrücke statt einer Variablen Disjunktionen und Negationen enthalten, die die Erfüllbarkeit der Ausdrücke untereinander beeinflussen. Damit repräsentiert der Begriffsverband auch Varianten, die nicht erzeugt werden können. Um die exakte Menge der Varianten zu berechnen, muß die Erfüllbarkeit der durch die Begriffe vorgegebenen Terme noch geprüft werden. Alternativ kann man den Verband als Näherung der tatsächlichen Varianten auffassen. Er kann zum Beispiel als Metrik für die Komplexität einer Konfiguration dienen. Die Problematik wird ein wenig dadurch entschärft, daß Varianten durch den Einsatz des populären Werkzeuges

GNU Autoconf [27] an Hand der Existenz einzelner Eigenschaften einer Plattform gebildet werden. Dies führt zu Konjunktionen von Variablen, die durch das Verfahren exakt behandelt werden. Die alternative Strategie, Varianten für ganze Plattformen zu unterscheiden ($\texttt{sun} \vee \texttt{hpux} \vee \texttt{aix}$), führt zu den problematischen Disjunktionen.

7.3.3 Bewertung von Modulstrukturen

Die fortschreitende Modifikation von Software geht oft einher mit der Degenerierung ihrer Struktur, so daß mit jede Änderung weitere Änderungen erschwert werden. Da die finanziellen und intellektuellen Investitionen in ein Softwaresystem beträchtlich sein können, wird häufig eine Verbesserung der Struktur angestrebt, die die Software wieder wartbar macht. Der erste Schritt dahin ist eine Bestandsaufnahme der aktuellen Situation; in [63] haben Snelting und Lindig die Struktur von Software an Hand ihres Gebrauchs von Prozeduren und Variablen mit formaler Begriffsanalyse untersucht.

Softwareentwicklern stehen bei der Verwendung der Programmiersprachen Cobol oder Fortran nur wenig Mittel zur Strukturierung zur Verfügung. Die Folge ist eine weitgehende Verwendung globaler Variablen und Prozeduren. Dabei kann eine Strukturierung wegen der fehlenden syntaktischen Unterstützung durch die Programmiersprache nur noch durch Konventionen erzielt werden. Formale Begriffsanalyse hilft, die möglichweise durch Konventionen geschaffenen Strukturen aufzudecken.

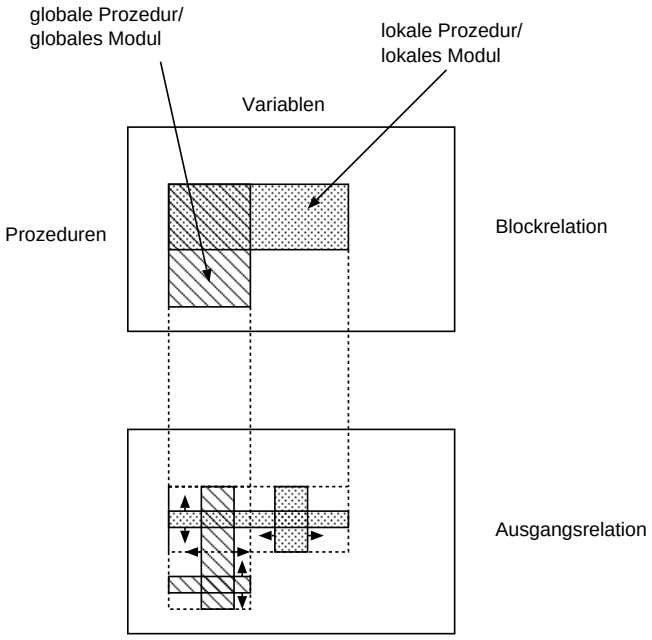

Abbildung 7.7: Re-Engineering von Sichtbarkeitsstrukturen mit Blockrelationen.

Das primäre Prinzip zur Strukturierung von Software sind geschachtelte Sichtbarkeitsbereiche für Bezeichner. Ein lokales Modul oder eine lokale Prozedur besitzt gegenüber ihrer umschließenden Umgebung mehr sichtbare Variablen: zusätzlich zu allen in der Umgebung sichtbaren Namen kommen lokale Namen hinzu. Geht man davon aus, daß in jedem Sichtbarkeitsbereich alle Prozeduren alle ihnen sichtbare Namen benutzen, entsprechen lokale Sichtbarkeitsbereiche Unterbegriffen in einem Kontext, der die Verwendung von Namen durch Prozeduren in einem Programm erfaßt (Abbildung 7.7).

Die eben getroffene Annahme über die Verwendung von Variablen durch Prozeduren ist in der Praxis natürlich nicht haltbar. Eine Kontexttabelle über die Verwendung von Prozeduren und Variablen enthält in der Regel nur kleine ausgefüllte Rechtecke. Die gegenüber der Annahme fehlenden Elemente können der Ausgangsrelation systematisch hinzugefügt werden, um so dennoch die nur durch Konventionen entstandenen Sichtbarkeitsstrukturen zu entdecken. Die mit neuen Elementen erweiterte Ursprungs-Relation ist eine Blockrelation (vergleiche Abschnitt 3.9): sie entsteht, indem in der Kontexttabelle (mehrfach) zwei Begriffe ausgedehnt werden, bis sie das gleiche Rechteck überdecken – siehe Abbildung 7.7 unten. Dazu wird ein Begriff nur horizontal, der andere nur vertikal ausgedehnt. Zu jedem Begriff einer Blockrelation gehören also zwei Begriffe in der Ursprungsrelation, die die maximale Ausdehnung des neuen Rechtecks in seiner Kontexttabelle bestimmen.

Die beiden Begriffe in der Ausgangsrelation für jeden Begriff der Blockrelation entsprechen in dem Sichtbarkeitsbereich einer Variablen, die von allen Prozeduren des Sichtbarkeitsbereiches benutzt wird und einer Prozedur, die alle Variablen des Sichtbarkeitsbereiches benutzt – eine zwar gegenüber der Ausgangsannahme abgeschwächte, aber immer noch strenge Anforderung an einen Sichtbarkeitsbereich. Bei den Untersuchungen in [63] zeigt sich dann auch, daß Programme ohne syntaktische Unterstützung für Sichtbarkeitsbereiche diese auch nicht per Konvention enthalten. Siff und Reps untersuchen auf ähnliche Weise zur Aufdeckung von Strukturen die Verwendung von Datentypen durch Funktionen in C- und C++-Programmen [97]. Der Begriffsverband dieser Verwendungsrelation wird zur Partitionierung der Funktionen in disjunkte Module herangezogen.

7.3.4 Analyse von Klassenstrukturen

In einem Begriffsverband enthält jeder Oberbegriff die Gemeinsamkeiten seiner Unterbegriffe, oder anders betrachtet, sind die Gemeinsamkeiten einer Menge von (Unter-) Begriffen in einen gemeinsamen Oberbegriff ausfaktorisiert. Dieses Prinzip der Ausfaktorisierung von Gemeinsamkeiten findet sich bei objektorientierten Programmiersprachen wieder: die Gemeinsamkeiten verschiedener Klassen von Objekten werden in einer Oberklasse zusammengefaßt. Dies ermöglicht ein uniformes Verhalten alle Klassen in Bezug auf die gemeinsamen Aspekte und erleichtert die Wartung. Nun unterliegt auch objektorientierte Software einem per-

```
class A {
    public:
    virtual int f() {return g();}
    virtual int g() {return x;}
    int x;
}
class B : public A {
    public:
    virtual int g() {return y;}
    int z;
}
class C : public B {
    public:
    virtual int f() {return g() + z;}
    int z;
}
int main () {
    A a; B b; C c;
    A *ap;
    if ( ... )
        ap = &a;
    else if ( ... )
        ap = &b;
    else
        ap = &c;
    ap->f();
}
```

Abbildung 7.8: Skizze eines C++ Programmes aus [101]

manenten Änderungsprozeß, der zu einer Verletzung dieses Prinzips führen kann, da seine Einhaltung in der Verantwortung des Programmierers liegt. Tip und Snelting leiten aus der tatsächlichen Verwendung einer Klassenhierarchie durch Begriffsanalyse eine neue, maximal faktorisierte Klassenhierarchie ab [101].

Das Hauptproblem dieser Anwendung der Begriffsanalyse liegt in der Erfassung des Ist-Zustandes, also der Verwendung einer Klassenhierarchie. Die dynamische Bindung von Bezeichnern in objektorientierten Sprachen zur Laufzeit macht eine statische Analyse im Allgemeinen unentscheidbar; durch Datenflußverfahren (*points-to-analysis*) [108] kann die Belegung objektwertiger Variablen in einem Programm aber annähernd bestimmt werden, und damit auch die Verwendung der Klassen in einem Programm.

Abbildung 7.8 zeigt ein Beispiel aus [101], das zur Demonstration der Metho-

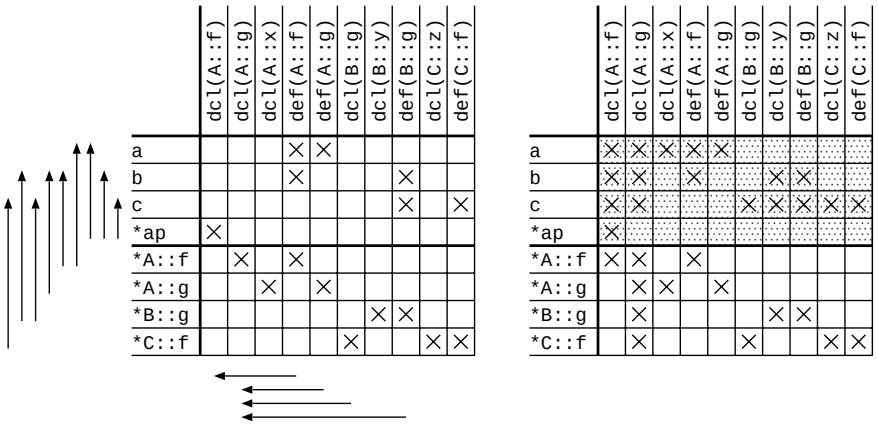

Abbildung 7.9: Konstruktion des Kontextes (aus [101].

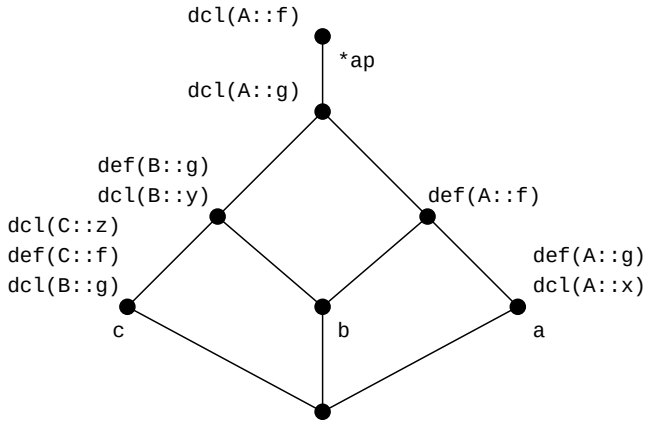

Abbildung 7.10: Verband für das Programm in Abbildung 7.8.

de von Tip und Snelting dienen soll. In dem Kontext (Abbildung 7.9 links) wird zunächst die in dem Quelltext unmittelbar ablesbare Verwendung von Methoden-Deklarationen (`dcl`) und -Definitionen (`def`) durch die Variablen `a`, `b`, `c` und `ap` festgehalten. Jede Methode (`f`, `g`) besitzt einen impliziten Parameter (*this pointer*), mit dem das aktuelle Objekt an die Methode übergeben wird. Die Verwendung dieser Zeiger auf Objekte wird unter dem Namensschema **Klasse*::*Methode* ebenfalls erfaßt: jeder *this pointer* referenziert zunächst die Definition seiner eingenen Methode und die Deklarationen der in dem Rumpf der Methode verwendeten Methoden.

Neben den unmittelbar am Quelltext ablesbaren Beziehungen ermittelt die *points-to-analysis* für jeden Zeiger Mengen möglicher Ziele: `ap` kann auf die Variablen `a`, `b` und `c` zeigen. Diese Beziehungen drücken sich in Implikationen zwischen Zeilen der Kontexttabelle aus, die in Abbildung 7.9 auf der linken Seite durch Pfeile markiert sind. Die Vererbungsbeziehung zwischen Klassen (-Definitionen und -Deklarationen) drückt sich durch Implikationen zwischen Attributen aus,

die unten in Abbildung 7.9 eingetragen sind.

Wenn der Kontext die Verwendungsbeziehung zwischen Variablen und Klassen richtig wiedergeben soll, muß er die gefundenen Implikationen respektieren. Deswegen werden in einer Fixpunktiteration Zeilen und Spalten der Tabelle entsprechend den gefundenen Implikationen ineinanderkopiert. Der damit gefundene Kontext (Abbildung 7.9 rechts) repräsentiert die tatsächliche Verwendung von Klassen durch Variablen. Für die Berechnung des maximal faktorisierten Klassenverbandes (Abbildung 7.10) brauchen die *this pointer* nicht berücksichtigt werden, so daß nur der rechts hervorgehobene Teilkontext in Abbildung 7.9 verwendet wurde.

Die maximal faktorisierte Vererbungshierarchie muß nicht notwendig zur Restrukturierung der Ausgangshierarchie verwendet werden. Sie läßt auch Anomalien und bestimmte Programmierfehler hervortreten, oder kann unsichtbar für den Programmierer nur durch einen Compiler verwendet werden.

Kapitel 8

Ergebnisse

Formale Begriffsanalyse ist eine algebraische Theorie über binäre Relationen und zu ihnen in enger Verbindung stehende Verbände. Jede binäre Relation $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$, Kontext genannt, enthält eine Menge von Begriffen $(O, A) \in 2^{\mathcal{O}} \times 2^{\mathcal{A}}$, die einen Begriffsverband bilden. Ein Begriff ist ein spezielles Paar aus einer Objektmenge O und einer Attributmenge A, die einander in der Relation $\mathcal{R}$ eindeutig bestimmen. Die Halbordnung $\leq$ unterscheidet Ober- und Unterbegriffe; die Operationen $\wedge$ und $\vee$ bestimmen den größten gemeinsamen Unterbegriff und kleinsten gemeinsamen Oberbegriff für eine Menge von Begriffen.

Der Wert der Begriffsanalyse liegt in der Fähigkeit eines Begriffsverbandes, viele Eigenschaften seiner Ausgangsrelation explizit und damit überhaupt sichtbar zu machen. Kontext und Verband sind zwei verlustfrei in einander überführbare Darstellung. Eine dritte ist die Menge der in einem Kontext gültigen Implikationen zwischen Attributen, die in dieser Arbeit aber nicht verwendet wurde.

8.1 Implementierung formaler Begriffsanalyse

Anwendungen formaler Begriffsanalyse setzen ihre Implementierung in Form von Algorithmen und Datenstrukturen voraus, da bereits kleine Kontexte eine Vielzahl von Begriffen enthalten können. Im ungünstigsten Fall wächst die Zahl der Begriffe eines Kontextes exponentiell mit der Größe des Kontextes, weswegen der Effizienz der Implementierung besondere Bedeutung zukommt. Eine Implementierung hat zwei wesentliche Aufgaben: erstens, die Berechnung aller Begriffe eines Verbandes und zweitens, die Berechnung der Verbandsstruktur der Menge aller Begriffe. Die Verbandsstruktur ist nur implizit in der Menge aller Begriffe enthalten und muß deshalb für Anwendungen explizit bestimmt werden.

Für alle Algorithmen der formalen Begriffsanalyse ist die Operation $'$ zur Bestimmung der Gemeinsamkeiten von Attribut- und Objektmengen zentral. Ihre Implementierung fußt auf dem Schnitt von Objekt- und Attributmengen, dessen Effizienz beim Design der Datenstrukturen im Vordergrund stehen muß. Die

redundante Repräsentation einer Kontexttabelle durch binäre Zeilen- *und* Spaltenvektoren garantiert eine zeit-effiziente Implementierung des Schnittes durch eine logische Und-Operation.

Ganters Algorithmus berechnet die Menge aller Begriffe eines Kontextes in einer totalen lektischen Ordnung und mit einer Komplexität von $O(|B(\mathcal{O}, \mathcal{A}, \mathcal{R})| \times |\mathcal{O}|^2 \times |\mathcal{A}|)$. Ausgehend von dem zuletzt berechneten Begriff generiert er bis zu $|\mathcal{O}|$ Kandidaten, bis der lektisch nächste Begriff gefunden ist. Dafür benötigte eine Implementierung in C etwa 200 μs auf einem 200 MHz AMD K6 Prozessor je Begriff des Ergebnisses aus circa 11 000 Begriffen, wobei dieser Zeitbedarf linear mit der Größe des Ergebnis zunimmt. Der Speicher- und Zeitbedarf erlaubt die problemlose Berechnung von Begriffsverbänden mit mehreren 10 000 Begriffen auf Personalcomputern der Pentium-Klasse.

Ganters Algorithmus kann durch Einführung eines Cache optimiert werden. Er nimmt die bei der Suche nach dem lektisch nächsten Begriff verworfenen Begriffe unter ihrer Attributmenge als Schlüssel auf. Vor der Neuberechnung eines Begriffes wird dieser zunächst in dem Cache gesucht, was im Erfolgsfall eine Operation der Komplexität $O(|\mathcal{O}| \times |\mathcal{A}|)$ einspart. Da Begriffe in einem Begriffsverband eindeutig sind, kann ein erfolgreich gefundener Begriff aus dem Cache entfernt werden.

Die Verbandsoperationen $\wedge$ und $\vee$ basieren auf der implizit in den Begriffen eines Kontextes enthaltenen Verbandsstruktur und benötigen nicht die angesprochene explizite Verbandsstruktur. Eine Implementierung von $\wedge$ und $\vee$ kann die durch Ganters Algorithmus vorgegebene lektische Sortierung der Begriffe ausnutzen. Allerdings orientiert sich diese Sortierung ausschließlich an den Objektmengen der Begriffe, so daß eine sortierte Folge von Begriffen unter Betrachtung ihrer Attributmengen unsortiert ist. Dies führt zu asymmetrischen Komplexitäten bei der Implementierung von $\wedge$ und $\vee$, da $\wedge$ mit Hilfe der Objekt- und $\vee$ mit Hilfe der Attributmengen der Argumente von $\wedge$ und $\vee$ berechnet wird: $\wedge$ kann mit $O(\log |B(\mathcal{O}, \mathcal{A}, \mathcal{R})| \times |\mathcal{O}|)$ implementiert werden, $\vee$ nur mit $O(|B(\mathcal{O}, \mathcal{A}, \mathcal{R})| \times |\mathcal{A}|)$. Abschnitt 3.8.3 diskutiert verschiedene Möglichkeiten diese Asymmetrie zu überwinden; eine versucht, durch eine nachträgliche Änderung der Ordnung von Attributen eine lektisch absteigende Ordnung von Attributmengen bei lektisch sortierten Begriffen zu erzielen. Sie erlaubt dann die effiziente Implementierung beider Operationen.

Die in den Begriffen eines Kontextes enthaltene Verbandsstruktur muß für viele Applikationen explizit bestimmt werden. Dazu hat sich eine erweiterte Version des Algorithmus von Ganter und Bittner als besonders geeignet herausgestellt: sie berechnet rekursiv vom kleinsten Begriff eines Verbandes aus dessen direkte Oberbegriffe. Damit bestimmt der Algorithmus gleichzeitig sowohl die Begriffe des Kontextes, als auch ihre Verbandsstruktur in Form der direkten Ober-/Unterbegriffsbeziehung. Ähnlich wie Ganters Algorithmus betrachtet er dazu eine Menge von bis zu $|\mathcal{O}|$ Kandidaten für die direkten Oberbegriffe des aktuellen Begriffes und besitzt auch die selbe asymptotische Komplexität

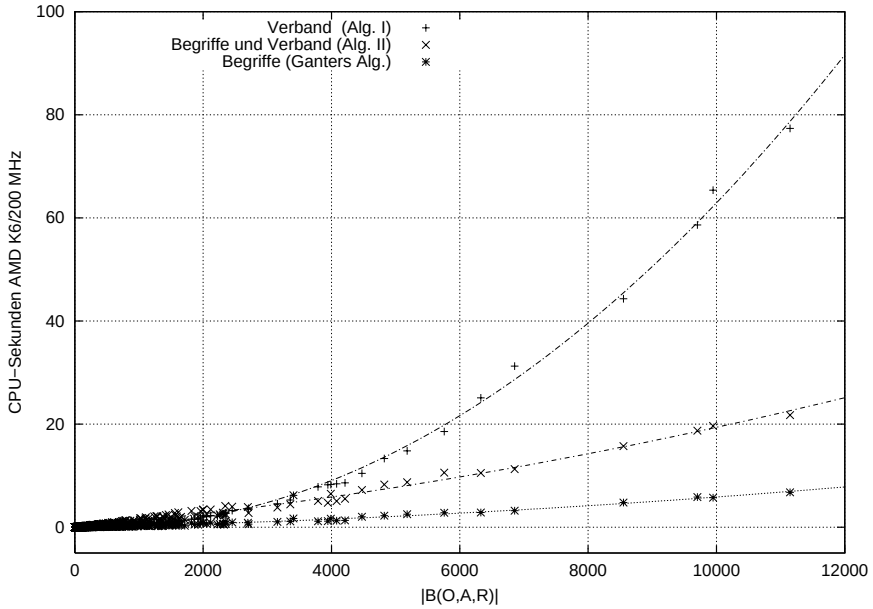

Abbildung 8.1: Zeitbedarf zur Berechnung aller Begriffe mit verschiedenen Algorithmen in Abhängigkeit der Verbandsgröße.

Algorithmus	a_2	a_1	a_0
Ganters Algorithmus	3.255×10^{-8}	0.00026	0.00101
Algorithmus-I	6.721×10^{-7}	-0.00043	0.04699
Algorithmus-II	7.764×10^{-8}	0.00116	-0.03158

Tabelle 8.1: Koeffizienten für Polynome zweiten Grades $t = a_2|B(\mathcal{O},\mathcal{A},\mathcal{R})|^2 + a_1|B(\mathcal{O},\mathcal{A},\mathcal{R})| + a_0$ zur Annäherung des Zeitbedarfs in CPU-Sekunden auf einem AMD K6 200 MHz für die Berechnung aller Begriffe und ihrer Verbandsstruktur.

$O(|B(\mathcal{O},\mathcal{A},\mathcal{R})| \times |\mathcal{O}|^2 \times |\mathcal{A}|)$. Bei praktischen Versuchen hat sich eine schwache quadratische Abhängigkeit der Laufzeit des Algorithmus von $|B(\mathcal{O},\mathcal{A},\mathcal{R})|$ gezeigt, so daß Verbände effizient bestimmt werden können.

Der Algorithmus von Ganter und Bittner wurde mit einem weiteren Algorithmus (Algorithmus-I) verglichen, der die Verbandsstruktur aus einer bereits vorhandenen Menge von Begriffen bestimmt. Seine theoretisch kubische und in der Praxis noch quasi quadratische Abhängigkeit von der Verbandsgröße macht ihn für die Berechnung großer Verbände ungeeignet; lediglich bei kleinen Verbänden erwies er sich als effizienter. Abbildung 8.1 vergleicht die mit zufällig generierten Kontexten ermittelten Laufzeiten von Ganters Algorithmus, Algorithmus-I und -II (Ganter und Bittner) in Abhängigkeit der Verbandsgröße; in Tabelle 8.1 sind die Koeffizienten der zur Approximation verwendeten Polynome angegeben.

8.2 Softwarewiederverwendung

Softwarewiederverwendung hat das Ziel, die Qualität und Produktivität der Softwareerstellung zu verbessern, um so große Softwaresysteme in einem industriellen Maßstab erstellen zu können. Die Wiederverwendung kann sich entweder auf den Prozeß der Softwareerstellung beziehen, oder auf ihre Ergebnisse. Im ersten, *generativen*, Ansatz wird der Prozeß der Softwareerstellung durch Generatoren wiederholbar. Im zweiten, *komponentenbasierten*, Ansatz werden die Ergebnisse eines Softwareprozesses abgelegt und später wiederverwendet. Die meisten Ansätze zur Wiederverwendung in der Literatur sind ablage- und quelltextorientiert; erst in jüngerer Zeit haben *Design Patterns*, als eine nicht quelltextorientierte Methode Aufmerksamkeit erfahren.

Systeme zur komponentenbasierte Softwarewiederverwendung werden in der Literatur vor allem auf Grund ihrer internen Organisation (und Implementierung) kategorisiert. Es hat sich allerdings gezeigt, daß diese Unterschiede nicht grundsätzlich zu unterscheidbaren Qualitäten bei der Suche von Komponenten führen. Der Erfolg komponentenbasierter Wiederverwendung hängt neben der Qualität der verfügbaren Komponenten von zwei Faktoren ab: Benutzerakzeptanz und Wirtschaftlichkeit. Die häufig in den Vordergrund gestellten Maße *Precision* und *Recall* sind sowohl methodisch fragwürdig, als auch von geringer Vorhersagekraft für den Erfolg eines Systemes.

Die Benutzerakzeptanz eines Systems wird maßgeblich durch seine Benutzungsoberfläche bestimmt. Obwohl die Ansprüche daran auch vom Softwareentwicklungsprozeß abhängen, sind syntaktische Unterstützung bei Eingaben, unscharfe Suchen und Hinweise zur Umformulierung einer Anfrage grundsätzlich wünschenswerte Eigenschaften. Die Suche von Komponenten sollte statt eines einmaligen Suchen-und-Findens als iterativer Prozeß aufgefaßt und unterstützt werden, an dessen Ende die gesuchte Komponente steht.

Systeme, deren Einrichtung und Pflege hohe intellektuelle und manuelle Investitionen erfordern, ohne einen augenblicklichen Gegenwert zu liefern, sind wenig aussichtsreich. Die Hauptursache für hohe Kosten sind manuelle Tätigkeiten zur Spezifikation und Organisation von Komponenten. Sie behindern insbesondere die Evolutionsfähigkeit von Komponentensammlungen, da sie häufig bei einer Restrukturierung wiederholt werden müssen. Verständliche, automatisierbare und benutzerfreundliche Komponentenablagen sind deswegen ein aussichtsreicher Weg für die praktische Wiederverwendung von Komponenten.

8.3 Begriffsbasierte Organisation von Software-Komponenten

Werden Softwarekomponenten zur Suche in einer Bibliothek durch Mengen von Attributen indexiert, so bilden die Komponenten und ihre Indexierung einen for-

malen Kontext. Die Indexierung einer Komponente, wie auch eine Anfrage zur Suche, ist eine Attributmenge, die die *Konjunktion* vorhandener oder gesuchter Eigenschaften beschreibt. Durch die Abwesenheit von Disjunktion und Negation ist die Anfrage- und Indexierungssemantik vergleichsweise einfach, ihr stehen aber Vorteile durch eine enge Beziehung zur formalen Begriffsanalyse gegenüber: Diese erlaubt eine effiziente Bearbeitung von Anfragen sowie die Analyse einer Komponentensammlung, die zur Optimierung der Anfragebearbeitung und Komponentenindexierung führt.

Eine Menge von Attributen selektiert als Anfrage alle Komponenten, die mindestens mit den Attributen der Anfrage indexiert sind. Alle Anfragen A mit dem gleichen Ergebnis $[\![A]\!]$ bilden eine Äquivalenzklasse $[A]$. Zwischen den Anfrage-Äquivalenzklassen einer Komponentensammlung und ihrem Begriffsverband besteht eine Bijektion: ein Begriff (O, A) enthält das Ergebnis O der Anfragen aus der Äquivalenzklasse $[A]$. Die Halbordnung $\leq$ der Begriffe korrespondiert mit der Spezialisierung von Anfragen. Der Begriffsverband einer Sammlung ist deswegen einem Kompilat aller möglichen Anfragen vergleichbar, das ihre Ergebnisse und Spezialisierungsbeziehungen enthält.

Anfragen werden durch die Aufnahme weiterer Attribute spezieller, das heißt, ihr Ergebnis verkleinert sich. Zur Spezialisierung einer Anfrage A sind nicht alle Attribute $\mathcal{A} \setminus A$ sinnvoll, da einige von ihnen zu einem unveränderten oder leeren Ergebnis führen. Die Menge $\langle\langle A \rangle\rangle$ der zur Verfeinerung sinnvollen Attribute führt genau zu spezielleren und nicht leeren Ergebnissen; $\langle\langle A \rangle\rangle$ ist eine Funktion der aktuellen Anfrage A und kann durch die Bijektion zwischen Anfragen und Begriffen für alle Anfragen vorberechnet werden. Alternativ zur Auswahl eines Attributes aus $\langle\langle A \rangle\rangle$ kann eine Anfrage A durch einer *Relevance-Feedback* vergleichbaren Technik interaktiv verfeinert werden: Eine Teilmenge O des aktuellen Ergebnis $[\![A_i]\!]$ bestimmt ebenfalls eine neue Anfrage A_{i+1}, mit einem spezielleren Ergebnis $[\![A_{i+1}]\!] \supseteq O$.

Da Ergebnisse und sinnvolle Attribute zur Verfeinerung eine Funktion der Anfrage sind, können sie durch die Bijektion zwischen Anfragen und Begriffen vorberechnet werden. Zum Zeitpunkt einer Anfrage ist diese dann selbst der Schlüssel für die Suche nach den vorberechneten Daten. Die gegenüber einer Implementierung ohne Verwendung des Verbandes erzielte Effizienzsteigerung ist um so größer, je mehr von einer Anfrage abhängende Daten (wie $\langle\langle A \rangle\rangle$) vorberechnet werden können. Dieses Prinzip wurde an einem interaktiven Prototypen demonstriert (Abbildung 8.2), der eine schrittweise Verfeinerung einer Anfrage durch die Auswahl von Attributen aus $\langle\langle A \rangle\rangle$ erlaubt.

Die Äquivalenz zwischen Anfrageklassen und Begriffen ermöglicht die statistische Analyse einer Komponentensammlung; insbesondere lassen sich Aussagen über die Qualität der Indexierung gewinnen. Der Vorteil im Vergleich zu anderen Verfahren liegt in seiner Vollständigkeit der erfaßten Daten und der Unabhängigkeit von Versuchen, da der Begriffsverband eine statische Analyse schon vor dem Einsatz einer Komponentensammlung erlaubt. Eine exakte Analyse verlangt die

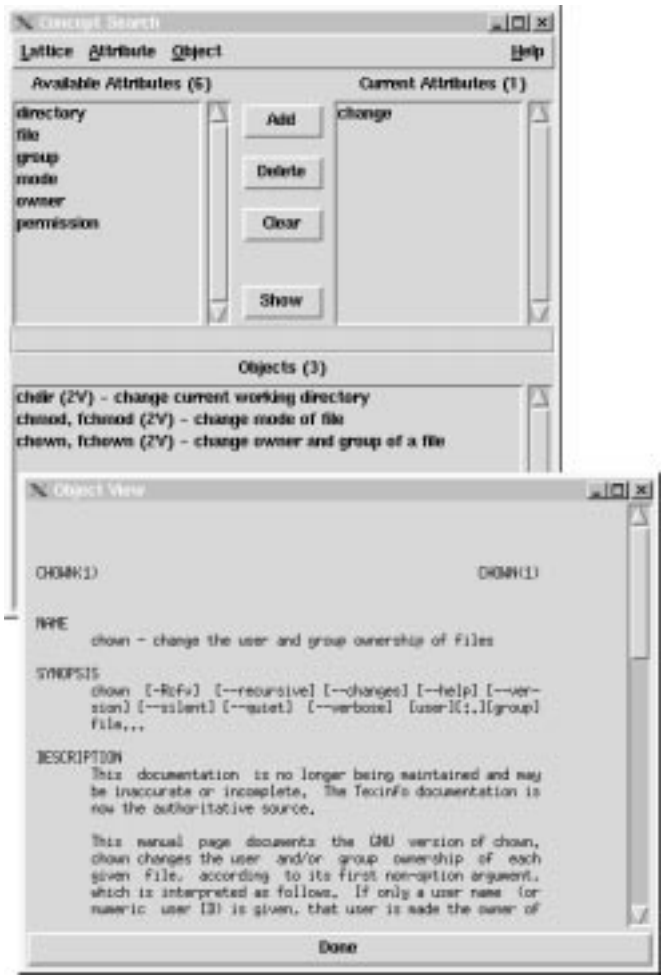

Abbildung 8.2: Prototyp einer begriffsbasierten Suche von Komponenten.

mehrfache Betrachtung jedes Begriffes und führt zu einer kombinatorischen Explosion. Deswegen wird eine leicht vereinfachte Analyse vorgeschlagen, die jeden Begriff des Verbandes nur noch einmalig betrachtet.

8.4 Größe von Begriffsverbänden

Die Größe eines Begriffsverbandes kann im ungünstigsten Fall exponentiell mit der Größe seiner Relation anwachsen. Zur Untersuchung des Zusammenhanges für praktische Anwendungen wurden in einer Testreihe über 3000 Kontexte zufällig generiert und ihre Größe untersucht. Dabei lag das Hauptaugenmerk auf Kontexteigenschaften, die eine Vorhersage der Größe ihres Verbandes erlauben. Gleichzeitig diente die Testreihe der Evaluierung verschiedener Algorithmen, deren Ergebnis bereits in der Tabelle 8.1 zusammengefaßt wurde.

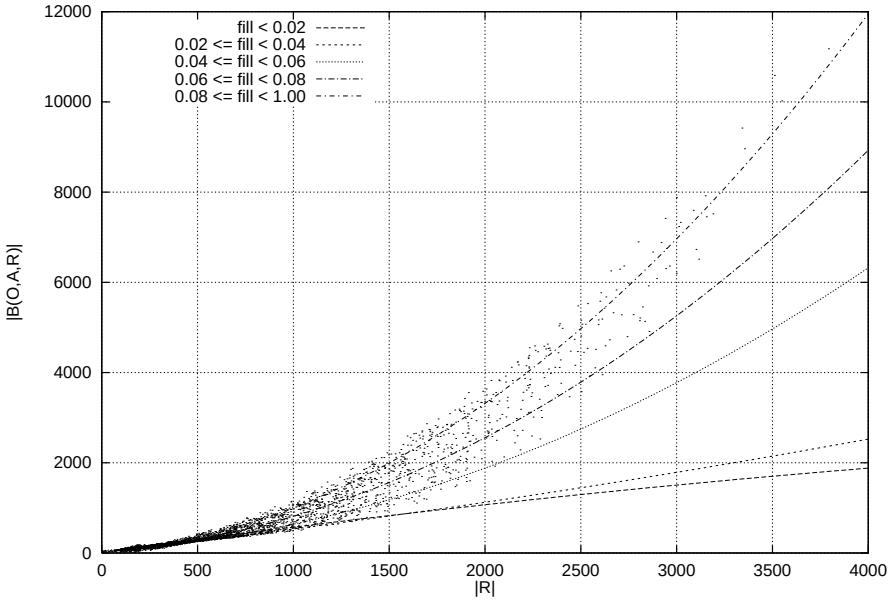

Abbildung 8.3: Größe von Begriffsverbänden in Abhängigkeit der absoluten Größe der Ausgangsrelation

$\lvert\mathcal{R}\rvert/\lvert\mathcal{O}\rvert \times \lvert\mathcal{A}\rvert$	a_2	a_1	a_0
0.00 ... 0.02	-3.0688×10^{-5}	0.58955	13.776
0.02 ... 0.04	3.6527×10^{-5}	0.48174	14.153
0.04 ... 0.06	0.00032621	0.26237	52.876
0.08 ... 0.08	0.00048146	0.29659	36.275
0.08 ... 1.00	0.00067034	0.29714	40.730

Tabelle 8.2: Koeffizienten zur Approximation der Größe $|B(\mathcal{O},\mathcal{A},\mathcal{R})|$ von Begriffsverbänden durch ein Polynom $a_2|\mathcal{R}|^2 + a_1|\mathcal{R}| + a_0$. Klasseneinteilung an Hand des Füllgrades $|\mathcal{R}|/|\mathcal{O}| \times |\mathcal{A}|$ der Kontexttabelle.

Die zu erwartende Größe eines Begriffsverbandes $|B(\mathcal{O},\mathcal{A},\mathcal{R})|$ wird vor allen Dingen durch die Größe $|\mathcal{R}|$ der Relation $\mathcal{R}$ bestimmt. In der Versuchsreihe konnte ein quadratischer Zusammenhang zwischen $|\mathcal{R}|$ und $|B(\mathcal{O},\mathcal{A},\mathcal{R})|$ beobachtet werden, wobei allerdings die Größe von Begriffsverbänden mit identischer Relationengröße noch beträchtlich schwankte. Abbildung 8.3 zeigt als Punkte die beobachteten Verbandsgrößen in Abhängigkeit von $|\mathcal{R}|$.

Zur Unterscheidung von Kontexten gleicher Größe in Hinblick auf die Größe ihrer Verbände erwies sich der relative Füllgrad $|\mathcal{R}|/|\mathcal{O}| \times |\mathcal{A}|$ ihrer Kontexttabellen als geeignet. Die in Abbildung eingezeichneten Polynome zweiten Grades nähern die Größen der Verbände von Kontexten mit unterschiedlichen relativen Füllgraden an; ihre Koeffizienten sind in Tabelle 8.2 zusammengefaßt. Die Kombination von Kontextgröße und relativen Füllgrad erlaubt damit eine Abschätzung

der zu erwartenden Größe des Begriffsverbandes eines Kontextes.

Die empirischen Untersuchungen zur Größe und Berechnung von Begriffsverbänden haben gezeigt, daß ihre Komplexität in der Praxis gutmütig ist – trotz eines theoretisch möglichen exponentiellen Wachstums in Einzelfällen. Da die Laufzeit des Algorithmus von Ganter und Bittner zur Berechnung aller Begriffe und ihrer Verbandsstruktur nur eine sehr schwache quadratische Abhängigkeit von der Anzahl der Begriffe eines Verbandes besitzt, ergibt sich in der Praxis eine erstaunlich gute Handhabung von Begriffsverbänden, die ihnen eine breite Anwendung eröffnet.

8.5 Ausblick

Diese Arbeit untersucht formale Begriffsanalyse unter algorithmischen und anwendungsorientierten Aspekten; bei ihrer Behandlung haben sich eine Reihe weiterführender Fragen ergeben, die nachfolgend noch einmal zusammengestellt sind. Sie bieten einen Ansatzpunkt für die Weiterentwicklung formaler Begriffsanalyse unter dem Gesichtspunkt ihrer Anwendung.

- Ganters Algorithmus zur Berechnung aller Begriffe kann durch einen Cache optimiert werden (Abschnitt 3.5). Die Wirksamkeit eines Caches begrenzter Größe hängt von dem Verhalten des Algorithmus bei der Generierung von Kandidaten für den nächsten Begriff ab. Ein begrenzter Cache ist um so wirksamer, je lektisch näher die generierten Kandidaten dem gesuchten nächsten Begriff sind, da sie dann anschließend gleich benötigt werden. Ähnlich wie bei der Größe von Begriffsverbänden sind hier Aussagen über das zu erwartende Verhalten hilfreicher, als Aussagen über Extrema.

- Abschnitt 6 liefert statistische Aussagen über die Größe von Begriffsverbänden und Hinweise auf die Eigenschaften von Kontexten, die die Größe maßgeblich beeinflussen. Weil nur ein begrenztes Spektrum aus Kontexten mit geringen relativen Füllgrad betrachtet wurden, bietet sich die Untersuchung des verbleibenden Spektrums an. Eine wahrscheinlichkeitstheoretische Untersuchung könnte versuchen, den Erwartungswert für die Größe von Begriffsverbänden analytisch zu bestimmen. Die bislang durchgeführten Versuche bieten sowohl Hinweise auf wichtige Parameter von Kontexten, als auch Material zur Verifikation der Ergebnisse.

- In Abschnitt 5.5.2 wurde gezeigt, daß die Berechnung der sinnvollen Attribute $\langle\langle A \rangle\rangle$ zur Verfeinerung einer Anfrage A den Begriffsverband einer Komponentensammlung nicht ohne weiteres ausnutzen kann. Möglicherweise existiert aber dennoch ein *dynamic programing* Algorithmus, der die Beziehung der Begriffe ausnutzen kann, und so eine effizientere Berechnung zuläßt.

- Die folgende Abbildung φ bildet jede Teilmenge von $A \subseteq \mathcal{A}$ auf einen Begriff ab:

$$\begin{array}{rcl} 2^{\mathcal{A}} & \rightarrow & B(\mathcal{O}, \mathcal{A}, \mathcal{R}) \\ A & \mapsto & A' \end{array}$$

  Die Menge $\mathcal{A}$ wird dadurch in Äquivalenzklassen $[A] = \{A_1 \subseteq \mathcal{A} | [\![A_1]\!] = [\![A]\!]\}$ zerlegt. Wie können diese Äquivalenzklassen effizient bestimmt werden? Die statistische Analyse von Komponentensammlungen in Abschnitt 5.8 betrachtet gerade diese Äquivalenzklassen; ihre Aussagen wären genauer, wenn auch die Mächtigkeit der einzelnen Klassen effizient bestimmt werden könnte.

Literaturverzeichnis

[1] R. Adams. An experiment in software retrieval. In Ian Sommerville and Manfred Paul, editors, *Proceedings of the Fourth European Software Engineering Conference*, pages 380–396. Lecture Notes in Computer Science Nr. 717, Springer-Verlag, September 1993.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[3] American National Standards Institute, Computer and Business Equipment Manufacturers Association Secretariat, International Organization for Standardization, and International Electrotechnical Commission. *American National Standard for programming languages: C: ANSI/ISO 9899-1990 (revision and redesignation of ANSI X3.159-1989)*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1992. Approved August 3, 1992.

[4] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

[5] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[6] Wolfgang Bibel. *Deduktion*. R. Oldenbourg, München, 1992.

[7] Garrett Birkhoff. *Lattice Theorie*. Amer. Math. Soc. Coll. Publ., 25, Providence, R.I., 1st edition, 1940.

[8] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the unified modeling language. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 344–366, Jyväskylä, Finland, 9–13 June 1997. Springer.

[9] S.-C. Chou, J.-Y. Chen, and C.-G. Chung. A behavior-based classification and retrieval technique for object-oriented specification reuse. *Software–Practice and Experience*, 26(7):815ff, July 1996.

[10] Charles L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. Technical Report CS-24-30, Dept. of Computer Science University of Waterloo, Waterloo, Canada, August 1994.

[11] Thomas H. Cormen, Charles E. Leierson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, USA, 1990.

[12] Roberto Di Cosmo. *Isomorphisms of types: from lambda-calculus to information retrieval and language design*. Progress in theoretical computer science. Birkhauser, 1995.

[13] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proc. of the Conference on Domain-Specific Languages*, pages 229–242, Santa Barbara, USA, oct 1997. USENIX.

[14] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symp. on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[15] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, 1st edition, 1983.

[16] John Dawes. *The VDM-SL Reference Guide*. Pitman, London, 1 edition, 1991.

[17] Bernd Fischer. Specification-based browsing of software component libraries. In *Proc. Automated Software Engineering '98*, Hawaii, USA, October 1998. IEEE.

[18] Bernd Fischer, Johann Schumann, and Gregor Snelting. Deduction-based software component retrieval. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction - A basis for applications*, volume 3, pages 265–292. Kluewer, 1998.

[19] Gerhard Fischer and Helga Nieper-Lemke. HELGON: Extending the retrieval reformulation paradigm. In *Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems*, Innovative Designs for Information Systems, pages 357–362, 1989.

[20] K. Fisher and J. C. Mitchell. On the relationship between classes, objects and data abstraction. *Theory and Practice of Object Systems*, 4:3–25, 1998.

[21] L. Fitzpatrick and M. Dent. Automatic feedback using past queries: Social searching? In Nicholas J. Belkin, A. Desai Narasimhalu, and Peter Willett, editors, *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR-97)*, volume 31, special issue of *SIGIR Forum*, pages 306–313, New York, July 27–31 1997. ACM Press.

[22] David Flanagan, editor. *X Toolkit Intrinsic Reference Manual for X11 Release 4 and Release 5*. O'Reilly & Associates, Inc., Sebastopol CA, 3rd edition, 1992.

[23] W. B. Frakes and B. A. Nejmeh. An information system for software reuse. In Will Tracz, editor, *Software Reuse: Emerging Technology*, pages 142–151. IEEE Computer Society Press, 1988. Originally appeared in *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse, 1987*.

[24] William B. Frakes and Thomas P. Pole. An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering*, 20(8):617–630, August 1994.

[25] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[26] Burkhard Freitag. A hypertext-based tool for large scale software reuse. In *Proc. 6th Conference on Advanced Information Systems Engineering*, Utrecht, The Netherlands, June 1994.

[27] Bob Friesenhahn. Autoconf makes for portable software — use of OS features and a freeware scripting utility solves application portability across various flavors of Unix. *BYTE Magazine*, 22(11):45–46, November 1997.

[28] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, MA, USA, 1995.

[29] Bernhard Ganter. *Beiträge zur Begriffsanalyse*, Kapitel Algorithmen zur Formalen Begriffsanalyse. BI-Wissenschaftsverlag, 1987.

[30] Bernhard Ganter and Peter Bittner. A conceptual browsing tool using lattices. Nicht veröffentlicht – private Mitteilung von Bernhard Ganter, 1998.

[31] Bernhard Ganter and Sergei O. Kuznetsov. Stepwise construction of the Dedekind-MacNeille completion. In *Proc. 6th International Conference on Conceptual Structures*, Montpellier, August 1998.

[32] Bernhard Ganter und Rudolf Wille. *Formale Begriffsanalyse – Mathematische Grundlagen*. Springer, Berlin, 1996.

[33] Robert Godin, Rokia Missaoui, and Alain April. Experimental comparison of navigation in a galois lattice with conventional information rertrieval methods. *Int. J. Man-Machine Studies*, 38:747–767, 1993.

[34] Robert Godin, Jan Gecsei, and Claude Pichet. Design of a browsing interface for information retrieval. In N. J. Belkin and C. J. van Rijsbergen, editors, *12th Int. SIGIR Conference*, pages 32–39, Cambridge, Massachusetts, June 1989. ACM, ACM Press.

[35] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Inc., 1991.

[36] Joseph A. Goguen, Timothy Winkler, Jose Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In J. A. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification Using OBJ*. Cambridge University Press, 1992.

[37] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*, pages 674–681. Addison-Wesley, 1983.

[38] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[39] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software–Practice and Experience*, 13(8):671–685, August 1983.

[40] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, N.Y., 1993.

[41] Robert J. Hall. Generalized behavior-based retrieval. In *Proceedings of the 15th International Conference on Software Engineering*. IEEE Computer Society Press, April 1993.

[42] D. Harel, H. Lachover, A. Nammad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[43] Richard Helm and Yoëlle S. Maarek. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In *Proc. OOPLSA '91*, pages 267–295, 1991.

[44] Scott Henninger. Using iterative refinement to find resusable software. *IEEE Software*, 11(5):48–59, September 1994.

[45] Scott Henninger. An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering and Methodology*, 6(2):111–140, April 1997.

[46] Ellis Horowitz, Alfons Kemper, and Balaji Narasimhan. Survey of application generators. *IEEE Software*, 2(1):40–54, January 1985.

[47] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partian, and J. Peterson. Report on the programming language haskell, version 1.2. *ACM SIGPLAN*, 27(5), May 1992.

[48] IEEE Software, jan 1997. ISSN 0740-7459. Special issue about design patterns.

[49] Jani Jaakkola and Pekka Kilpelainen. Using sgrep for querying structured text files. Technical Report C-1996-83, University of Helsinki, Department of Computer Science, November 1996.

[50] J.-J. Jeng and B. H. C. Cheng. Using formal methods to construct a software component library. In Ian Sommerville and Manfred Paul, editors, *Proceedings of the Fourth European Software Engineering Conference*, pages 397–417. Lecture Notes in Computer Science Nr. 717, Springer-Verlag, September 1993.

[51] Ralph E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 63–76, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[52] Colin Kelley and Thomas Williams. Gnuplot. `http://www.cs.dartmouth.edu/-gnuplot_info.html`, 1998. Frei im Quelltext verfügbare Software zur Darstellung von Daten in Diagrammen, enthält Funktionen zur Approximation von Daten durch Funktionen.

[53] Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C.* Carl Hanser, Prentice–Hall International, 2. Ausgabe, 1989. Siehe auch [3].

[54] Donald E. Knuth. *The Art of Computer Programming*, volume 3 of *Computer Science and Information Processing.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1973.

[55] Donald E. Knuth. *The Art of Computer Programming*, volume 1 of *Computer Science and Information Processing.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1973.

[56] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[57] Sergei O. Kuznetsov. Learning of conceptual graphs from positive and negative examples. Private Mitteilung. Der Algorithmus wurde auf der *International Conference on Knowledge, Logic, Information* am 18.-20. Februar 1998 in Darmstadt vorgestellt., Februar 1998.

[58] Xavier Leroy. Objective Caml. Frei verfügbare Implementierung für eine Vielzahl von Plattformen. `http://pauillac.inria.fr/ocaml/`.

[59] Xavier Leroy. Polymorphism by name for references and continuations. In *Proc. 20th Symp. Principles of Programming Languages*, pages 220–231. ACM Press, 1993.

[60] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd symp. Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[61] H.-C. Liao, M.-F. Chen, and F.-J. Wang. A domain-independent software reuse framework based on a hierarchical thesaurus. *Software – Practice and Experience*, 8(28):799–818, July 1198.

[62] Christian Lindig. Analyse von Softwarevarianten. Informatik-Bericht 98-04, TU Braunschweig, Institut für Programmiersprachen und Informationssysteme, Abt. Softwaretechnologie, D-38106 Braunschweig, Januar 1998.

[63] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*. IEEE Computer Society Press, 1997.

[64] Christian Lindig. Inkrementelle, rückgekoppelte Suche in Software-Bibliotheken. Informatik-Bericht 94-07, Technische Universität Braunschweig, Institut für Programmiersprachen und Informationssysteme, Abteilung Softwaretechnologie, Gaußstraße 17, D-38106 Braunschweig, November 1994.

[65] Christian Lindig. Concept-based component retrieval. In Jana Köhler, Fausto Giunchiglia, Cordell Green, and Christoph Walther, editors, *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pages 21–25, Montreál, August 1995.

[66] Christian Lindig. Komponentensuche mit Begriffen. In *Softwaretechnik '95*, Braunschweig, October 1995.

[67] Christian Lindig. TkConcept. `http://www.cs.tu-bs.de/softech/tkconcept/`, 1995. Erweiterung für die Skriptsprache Tcl/Tk. Sie stellt Kommandos für die Berechnung von Begriffsverbänden und ihre persistente Ablage in Tcl/Tk bereit. Implementierung in C.

[68] Christian Lindig. Concepts. `ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/misc/concepts-0.3d.tar.gz`, 1997. Frei verfügbare und portable Implementierung einer Begriffsanalyse in C.

[69] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, SE-17(8):800–813, 1991.

[70] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[71] Ali Mili, Rym Mili, and R. T. Mittermeir. A survey of software storage and retrieval. *Annals of Software Engenieering*, 5, 1998. Baltzer Science Publishers.

[72] Hafdeh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[73] Rym Mili, Ali Mili, and Roland Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering*, 23(7):445–460, July 1997.

[74] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[75] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[76] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 365–458. North-Holland, New York, N.Y., 1990.

[77] Amy Moorman Zaremski and Jeanette M. Wing. Signature matching: A key to reuse. In David Notkin, editor, *Proc. of the 1st ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pages 182–190, Los Angeles, CA, December 1993. ACM, ACM Press.

[78] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, 1996.

[79] Brad A. Myers. User-interface tools: Introduction and survey. *IEEE Software*, 6(1):15–23, January 1989.

[80] Eduardo Ostertag, James Hendler, Rubén Prieto Diaz, and Christine Braun. Computing similarity in a reuse library system: An AI-based approach. *acm Transactions of Software Engineering and Methodology*, 1(3):205–228, July 1992.

[81] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, Reading, MA, 4th edition, 1994.

[82] Santanu Paul and Atul Prakash. A frame work for source code search using program patterns. *IEEE Transactions of Software Engineering*, 20(5):463–475, June 1994.

[83] John Penix and Perry Alexander. Design representation for automating software component reuse. In *Proceedings of the first international workshop on Knowledge-Based systems for the (re)Use of Program libraries*, November 1995.

[84] John Penix and Perry Alexander. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, pages 535–542. Knowledge Systems Institute, June 1997.

[85] Andy Podgurski and Lynn Pierce. Retrieving reusable software by sampling behaviour. *ACM Transactions on Software Engineering and Methodology*, 2(3):286–303, July 1993.

[86] *System Application Program Interface (API) [C Language].* Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1990.

[87] Rubén Prieto-Diáz. Implementing Faceted Classification for Software Reuse. In *Proc. of the 12th ICSE*, pages 300–304, March 1990.

[88] Rubén Prieto-Diáz. Classifying software for reuse. *IEEE Software*, 4(1):6–16, January 1987.

[89] Rubén Prieto-Diáz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):89–97, May 1991.

[90] Rubén Prieto-Diáz. Status report: Software reusability. *IEEE Software*, 10(3):61–6, May 1993.

[91] Uta Priss. A graphical interface for document retrieval based on formal concept analysis. In Eugene Santos, editor, *Proceedings of the 8th Midwest Artificial Intelligence and Cognitive Science Conference*, Dayton, Ohio, March 1997. AAAI Press, Menlo Park, CA.

[92] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, January 1991.

[93] E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries. In Koichi Furukawa, editor, *Logic Programming: Proc. of the 8th Int. Conference*, pages 173–187, Paris, France, June 1991. MIT Press.

[94] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. *J. Amer. Soc. for Information Sci.*, 41(4):288–297, 1990.

[95] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Tokio, 1983.

[96] Dieter Schütt. Abschätzung für die Anzahl der Begriffe von Kontexten. Diplomarbeit, Technische Hochschule Darmstadt, AG1: Allgemeine Algebra, Fachbereich Mathematik, Darmstadt, Mai 1987.

[97] Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *Proc. of. the Internation Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, October 1997.

[98] Martin Skorsky. *Endliche Verbände – Diagramme und Eigenschaften.* Dissertation, TH Darmstadt, Darmstadt, 1992.

[99] ACM. The ACM electronic guide to computing literature. CD ROM, ISSN 1089-5310, Association for Computing Machinery, 1998.

[100] Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1986.

[101] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proc. SIGSOFT Symposium on Foundations of Software Engineering*. ACM, 1998.

[102] Gregor Snelting. Paul Feyerabend und die Softwaretechnologie. *Informatik-Spektrum*, 21(5):273–276, Oktober 1998.

[103] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, New York, 1989.

[104] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, 6th (version 18) edition, February 1988.

[105] Guy L. Steele. *Common Lisp*. Digital Press, Burlington, 1985.

[106] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 3rd edition, 1997.

[107] Uta Stürmer. *Vergleichende Evaluierung verschiedener Interaktionsparadigmen für Informationsretrieval*. GMD-Studien, Nr. 277. GMD Formschungszentrum Informationstechnik, Sankt Augustin, 1995. ISBN 3-88457-277-6, auch 1995 als Diplomarbeit an der TH Darmstadt veröffentlicht.

[108] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[109] Laura A. Valaer and Robert G. Babb, II. Choosing a user interface development tool. *IEEE Software*, 14(4):29–39, July/August 1997.

[110] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[111] F. Vogt and R. Wille. TOSCANA — a graphical tool for analyzing and exploring data. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 226–233. DIMACS, Springer-Verlag, October 1994. ISBN 3-540-58950-3.

[112] Frank Vogt. *Formale Begriffsanalyse mit C++*. Springer, Berlin, 1996.

[113] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly, Sebastopol, 1991.

[114] Martin Wirsing. Algebraic specification. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 675–788. The MIT Press, New York, N.Y., 1990.

[115] Nikolaus Wirth. *Programming in Modula-2*. Springer, Berlin, 4th edition, 1988.

[116] Murray Wood and Ian Sommerville. An information retrieval system for software components. *SIGIR Forum*, 22(3):11–25, 1988.

[117] William Woods and James Schmolze. The KL-ONE family. *Computers and Mathematics with Applications, Special Issue on Semantic Networks in Artificial Intelligence*, 1991. Also available as Technical Report TR-20-90, Aiken Computation Laboratory, Harvard University.

[118] Edward Yourdon. *Decline and Fall of the American Programmer*. Yourdon Press, Englewood Cliffs, 1992.

[119] A. Moormann Zaremski and J. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.

[120] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, April 1995.

Lebenslauf

Name	Christian Lindig
Geburtsdatum	20. Januar 1967
Geburtsort	Bad Bevensen
Eltern	Dr. Klaus Lindig und Theresia Lindig, geb. Dellbrügge
1973–1976	Grundschule Bevensen
1976–1977	Grundschule Ahlden/Aller
1977–1979	Orientierungsstufe Schwarmstedt
1979–1986	Gymnasium Walsrode, Abschluß mit allgemeiner Hochschulreife
1986–1988	Dienst bei der Bundeswehr, zuletzt als Fähnrich
1.10.1988–29.9.1993	Studium der Informatik an der TU Branschweig Abschluß als Diplom-Informatiker
1.10.1993–30.6.1999	Wissenschaftlicher Mitarbeiter am Institut für Programmiersprachen und Informationssysteme, Abteilung Software-Technologie bei Prof. Dr. Snelting, Anfertigung der Dissertation
seit 15.7.1999	Gärtner Datensysteme GbR, Braunschweig