Formale Begriffsanalyse im Software Engineering

Christian Lindig, Gregor Snelting
TU Braunschweig
Abteilung Softwaretechnologie

Zusammenfassung. *Reuse* und *Reengineering* sind aktuelle Probleme im Software-Engineering. Reuse zielt auf die Wiederverwendung von Software-Komponenten oder -Schablonen aus einer Bibliothek; dazu ist es notwendig, effektive Verfahren zur Komponentensuche bereitzustellen. Reengineering befaßt sich mit der Rekonstruktion von Systemstrukturen aus alter Software; Ziel ist es, Altsoftware so zu transformieren, daß sie auch weiterhin lebensfähig bleibt.

Wir präsentieren zwei Werkzeuge zur Unterstützung von Reuse und Reengineering, die auf formaler Begriffsanalyse basieren. NORA/RECS rekonstruiert Konfigurationsstrukturen aus Quelltexten und stellt sie graphisch dar. Man erhält bemerkenswerte Einsichten in die Struktur des Konfigurationsraums: alle Abhängigkeiten zwischen Konfigurationspfaden werden dargestellt, und unerwünschte Interferenzen zwischen Konfigurationen werden aufgedeckt. NORA/FOCS bietet interaktive, inkrementelle Unterstützung zur Suche in Software-Komponentenbibliotheken, in denen die einzelnen Komponenten mit Attributen indiziert sind.

1 Einleitung: $(Re)^3$

Wiederverwendung und *Altlastensanierung* sind hochaktuelle Themen im Software Engineering. Früher befaßte sich Software Engineering allein mit dem Problem, *neue* Programme nach einer gegebenen Spezifikation zu entwickeln, wobei Qualitätskriterien wie Korrektheit, Effizienz, Robustheit usw. erfüllt werden müssen. Heute wird dies völlig anders gesehen, denn die klassische statische Sichtweise, daß Software einmal entwickelt und nach einer gewissen Lebensdauer ausrangiert wird, hat zu immensen Wartungskosten geführt. Allgemein zielt Software Engineering heute auf eine bessere *Evolutionsfähigkeit* von Software – möglichst schon beim Entwurf soll auf zukünftige Änderungen Rücksicht genommen werden (Antizipation des Wandels). Aus Rationalisierungs- und Qualitätsgründen sollen ferner vorgefertigte Komponenten und Schablonen eingesetzt werden, so wie dies in anderen Ingenieursbereichen längst üblich ist.

Ganz massiv ist in diesem Zusammenhang das Problem der Altsoftware in den Blickpunkt gerückt: viele (fast) unersetzliche Systeme sind schon 20 Jahre alt und wurden nicht nach modernen softwaretechnischen Kriterien entwickelt. Durch fortwährende Fehlerreparaturen und Funktionserweiterungen unterlagen sie zudem einem andauernden Strukturverlust (Gesetz der zunehmenden Entropie). Nur durch *Software-Geriatrie* [Pa94] können alte Systeme auch weiterhin lebensfähig erhalten werden.

Als Reaktion auf die genannten Problemfelder entwickelten sich in den letzten Jahren die „$(Re)^3$–Technologien“: Reuse, Reengineering und Restructuring.

Software-Reuse zielt auf die Wiederverwendung von Komponenten oder Schablonen. Schon vor 25 Jahren wurde die bis heute nicht erreichte Zielvorstellung formuliert: Software-Komponenten sollen aus einem Katalog anhand ihrer Beschreibungen auswählbar und zusammensetzbar sein, ganz so wie dies für elektronische Bauteile und Chips möglich ist. In der Tat sind ja Bibliotheken etwa mit mathematischen Funktionen oder Betriebssystemroutinen nichts Neues – hier ist aber das Zusammensetzen trivial, da derartige Komponenten eine festumrissene Funktionalität haben. Wesentlich lohnender, aber auch schwieriger wird es, wenn Schablonen, Entwürfe oder abstrakte Konzepte wiederverwendet werden sollen.

Ziel ist es heute, allgemeine Systemstrukturen und Lösungsschemata für einen Anwendungsbereich in Form sog. *Referenzarchitekturen* zu beschreiben, die dann für konkrete Systeme erweitert und instantiiert werden können. Ein anderes Problem ist das *Software-Komponentenretrieval*: geeignete Komponenten oder Schablonen müssen anhand von möglichst einfachen Beschreibungen aus einer Bibliothek ausgewählt werden; dabei wird gleichzeitig hohe Trefferquote (d.h. relevante Komponenten werden auch gefunden) und hohe Präzision (d.h. die gefundenen Komponenten sind relevant) angestrebt.

Software-Reengineering, manchmal emphatisch „Programmverstehen“ genannt, zielt auf die Rekonstruktion von Abstraktionen aus vorhandenen Quelltexten. In der Praxis ist man oft mit großen Altsystemen konfrontiert, deren Dokumentation unvollständig oder nicht vorhanden ist; die ursprünglichen Entwickler sind in der Regel nicht mehr greifbar. Die Software ist dann vom *Entropietod* bedroht: sie läuft zwar, kann aber – da intern das Chaos herrscht – nicht mehr verstanden, geschweige denn geändert werden. Eine komplette Neuentwicklung kann man sich aber meist auch nicht leisten, denn die enormen Investitionen in die Altsoftware (dazu gehören auch Hardware, Mitarbeiterschulung und andere Infrastruktur) kann man nicht einfach abschreiben. Mithin müssen Informationen wie Kontroll- und Datenflußabhängigkeiten, Modulkopplung, Systemarchitektur, Struktur des Konfigurationsraums aus dem Quelltext eruiert werden. Es gibt sogar Versuche, nachträglich die Spezifikation aus dem Code zu rekonstruieren (Retroengineering) – sicher nützlich, wenn der Verfall schon so fortgeschritten ist, daß man nicht mehr weiß, was die Software überhaupt tut.

Software-Restrukturierung zielt auf eine Transformation und Reorganisation des Codes, um das System wieder lebensfähig zu machen. Nach erfolgreichem Reengineering kann man daran gehen, Software-Engineering-Grundprinzipien wie Antizipation des Wandels und Modularisierung zu realisieren, wobei Kriterien wie hohe Kohäsion und schwache Kopplung von Modulen berücksichtigt werden müssen. Dabei werden Codestücke aufgebrochen und neu zu Prozeduren zusammengefaßt, Prozeduren werden zu Modulen gekapselt, für Prozeduren und Module werden exakte Schnittstellen definiert usw. Restrukturierung ist weitgehend manuelle Arbeit; es gibt erst wenige Verfahren zur vollautomatischen Restrukturierung. Ein klassisches Beispiel ist die Elimination von GOTOs: GOTOs erhöhen die Codeentropie und sind deshalb verpönt, alte Software in alten Sprachen enthält aber massenweise GOTOs. Bekanntlich läßt sich jedes Programm in strukturierte Form verwandeln (d.h. es werden geschachtelte WHILE-Schleifen und IF-Anweisungen statt GOTOs benutzt). Mit graphtheoretischen Verfahren kann der Datenflußgraph und damit der Quelltext entsprechend transformiert werden.

In diesem Artikel wenden wir formale Begriffsanalyse für $(Re)^3$ an. Wir präsentieren Werkzeuge, die den Begriffsverband nutzbar machen und zeigen

- zum Thema **Reuse**: Die Strukturierung von indizierten Sammlungen durch einen Begriffsverband erlaubt eine sehr effiziente, inkrementelle Komponentensuche mit Rückkopplung;

- zum Thema **Reengineering**: Die Analyse von Konfigurationsräumen mit Begriffsanalyse ermöglicht die Darstellung aller Abhängigkeiten und Interferenzen zwischen Konfigurationspfaden;
- zum Thema **Restrukturierung**: Die Reorganisation von Konfigurationsräumen durch Verbandszerlegung führt zu besserer Kohäsion.

Die vorgestellten Werkzeuge sind Teil der experimentellen Softwareentwicklungsumgebung NORA [SGS91, SFGKZ94]. NORA zielt auf die Nutzbarmachung neuer theoretischer Ergebnisse und Verfahren in Softwarewerkzeugen. Neben den auf Begriffsanalyse basierenden Werkzeugen bietet NORA zur Zeit

- deduktionsbasiertes Software-Komponentenretrieval: Komponenten können anhand einer formalen Spezifikation mit Vor- und Nachbedingungen gesucht werden; dazu werden Resolutionsbeweiser und Model Checker eingesetzt [FKSt95, FKSn95];
- Konfigurationsmanagement mit Feature-Logik: Feature-Logik bietet einen einheitlichen Formalismus zur Beschreibung von Versionen und Varianten von Softwarekomponenten, und unterstützt die konsistente Konfigurierung großer Systeme [ZS95, Ze95];
- semantikbasierten Architekturentwurf: Ein $\lambda\delta$–Kalkül mit *Dependent Types* erlaubt die Beschreibung von Referenzarchitekturen; fertige Systeme werden durch Ausführen von sog. Modulprogrammen montiert [Gr95a, Gr95b].

NORA ist ein offenes System, das keine vollständige Unterstützung des Software-Entwicklungszyklus anstrebt, sondern für solche ausgewählten Problembereiche Unterstützung bietet, für die vielversprechende neue theoretischer Resultate vorhanden sind. NORA hat insofern auch einen Technologietransferaspekt und soll die Kluft zwischen Grundlagenforschung und praktischer Softwaretechnologie verringern helfen.

2 Begriffsbasierte Komponentensuche

In diesem ersten Hauptteil der Arbeit wollen wir zeigen, wie Begriffsanalyse eingesetzt werden kann, um effizient Software-Komponenten in einer Bibliothek wiederzufinden. Wir gehen dabei davon aus, daß für jede Komponente eine Indexierung mit *Schlüsselworten* gegeben ist. Der Benutzer kann interaktiv Schlüsselworte angeben, zu denen dann passende Komponenten angezeigt werden. Schlüsselworte können auch inkrementell hinzugefügt oder gelöscht werden. Begriffsanalyse wird angewendet, um die möglichen Suchpfade und Entscheidungsmöglichkeiten quasi zu „compilieren“. Ziel ist also nicht eine direkte Nutzung des Verbandes; vielmehr dient der Verband im Hintergrund als effizienzsteigernde Datenstruktur. Anstatt also die Bibliothek sequentiell nach passenden Komponenten zu durchsuchen (was bei großen Bibliotheken sehr lange dauern kann), liefert der Verband in „Nullzeit“ die richtigen Antworten.

Der Verband wird aus der Indexierung einmal berechnet. Da der Inhalt der Komponenten keine Rolle spielt, können genausogut Programmstücke, Dokumente, Entwürfe usw. bearbeitet werden. Durch eine Indexierung der Grundkomponenten mit Schlüsselwörtern entsteht so automatisch ein syntaxorientiertes Retrievalsystem für Softwarekomponenten [Li95a, Li95b].

Sys.-Call	Kurzbeschreibung	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

Abbildung 1: Unix-System-Calls mit Kurzbeschreibung und Indexierung

2.1 Beispiel: Unix Dokumentation

Unix-Systeme besitzen traditionell eine Online-Dokumentation ihrer Programme, Systemaufrufe, Bibliotheken und Dateiformate, die zusammen etwa 1 800 Dokumente umfaßt. Ein kleiner Ausschnitt aus der Dokumentation der Betriebssystemaufrufe (System Calls) soll demonstrieren, wie Dokumente, stellvertretend für ihren Programmcode, indexiert und gesucht werden können. Abbildung 1 zeigt die Namen von 12 Unix System-Calls zusammen mit einer einzeiligen Beschreibung aus ihrer Dokumentation [Sun90] und einer Menge von Schlüsselwörtern als Indexierung.

Die Schlüsselwörter ergeben sich aus den Beschreibungen der Komponenten und bilden ihre Indexierung, die die Grundlage für die spätere Suche ist. Die Indexierung der einzelnen Komponenten ist zunächst völlig unabhängig voneinander, sollte aber praktischerweise auf einem Grundwortschatz von Schlüsselwörtern aufbauen. In Abbildung 2 sind die Komponenten und ihre Indexierung nochmals in der bekannten Form der Kontexttabelle wiedergegeben. Die Relation $\mathcal{R}$ zwischen Komponenten $\mathcal{O}$ (Objekten) und Schlüsselwörtern $\mathcal{A}$ (Attributen) wird als formaler Kontext $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ aufgefaßt.

2.2 Suche von Komponenten

Komponenten werden durch die Angabe von Schlüsselwörtern gesucht: eine *Anfrage* $A \subseteq \mathcal{A}$ ist eine Menge von Schlüsselwörtern. Eine Komponente $o \in \mathcal{O}$ *erfüllt* eine Anfrage, wenn sie mindestens mit den in der Anfrage geforderten Attributen indexiert ist: $\{o\}' \supseteq A$. Alle Komponenten, die eine Anfrage erfüllen, bilden zusammen das *Ergebnis* $[\![A]\!]$ einer Anfrage: $[\![A]\!] := \{o \in \mathcal{O} \mid \{o\}' \supseteq A\}$.

Abbildung 3 zeigt vier Beispiele für Anfragen und die zugehörigen Ergebnisse. Das erste Beispiel selektiert alle Komponenten, die mindestens mit *change* indexiert sind. Im zweiten Beispiel wird diese Anfrage dann verschärft, indem zusätzlich eine Indexierung mit *file* gefordert wird. Das zweite Ergebnis ist eine Untermenge des ersten, die Anfrage eine Obermenge der ersten Anfrage. Eine leere Anfrage selektiert alle Dokumente der Sammlung und bei unvereinbaren Attributen wie im vierten Beispiel ergibt sich ein leeres Ergebnis. Diese Situation

	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`						×								×						×

Abbildung 2: Kontext für Unix-System-Calls

Beispiel	Anfrage A	Ergebnis $[\![A]\!]$
1	*change*	`chdir chmod chown`
2	*change file*	`chmod chown`
3	$\emptyset$	$\mathcal{O}$
4	*change new*	$\emptyset$

Abbildung 3: Beispiele für die Suche von Komponenten

soll natürlich möglichst vermieden werden, da sie den Mißerfolg einer Suche dokumentiert.

Die Berechnung eines Ergebnisses kann natürlich an Hand der Kontexttabelle vorgenommen werden – sie wird dazu linear durchlaufen und alle die Anfrage erfüllenden Komponenten aufgesammelt. Eleganter und effizienter ist es, das Ergebnis mit Hilfe des Begriffsverbandes $\mathcal{B}(\mathcal{O}, \mathcal{A}, \mathcal{R})$ zu bestimmen. Abbildung 4 zeigt den Begriffsverband des Beispiels, wobei jeder Begriff b wie üblich mit dem Attribut a und dem Objekt o gekennzeichnet wurde, wenn $\mu(a) = b$, bzw. $\gamma(o) = b$ gilt.

Welche Objekte bilden nun das Ergebnis $[\![A]\!]$ einer Anfrage A? Jedes Attribut $a \in A$ wird durch einen Begriff $\mu(a)$ in den Begriffsverband eingeführt und von dort an Unterbegriffe „vererbt“. Alle Objekte des größten gemeinsamen Unterbegriffs (Infimum) dieser Begriffe enthalten die Attribute aus A. Da das Infimum gerade der größte Begriff mit dieser Eigenschaft ist, existiert keine größere Menge von Objekten, die die Anfrage erfüllen. Also bilden die Objekte des Infimums das Ergebnis: $[\![A]\!] = O_1$ mit $(O_1, A_1) = \bigwedge_{a \in A} \mu(a)$. Algorithmisch wird die Berechnung des Infimums nach dem Hauptsatz der Begriffsanalyse auf den Schnitt der beteiligten Objektmengen zurückgeführt: $[\![A]\!] = \bigcap_{a \in A} O_a$ mit $(O_a, A_a) = \mu(a)$ – die Schnittmenge ist das Ergebnis.

Eine Anfrage A selektiert eine Menge von Komponenten $[\![A]\!]$, die mindestens die Attribute aus A besitzen. Darüberhinaus können die Objekte aus $[\![A]\!]$ noch weitere gemeinsame Attribute besitzen, die nicht in A enthalten sind; der Benutzer könnte sie zusätzlich fordern, ohne das Ergebnis zu beeinflussen. Zum Beispiel besitzt die Anfrage $A = \{\mathit{file}, \mathit{create}\}$ aus dem System-Call-Kontext das Ergebnis $[\![A]\!] = \{\mathit{creat}\}$; *creat* besitzt aber auch noch das Attribut *new*, so daß $[\![\mathit{file}, \mathit{create}, \mathit{new}]\!] = \{\mathit{creat}\}$ ebenfalls gilt. Die vollständige Menge aller Attribute,

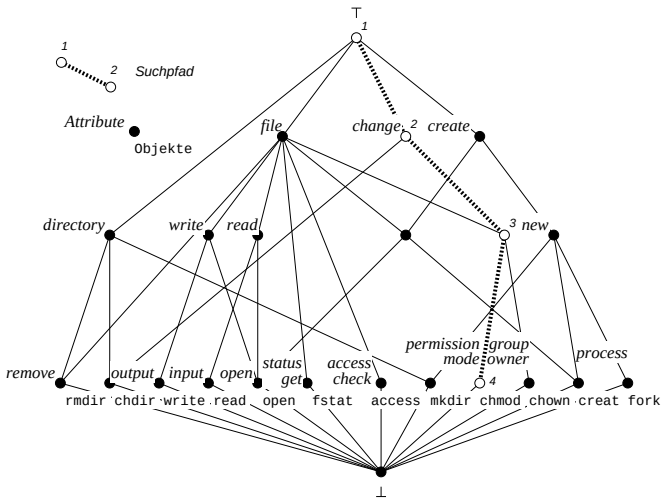


Abbildung 4: Begriffsverband der System-Call Kontextes

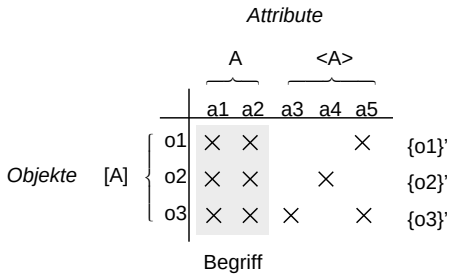


Abbildung 5: Berechnung der signifikanten Attribute

die zu dem selben Ergebnis einer Anfrage führen, ist die Attributmenge des Infimums, also A_1 mit $(O_1, A_1) = \bigwedge_{a \in A} \mu(a)$. In einer Implementierung der Suche sollte der Benutzer über diese impliziten Attribute $A_1 \setminus A$ seiner Anfrage informiert werden und die Anfrage entsprechend vervollständigt werden, da andernfalls nicht jede Erweiterung der Anfrage zu einem kleineren Ergebnis führt.

2.3 Verfeinerung von Anfragen

Eine Anfrage A kann durch Hinzunahme eines Attributs $a \in \mathcal{A} \setminus A$ verfeinert werden, so daß $[\![A \cup \{a\}]\!] \subseteq [\![A]\!]$ gilt. Wenn die Anfrage A bereits alle impliziten Attribute enthält, ist das neue Ergebnis sogar eine echte Teilmenge des alten. Nun sind nicht alle Attribute zur Verfeinerung einer bestehenden Anfrage sinnvoll – als nicht sinnvoll sollen die bezeichnet werden, die im Widerspruch zu den bisherigen Attributen stehen und deshalb zu einem leeren Ergebnis führen. Umgekehrt sollen als sinnvolle Attribute $\langle\langle A \rangle\rangle$ zu einer Anfrage A diejenigen bezeichnet werden, die zu einem kleineren, aber nicht leeren Ergebnis führen: $\langle\langle A \rangle\rangle := \{a \in \mathcal{A} \mid \emptyset \subset [\![A \cup \{a\}]\!] \subset [\![A]\!]\}$. Durch die Wahl eines sinnvollen Attributs zur Verfeinerung wird sichergestellt, daß das neue Ergebnis nicht leer ist.

Die sinnvollen Attribute $\langle\langle A \rangle\rangle$ zur Verfeinerung einer Anfrage A lassen sich ebenso wie das Ergebnis mit Hilfe des Begriffsverbandes bestimmen. Die Objekte O_1 des Ergebnisses weisen

neben den gemeinsamen Attributen A_1 mit $(O_1, A_1) = \bigwedge_{a \in A} \mu(a)$ auch nicht gemeinsame Attribute auf, wie es in Abbildung 5 skizziert ist. Diese Attribute sind die sinnvollen Attribute zur Verfeinerung einer Anfrage: $\langle\langle A \rangle\rangle = (\bigcup_{o \in [\![A]\!]} \{o\}') \setminus A$. Offensichtlich muß das neue Ergebnis $[\![A \cup \{a\}]\!]$ mit $a \in \langle\langle A \rangle\rangle$ eine echte Teilmenge des alten sein, da a nicht gemeinsames Attribut der Objekte des alten Ergebnisses ist, also nicht alle Objekte die neue Anfrage erfüllen. Gleichzeitig ist das neue Ergebnis nicht leer, denn es existiert mindestens eine Objekt in dem alten Ergebnis, das alle bisherigen und das neue Attribut aufweist, also die neue Anfrage erfüllt.

2.4 Inkrementelles Retrieval

Die Suche von Komponenten und die anschließende Verfeinerung einer Anfrage lassen sich zu einem inkrementellen Retrieval von Komponenten kombinieren:

- Die Suche beginnt mit einer leeren Anfrage: $A = \emptyset$ – es gilt $[\![A]\!] = \mathcal{O}$ und $\langle\langle A \rangle\rangle = \mathcal{A}$, wenn $\top = (\mathcal{O}, \emptyset)$ gilt. Der zu der Anfrage gehörende Begriff b ist der größte Begriff $\top$ des Verbandes $\mathcal{B}(\mathcal{O}, \mathcal{A}, \mathcal{R})$.

- Der Benutzer wählt interaktiv aus der Liste der präsentierten sinnvollen Attribute $\langle\langle A \rangle\rangle$ ein Attribut a zur Verfeinerung seiner Anfrage aus. Der neue Begriff $\tilde{b}$ ergibt sich aus dem Infimum des alten und $\mu(a)$: $\tilde{b} = (O_{\tilde{b}}, A_{\tilde{b}}) = b \wedge \mu(a)$. Mit Hilfe von $\tilde{b}$ kann nun das neue Ergebnis und die neue Liste der sinnvollen Attribute berechnet werden: $[\![A \cup \{a\}]\!] = O_{\tilde{b}}$. Dabei wird die bisherige Anfrage um die impliziten Attribute erweitert.

- Wenn das Ergebnis noch keine überschaubare Größe erreicht hat, wird der Verfeinerungsprozeß wiederholt. Dies ist so lange möglich, bis $\langle\langle A \rangle\rangle = \emptyset$ gilt. Dies ist spätestens nach sovielen Schritten der Fall, wie die Komponente in $\mathcal{O}$ mit maximaler Attributanzahl Attribute besitzt.

Da jede Anfrage einem Begriff des Verbandes entspricht, ergibt sich durch die Folge der Anfragen ein Suchpfad entlang den Kanten des Verbandes, der am größten Element $\top$ des Verbandes beginnt. In Abbildung 4 ist ein solcher Pfad eingezeichnet – er ergibt sich durch die Wahl der sinnvollen Attribute *change*, *file* und *mode* und dem Ergebnis `chmod`

2.5 Prototyp

Das beschriebene Verfahren wurde als Prototyp zur Verwaltung einer Unix Online-Dokumentation implementiert. Insgesamt 1658 Dokumente wurden mit mehreren Attributen aus einer Menge von 92 Attributen versehen – der zugehörige Begriffsverband enthält 714 Begriffe. Abbildung 6 zeigt die Schnittstelle des Prototyps zum Benutzer, nachdem die Attribute *change* und *file* ausgewählt wurden.

In der mittleren Liste werden die ausgewählten Attribute A angezeigt und rechts die noch wählbaren Attribute, also die Menge $\langle\langle A \rangle\rangle$. Die linke Liste enthält die durch die Attribute A ausgewählten Dokumente $[\![A]\!]$; durch Anklicken kann das entsprechende Dokument angesehen werden, wie es im Hintergrund für das Kommando `chmod` zu sehen ist. Ein rechts ausgewähltes Attribut wird in die mittlere Liste übernommen und führt zu einer Verringerung der ausgewählten Dokumente und der noch wählbaren Attribute. Die Auswahl eines Attributs geschieht entweder direkt mit der Maus oder durch schriftliche Eingabe, wobei die Angabe

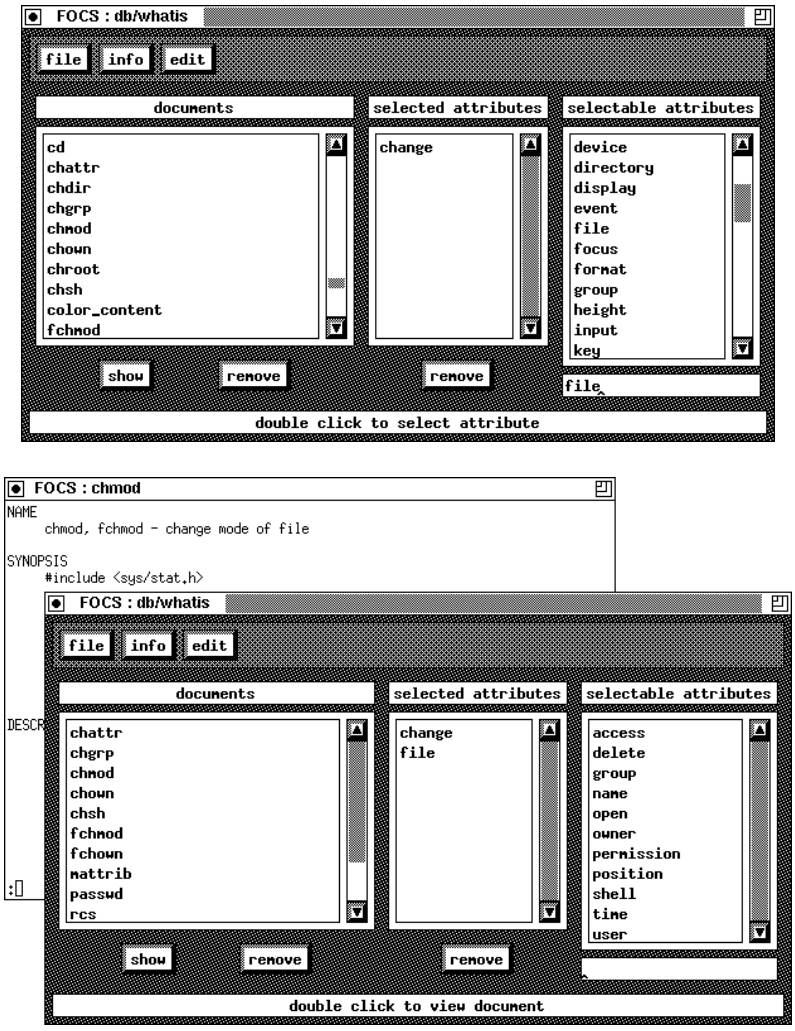


Abbildung 6: Zwei Schritte einer Suche mit dem Prototyp

eines Präfixes schon genügt, um den Listen-Cursor an das entsprechende Attribut springen zu lassen. Ein beliebiges, bereits ausgewähltes, Attribut kann durch nochmaliges Auswählen mit der Maus zurückgenommen werden, woraufhin die Darstellung entsprechend aktualisiert wird. Dabei kann eine völlig neue Situation entstehen, wenn die Menge der dann noch ausgewählten Schlüsselwörter neu ist, so daß das Ergebnis neu berechnet werden muß, statt das auf ein altes Zwischenergebnis zurückgegriffen werden kann.

Die Reaktion des Prototyps auf die Aktionen des Benutzers ist verzögerungsfrei – der Benutzer kann sehr schnell durch den Datenbestand navigieren. Durch die Präsentation seiner verbleibenden Wahlmöglichkeiten wird er dabei stark kontextsensitiv unterstützt. Das Berechnen des Begriffsverbandes aus einer Indexierung von Objekten mit Attributen benötigt 150 Sekunden auf einer SPARCstation ELC. Diese Berechnung ist allerdings nur nötig, wenn sich die Ausgangsdaten geändert haben, ansonsten wird der vorberechnete Begriffsverband benutzt.

Für die mit dem Prototyp verwalteten Dokumente wurde untersucht, wie sich die Zahl der selektierten Dokumente $[\![A]\!]$ und der dann noch wählbaren Schlüsselwörter $\langle\langle A \rangle\rangle$ nach zwei

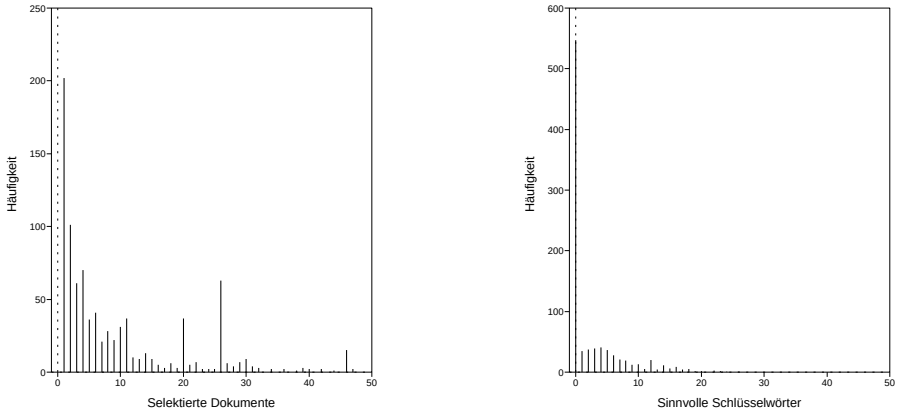


Abbildung 7: Verteilung von Dokumenten und Schlüsselwörtern nach 2 Suchschritten

Suchschritten ($|A| = 2$) verteilen. Von den $\binom{92}{2} = 4186$ theoretisch möglichen Kombinationen von Schlüsselwörtern sind 903 tatsächlich erreichbar. Für diese wurden jeweils die Zahl der selektierten Dokumente und im dritten Schritt wählbaren Schlüsselwörter bestimmt. Abbildung 7 zeigt die beiden Verteilungen: links für die Dokumente, rechts für die Schlüsselwörter.

Bereits nach zwei Suchschritten ist die Zahl der selektierten Dokumente in 52% aller Fälle von anfänglich 1658 auf weniger als 6 gefallen. Gleichzeitig ist die Zahl der noch zusätzlich wählbaren Schlüsselwörter in 68% aller Fälle kleiner als 2. Dies zeigt, wie das Verfahren den Suchraum eingrenzt und damit die Navigation zielgerichtet und schnell wird.

2.6 Ergebnis

Der Begriffsverband einer Sammlung von Komponenten und ihrer Indexierung bildet eine automatisch gewonnene Organsiation der Sammlung, die eine effiziente und inkrementelle Suche nach Komponenten erlaubt. Der Benutzer wird dabei durch die vollständige und genaue Präsentation seiner Wahlmöglichkeiten unterstützt und so schnell und gerichtet zu seinem Ziel geführt. Die Effizienz resultiert aus der Tatsache, daß der Begriffsverand sozusagen eine kompilierte Form aller Entscheidungsmöglichkeiten darstellt. Eine prototypische Implementierung als Teil von NORA demonstriert die praktische Anwendung und die Eignung des Verfahrens zur Verwaltung einer Unix Online-Dokumentation. Da das Suchverfahren unabhängig vom Inhalt der verwalteten Komponenten ist, können auch inhomogene und multimediale Sammlungen begriffsbasiert verwaltet werden.

3 Analyse von Konfigurationsstrukturen

In diesem Hauptteil der Arbeit beschreiben wir NORA/RECS, eine Werkzeug zur Analyse und Restrukturierung von Konfigurationsstrukturen. Wir beginnen mit einer kurzen Beschreibung des softwaretechnologischen Hintergrundes.

3.1 Software-Konfigurationsmanagement

Früher war die Welt des Software-Entwicklers einfach: er entwickelte ein Programm, das in nur *einer* spezifischen Hard- und Softwareumgebung (die sog. Plattform) ablauffähig sein mußte.

```
...I...
#ifdef DOS
...II...
#endif
#ifdef OS2
...III...
#endif
#if defined(DOS) && defined(X_win)
...IV...
#endif
#ifdef X_win
...V...
#endif
...VI...
```

Abbildung 8: Variantenauswahl mit dem Präprozessor

Software wurde von den Hardwareherstellern eifersüchtig gehütet; es war nahezu unmöglich, Software zwischen Plattformen zu migrieren. Dies hatte den großen „Vorteil“, das Kunden an den jeweiligen Hersteller gebunden wurden.

Mit dem Aufkommen von Workstations und PCs änderte sich diese Welt. Die neuen Maschinen haben herstellerunabhängige Betriebssysteme (jedenfalls im Prinzip), und dies führte zur Forderung nach *Interoperabilität*: moderne Anwendungssoftware muß – im Gegensatz zu Systemsoftware – heute in ganz verschiedenen Umgebungen laufen, sei es DOS, UNIX oder gar BS2000, sonst lassen sich keine hinreichend großen Marktanteile erreichen. Nach wie vor sind jedoch Teile des Systems und damit des Quelltextes plattformspezifisch. Für jede Installation müssen die passenden Codestücke bzw. Komponenten zusammengebaut werden; dies nennt man *Konfigurieren*. Konfigurationsmanagement umfaßt das Verwalten, Selektieren und Zusammensetzen von konfigurationsspezifischem Code. Plattformspezifische Codestücke müssen nach vorgegebenen Kriterien ausgewählt und zu einem vollständigem Quelltext komponiert werden. Die Menge aller möglichen Konfigurationen wird als Konfigurationsraum bezeichnet, und eine plattformspezifische Komposition von Codestücken nennt man Konfigurationspfad oder kurz Konfiguration.

Es gibt heute sehr ausgefeilte Konfigurationsmanagementsysteme. Industriestandard im UNIX-Bereich ist jedoch der C-Präprozessor CPP. Dieser wird sowieso bei jedem Compilerlauf zur Vorverarbeitung des Quelltextes aufgerufen und kann dann die Auswahl und Komposition eines Konfigurationspfades gleich mitübernehmen. Dazu wird die Fähigkeit des CPP zur *bedingten Textauswahl* ausgenutzt: konfigurationsspezifischer Code wird in sog. `#ifdef...#endif` Klammern eingeschlossen; dieser Code wird nur bei Erfülltsein der *regierenden Bedingung* in den Konfigurationspfad übernommen (inkludiert). Regierende Bedingungen sind boolesche Ausdrücke über sog. *Präprozessorsymbolen*, die beim Aufruf des Compilers „gesetzt“ (definiert) werden können und deren Definiertsein in regierenden Bedingungen abgefragt werden kann.

Abbildung 8 zeigt ein einfaches Beispiel. Die Codestücke I und VI sind konfigurationsunabhängig und werden immer inkludiert. Codestück II wird nur inkludiert, wenn “DOS“ definiert ist, Codestück III nur wenn „OS2“ und Codestück V nur wenn „X_win“ gesetzt ist.

Codestück IV wird nur inkludiert, wenn sowohl „DOS“ als auch „X_win“ definiert sind. Letzteres kann eigentlich nicht vorkommen, da das X-Window-System nur unter UNIX läuft; das Beispiel soll aber gerade zeigen, wie solche problematischen Codestücke durch einen Begriffsverband aufgedeckt werden können.

Um eine spezifische Konfiguration zu erzeugen, müssen beim Compileraufruf „cc“ die entsprechenden CPP-Symbole definiert werden. Für die DOS-Version muß „DOS“ definiert werden:

```
cc -DDOS prog.c
```

und für die OS/2–Version muß „OS2“ definiert werden:

```
cc -DOS2 prog.c
```

Da `#ifdef`s beliebig geschachtelt werden und beliebig komplexe regierende Ausdrücke enthalten können, können sich beliebig unverständliche Quelltexte ergeben. Ein Beispiel ist das X-Window-Tool „xload“, das gewisse Systemauslastungsfaktoren anzeigt. Das 724-zeilige Programm benutzt 43 CPP-Symbole, um eine Fülle von Konfigurationen zu verwalten (z.B. SYSV, macII, ultrix, sun, CRAY, sony). Abbildung 9 zeigt einen Quelltextausschnitt. Man beachte, daß sogar Teile von Ausdrücken konfigurationsspezifisch ausgewählt werden. Das Programm ist völlig unverständlich, eine Dokumentation ist nicht vorhanden, Änderungen oder Erweiterungen sind mit einer hohen Fehlerwahrscheinlichkeit behaftet, der Entropietod ist nahe – es ist Zeit für Reengineering und Restrukturierung.

3.2 Reengineering von Konfigurationsstrukturen

Ziel des Reengineering ist es, aus Quelltexten wie „xload.c“ die Struktur des Konfigurationsraumes zu extrahieren. Dies erlaubt erst ein Verständnis des Quelltextes und ist die Grundlage für spätere Restrukturierungsversuche. Ausgangspunkt unseres Werkzeugs Nora/Recs ist ein Quelltext mit Präprozessoranweisungen. Das Verfahren kann auch auf die moderneren Konfigurationsmanagementsysteme übertragen werden, jedoch wäre dies eine rein technische Aufgabe. Das Vorgehen ist im Prinzip wie folgt:

1. Aufstellen der Konfigurationstabelle
2. Berechnung und Anzeige des entsprechenden Begriffsverbands
3. Analyse und Zerlegung des Verbandes (Reengineering)
4. entsprechende Zerlegung und Transformation des Quelltextes (Restrukturierung)

Die *Konfigurationstabelle* enthält die Klassifikation der Codestücke nach regierenden CPP-Symbolen (Abbildung 10). Falls ein Codestück von einem CPP-Symbol regiert wird, wird ein Kreuz in die Tabelle eingetragen; auf diese Weise ergibt sich ein formaler Kontext. Da Quelltexte geschachtelte `#ifdef`s und boolsche Ausdrücke über CPP-Symbolen enthalten können, ist die Konstruktion der Tabelle nicht trivial. Komplexe regierende Ausdrücke müssen zuerst in konjunktive Normalform gebracht werden. Für negierte CPP-Symbole müssen zusätzliche Spalten eingeführt werden, da formale Kontexte nur positive Aussagen ausdrücken können (dichotomisierter Kontext). Falls die konjunktive Normalform elementare

```
#if (!defined(SVR4) || !defined(__STDC__)) && !defined(sgi) &&
!defined(MOTOROLA)
 extern void nlist();
#endif
#ifdef AIXV3
 knlist( namelist, 1, sizeof(struct nlist));
#else
   nlist( KERNEL_FILE, namelist);
#endif
#ifdef hcx
    if (namelist[LOADAV].n_type == 0 &&
#else
 if (namelist[LOADAV].n_type == 0 ||
#endif /* hcx */
 namelist[LOADAV].n_value == 0) {
 xload_error("cannot get name list from", KERNEL_FILE);
 exit(-1);
   }
 loadavg_seek = namelist[LOADAV].n_value;
#if defined(umips) && defined(SYSTYPE_SYSV)
    loadavg_seek &= 0x7fffffff;
#endif /* umips && SYSTYPE_SYSV */
#if (defined(CRAY) && defined(SYSINFO))
   loadavg_seek += ((char *) (((struct sysinfo *)NULL)->avenrun)) -
((char *) NULL);
#endif /* CRAY && SYSINFO */
   kmem = open(KMEM_FILE, O_RDONLY);
 if (kmem < 0) xload_error("cannot open", KMEM_FILE);
#endif
```

Abbildung 9: Quelltextausschnitt aus dem X-Window Tool „xload“

Disjunktionen der Form $a \vee b \vee c$ enthält (wobei a, b, c einfache oder negierte CPP-Symbole sind), müssen für diese Disjunktionen gleichfalls zusätzliche Spalten eingeführt werden; jedes Kreuz, das in einer Spalte für a, b oder c eingeführt wird, führt zum Eintragen eines zusätzlichen Kreuzes in der Spalte für die Disjunktion. Hierdurch werden triviale Implikationen der Form $a \Rightarrow a \vee b \vee c$ in die Konfigurationstabelle eingebracht [Kr93]. Der Prozeß der Tabellenerzeugung aus dem Quelltext ist vollautomatisiert, Details finden sich in [Sn95].

Nachdem die Konfigurationstabelle erzeugt ist, wird der entsprechende Begriffsverband berechnet. Abbildung 11 zeigt den Begriffsverband zu Abbildung 10, wie er vom Werkzeug Nora/Recs angezeigt wird. Am Begriffsverband läßt sich die Struktur des Konfigurationsraums genau ablesen:

- für jede Konfiguration kann ihr *Umfang* (i.e. die Codestücke, die zur Konfiguration gehören) und ihr *Inhalt* (i.e. die CPP-Symbole, die die Konfiguration „regieren“) abgelesen werden;

```
...I...
#ifdef DOS
...II...
#endif
#ifdef OS2
...III...
#endif
#if defined(DOS) && defined(X_win)
...IV...
#endif
#ifdef X_win
...V...
#endif
 ...VI...
```

	DOS	OS2	X_win
I			
II	×		
III		×	
IV	×		×
V			×
VI			

Abbildung 10: ein einfacher Quelltext und seine Konfigurationstabelle

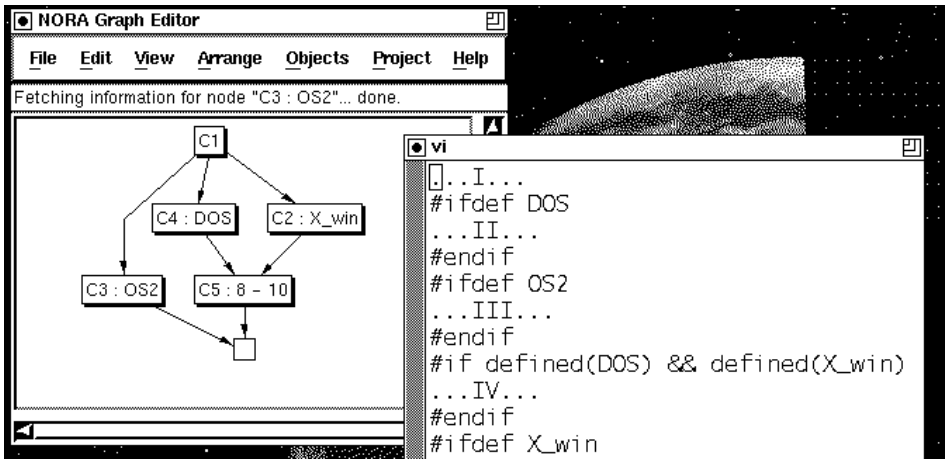


Abbildung 11: entsprechender Begriffsverband

- alle *Abhängigkeiten* (Implikationen) zwischen Konfigurationen werden sichtbar, wobei Abhängigkeiten von der Form sind: „Alle Codestücke, die in der SUN-Konfiguration vorkommen, kommen auch in der Ultrix- und Sony-Konfiguration vor“;
- Der Konfigurationsverband macht eine *Taxonomie* von Konfigurationen sichtbar;
- unzulässige Abhängigkeiten (sog. *Interferenzen*) erscheinen als Infima im Verband; dabei zeigt eine Interferenz an, daß Konfigurationen gemeinsamen Code haben, was u.U. softwaretechnischen Prinzipien widerspricht;
- anhand der Verbandsgestalt kann die Qualität des Konfigurationsraums entsprechend softwaretechnischer Kriterien beurteilt werden.

Betrachtet man etwa den Verband in Abbildung 11, so tauchen die vier möglichen Konfigurationen des Quelltextes als Begriffe auf (Top- und Bottomelement stehen normalerweise nicht für zulässige Konfigurationen: das Topelement steht für alle Codestücke, die konfigurationsunabhängig sind, das Bottomelement steht für CPP-Symbole, die nichts regieren). Man

sieht, daß die „OS2“-Konfiguration unabhängig von den drei anderen ist. Hingegen haben die „DOS“- und die „UNIX“-Konfiguration gemeinsamen Code, denn Zeilen 8 – 10 hängen sowohl von „DOS“ als auch von „X_win“ ab. Dies ist sehr verdächtig, denn DOS und X_win sind eigentlich unverträglich, da das X-Window-System – wie alle Eingeweihten wissen – nur auf UNIX-Plattformen läuft. Mithin haben wir durch einfaches Betrachten des Verbandes verdächtige Codeteile entdeckt, die im Verband als Interferenz sichtbar werden. Mag dieses Beispiel auch trivial sein, so ist eine derartige Analyse für komplexe Programme wie „xload“ von Hand praktisch nicht zu leisten!

Das Beispiel zeigt aber auch, daß die erhaltenen Verbände interpretiert werden müssen. Ob ein Infimum tatsächlich eine Interferenz ist, hängt nämlich von der Bedeutung der involvierten CPP-Symbole ab. Diese kann aus einer „syntaktischen“ Analyse, wie sie die elementare Begriffsanalyse darstellt, allein nicht erschlossen werden – wenngleich der Begriffsverband eine enorme Hilfe darstellt. Der Restrukturierer muß i.a. selbst Wissen einbringen, z.B. daß DOS und X-Windows nicht zusammenpassen. Andererseits kann es sehr wohl sinnvoll sein, daß Konfigurationen gemeinsamen Code haben, wenn nämlich die regierenden Präprozessorsymbole verschiedene Varianten desselben Konfigurationsteilraums beschreiben. Hat man etwa verschiedene Varianten eines Fenstersystems, die verschiedene „Features“ anbieten, so kann Code, der eine gewisse *Kombination* von Features behandelt, sehr wohl sinnvoll sein; das gleiche gilt für verschiedene Features einer Betriebssystemfamilie. Werden aber wie im Beispiel Betriebssystem- und Fenstersystemaspekte vermischt, so widerspricht dies dem softwaretechnischen Prinzip der *schwachen Kopplung*. Dieses besagt, daß orthogonale Aspekte eines Softwaresystems wie Betriebssystem- bzw. Fenstersystemspezifika im Quelltext streng getrennt werden sollten. Nach der herrschenden Lehrmeinung müssen Fenster- bzw. Betriebssystemspezifika in getrennten Modulen gekapselt werden (*Geheimnisprinzip* von Parnas).

Aus diesen Überlegungen ergibt sich schon, daß die entstehenden Verbände möglichst flach sein sollten. Ein flacher Konfigurationsverband zeigt an, daß es keine wie auch immer gearteten Abhängigkeiten zwischen Konfigurationen gibt, was aus softwaretechnischer Sicht optimal ist. Ist der Verband nicht flach, so sollte er wenigstens horizontal zerlegbar sein. Die CPP-Symbole in den Teilverbänden (horizontalen Summanden) sollten orthogonal, also unabhängig voneinander sein: ein Unterverband enthält z.B. alle Fenstersystemvarianten, der andere alle Betriebssystemvarianten (Abbildung 12).

Wir haben diverse UNIX-Programme mit Nora/Recs analysiert, darunter auch das RCS-System zur Versionsverwaltung. Ein besonders eindrucksvolles Beispiel für die Leistungsfähigkeit der Begriffsanalyse lieferte der RCS-Stream Editor. Dieses Hilfsprogramm des RCS-Systems hat 1656 Zeilen und verwendet 21 CPP-Symbole zum Konfigurationsmanagement. Abbildung 13 zeigt den entsprechenden Begriffsverband. Die linke Verbandshälfte ist flach, was positiv zu bewerten ist (die entsprechenden Varianten befassen sich mit gewissen Fea-

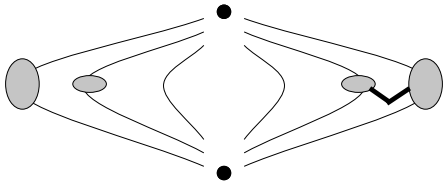


Abbildung 12: Horizontale Verbandszerlegung und eine Interferenz

tures diverser UNIX-Versionen). Hingegen sind rechts einige Interferenzen zu erkennen. So ist etwa C3 mit „has_rename“ markiert; C26 ist mit „has_NFS“ markiert, und C27 ist mit „1425 – 1427“ markiert. Mithin werden die Zeilen 1425 - 1427 sowohl von „has_rename“ als auch von „has_NFS“ regiert. Wie jeder UNIX-Kenner weiß, hat „rename“ etwas mit dem Dateisystem zu tun, „NFS“ ist hingegen das Netzwerk. Beide Aspekte sind eigentlich orthogonal, denn Dateisystem und Netzwerksystem sollten unabhängig voneinander konfiguriert werden können. Die Interferenz in C27 ist also äußerst verdächtig. Und tatsächlich: ein Blick in den Quelltext zeigt, daß der entsprechende Code der Reparatur eines NFS-Fehlers dient: eine bestimmte Kombination von Dateisystem und Netzwerk kann nämlich dazu führen, daß „rename“ die RCS-Datei zerstört! Dies war den Programmierern durchaus bewußt, denn in einem Kommentar ist zu lesen: „*An even rarer NFS bug can occur when clients retry requests. ... This not only wrongly deletes B's lock, it removes the RCS file! ... Since this problem afflicts scads of Unix programs, but is so rare that nobody seems to be worried about it, we won't worry either*“.

3.3 Verbandsanalyse mit NORA/RECS

Nachdem der Verband berechnet ist, kann er interaktiv inspiziert und zerlegt werden. NORA/RECS bietet dazu die folgenden Funktionen an:

- Die Begriffsmarkierungen (Codestücke und CPP-Symbole) können angezeigt werden;
- Der entsprechende Quellcode kann angezeigt werden;
- Der Verband kann in horizontale Summanden zerlegt werden (falls möglich);
- Unterverbände bzw. Unterhalbordnungen können ausgewählt werden;
- Interferenzen können automatisch bestimmt werden;

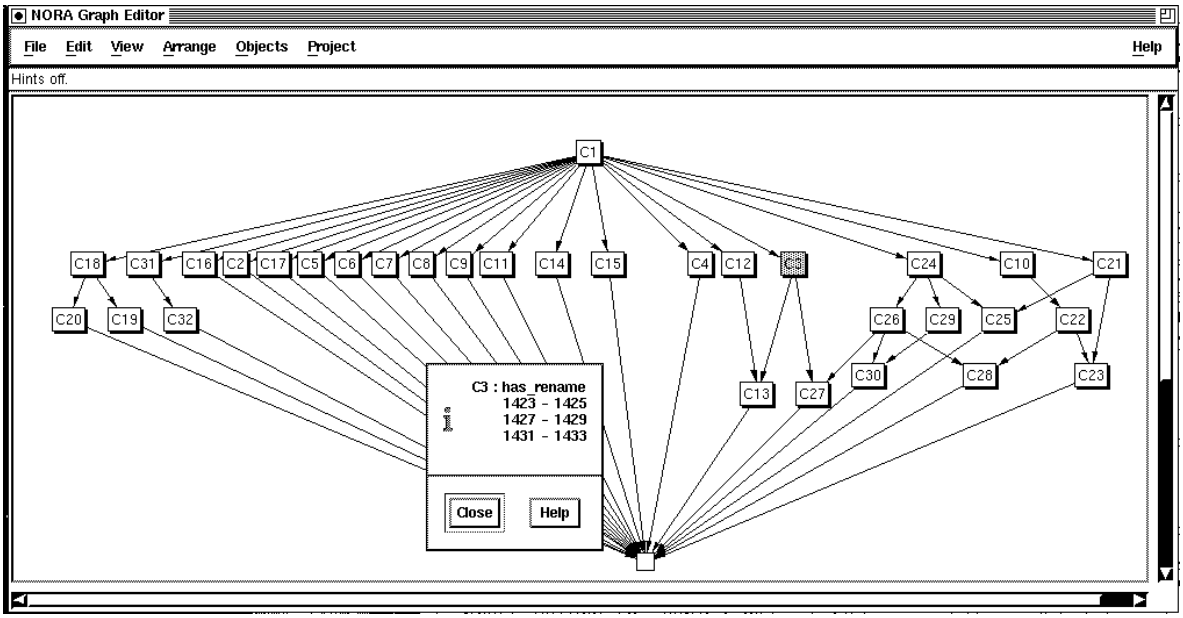


Abbildung 13: Konfigurationsraum des RCS-Stream-Editors

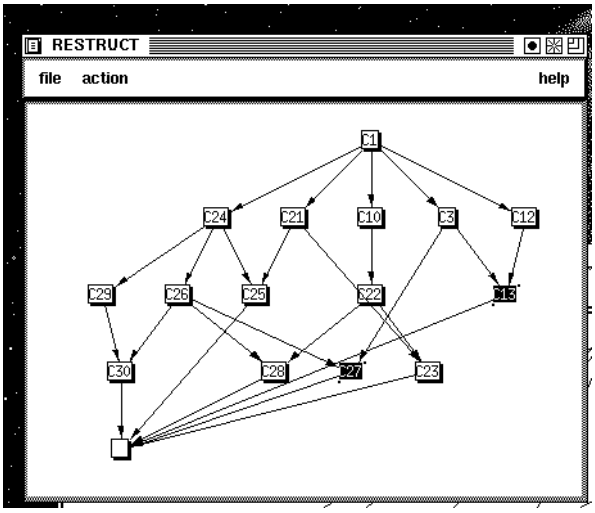


Abbildung 14: Interferenzanalyse in einem Unterverband

- Restrukturierungen können angestoßen werden.

Für kleine Verbände werden die Markierungen direkt in die Begriffe gezeichnet; dies ist aber schon bei mittelgroßen Verbänden unmöglich. Deshalb werden die Markierungen und entsprechenden Codestücke durch Anklicken in einem separaten Fenster dargestellt (Abbildung 13). Nach einer horizontalen Zerlegung können die entstandenen Teilverbände rekursiv weiterverarbeitet werden. Unterhalbordnungen sind gleichfalls sehr interessant: eine Unterhalbordnung $\downarrow \{a_1, a_2, ..., a_n\}$ gibt alle Codestücke an, die gleichzeitig von $a_1, a_2, ..., a_n$ abhängen; Durchschnitte von Unterverbänden enthalten in der Regel Interferenzen. In großen Verbänden (z.B. Abbildung 15) ist diese Information nicht ohne weiteres zu erkennen. Es ist geplant, Unterverbände farbig darzustellen, so daß Interferenzen durch ihre „Mischfarbe“ sofort sichtbar werden.

Eine andere Methode zur Interferenzbestimmung nutzt die Konnektivität des Verbandsgraphen: Wenn der Graph nach Entfernen des Top- und Bottomelementes in Teilgraphen zerfällt, sind die Teilgraphen horizontale Summanden. Gibt es zwischen Teilgraphen eine „Brücke“ – zwei Kanten, die zu einem Infimum führen und deren Entfernung den Graph zerfallen läßt – so entspricht die Brücke einer Interferenz der Konnektivität 1. Muß man zwei derartige Brücken entfernen, um den Graphen zerlegbar zu machen, bilden diese eine Interferenz der Konnektivität 2 usw. Diese Interferenzen können gleichfalls am Bildschirm sichtbar gemacht werden. Wie oben ausgeführt, müssen alle Interferenzkandidaten jedoch noch manuell überprüft werden, ob wirklich aus semantischer Sicht eine unzulässige Verkupplung von Konfigurationspfaden vorliegt.

Abbildung 14 zeigt den rechten Teilgraphen von Abbildung 13, der durch horizontale Zerlegung entstanden ist. Er enthält zwei Interferenzen der Konnektivität 1, die markiert dargestellt werden. Die C27-Interferenz wurde bereits erläutert, hingegen ist die C13-Interferenz in Wirklichkeit keine, weil sich sowohl C3 als auch C12 mit Dateisystemvarianten befassen; mithin werden keine orthogonalen Konfigurationsaspekte verkoppelt.

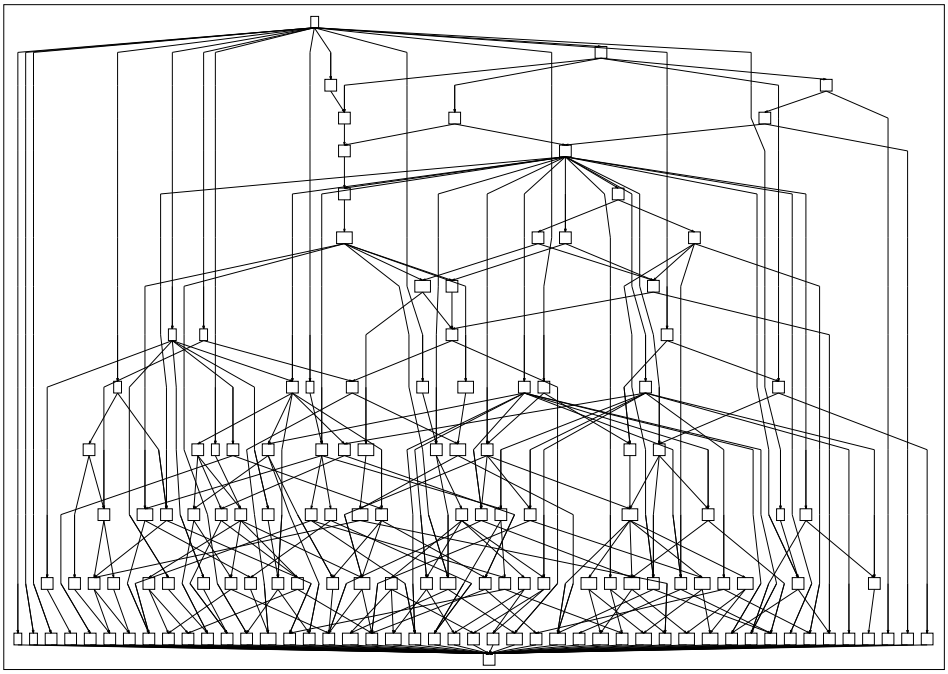


Abbildung 15: Konfigurationsstruktur von xload.c

3.4 Restrukturierung durch Verbandszerlegung

Das Beispiel des RCS-Stream-Editors zeigt, daß Interferenzen manchmal nicht zu verhindern oder aufzulösen sind, denn die zugrundeliegende Ursache – der NFS-Fehler – kann nicht behoben werden. In anderen Fällen hat man es aber mit regelrechtem Konfigurationshacking zu tun, das einer Restrukturierung bedarf. Als Beispiel betrachten wir den Konfigurationsverband zu „xload“ (vgl. Abbildung 9), der in Abbildung 15 zu sehen ist. Der Verband hat über hundert Begriffe, und von horizontaler Zerlegbarkeit kann keine Rede sein – sogar die Interferenzen der Konnektivität 3 führen nicht zum Zerfall des Verbandes, sondern nur zum Abspalten einzelner Begriffe.

Wir wollen nun zeigen, wie der Begriffsverband zur Restrukturierung verwendet werden kann. Ziel ist es, unverständliche Konfigurationsräume zu vereinfachen und in Teilräume aufzuspalten, wobei die Software-Engineering-Prinzipien „hohe Kohäsion“ (in einem Teilraum) und „schwache Kopplung“ (zwischen Teilräumen) beachtet werden müssen. Dies entspricht einer „Modularisierung“ des Quelltextes und geschieht im Prinzip durch eine horizontale Verbandszerlegung. Ein weiteres Ziel ist die Vereinfachung regierender Ausdrücke, wobei nach Möglichkeit nicht nur die bekannten Regeln für boolsche Ausdrücke, sondern auch verbandsspezifische Eigenschaften ausgenutzt werden sollen. In der Tat können die irreduziblen Elemente zur Ausdrucksvereinfachung verwendet werden.

Wir wollen zunächst die Modularisierung beschreiben. Diese erzeugt keine traditionellen Module, aber liefert eine möglichst entkoppelte Zerlegung des Quelltextes, wobei jeder Quelltext einen Teil des Konfigurationsraumes abdeckt. Läßt sich etwa der Verband interferenzfrei in horizontale Summanden zerlegen, und sind die CPP-Symbole in den Unterverbänden jeweils disjunkt (d.h. sie können nicht gleichzeitig definiert werden), so kann man den Quelltext in Module zerlegen, wobei jedem Unterverband ein Modul entspricht. Trotzdem bleiben alle

Konfigurationspfade erhalten.

Die Quelltextzerlegung basiert auf partieller Auswertung von CPP-Ausdrücken. Partielle Auswertung beruht auf der Annahme, daß *einige* CPP-Symbole einen bekannten Wert (also „gesetzt“ bzw. „nicht gesetzt“) haben. In diesem Fall kann man den Quelltext vereinfachen: eventuell fallen Codestücke komplett weg, zumindest werden regierende Ausdrücke kürzer. Im vereinfachten Quelltext kommen nur noch CPP-Symbole vor, deren Wert gerade nicht bekannt war. Das gewöhnliche CPP-Verhalten ergibt sich als Grenzfall, wenn nämlich *alle* CPP-Symbole einen definierten Wert (gesetzt / nicht gesetzt) haben. Abbildung 16 zeigt ein einfaches Beispiel, bei dem Code unter der Annahme vereinfacht wird, daß „A“ nicht gesetzt ist, die Werte der anderen CPP-Symbole hingegen unbekannt sind. Die Implementierung der partiellen Auswertung ist nicht trivial, denn es können beliebige „Kontextausdrücke“ angegeben werden (nicht nur einfache Symbole). Das Verfahren ist Teil des inferenzbasierten Konfigurationsmanagementsystems NORA/ICE; Details finden sich in [ZS95].

Falls nun der Verband horizontal in Unterverbände zerlegt werden kann, deren CPP-Symbole disjunkt sind, können daraus Module erzeugt werden. Dies geschieht, indem die CPP-Symbole eines Unterverbandes als „nicht gesetzt“ definiert werden und der Quelltext

```
#ifdef A
...I...
#endif
#ifdef B
...II...
#endif
#if defined(B)||defined(C)
...III...
#endif
#ifdef A
...IV...
#endif
#if defined(A) ||
    (defined(B)&&defined(C))
...V...
#endif
```

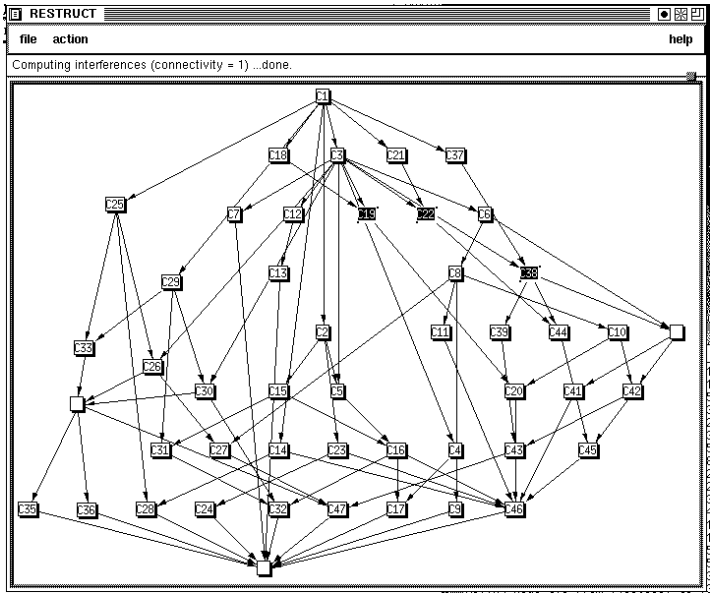
$\Longrightarrow$

```
#ifdef B
...II..
#endif
#if defined(B) || defined(C)
...III...
#endif
#if defined(B)&&defined(C)
...V...
#endif
```

Abbildung 16: Partielle Auswertung unter Kontextausdruck $\neg defined(A)$

```
/* OS variants */ /* WS variants */ /* problematic variant */
...I...           ...I...           #if defined(DOS)
#ifdef DOS        #ifdef X_win          && defined(X_win)
...II...          ...V...           ...I...
#endif            #endif            ...II...
#ifdef OS2        ...VI...          ...IV...
...III...                           ...V...
#endif                              ...VI...
...VI...                            #endif
```

Abbildung 17: Zerlegung in Konfigurationsteilräume



C1		0-36 38-40 42-44 50-53 60-63 72-75 77-82 86 -93 180-181 202-203 205-206 208-209 213-214 224 - 225 228-229 231-232 252-253 255-256 258-259 263 - 264 418-419 493-494 497-506
C2	SVR4 OR sun	
C3	!SYSV OR !SYSV386	181-191 193-195 197-2022 214-220 223-224 232-240 244-248 250-252 264-273 282-285 300-307 353-364 368-370 417-418 494-497
C4	!USG	276-277 281-282
C5		370-376
C6	!STDC__ OR !SVR4	
C7	USG 273-276	
C8	!SVR4	222-223
C9		279-281
C10	!sun	376-377 416-417
C11	!SYSV	229-231
C12	hpux	191-193 240-241 243-244
C13	hp9000s800	241-243
C14	SYSV386	
C15	SVR4	82-86
C16		220-222
C17		277-279
C18	macII	63-72
C19		195-197 285-294 307-328 364-366
C20		377-382
C21	AIXV3	53-60
C22		248-250
C23	sun	44-46 49-50
C24	i386	46-49
C25	SYSV	
C26	!SYSV386	
C27		225-228
C28		93-180
C29	MOTOROLA	
C30		209-210 212-213 259-260 262-263
C31		75-77
C32		210-212 260-262
C33		40-42
C34		
C35	m88k	206-208 256-258
C36	m68k	203-205 253-255
C37	!macII	36-38
C38		294-300 328-329 331-333 337-353 366-368
C39	!AIXV3	335-337
C40		
C41	!MOTOROLA	329-331
C42		382-383 415-416
C43		414-415
C44		333-335
C45		383-414
C46		490-493
C47		419-490
C48		

Abbildung 18: „xload“ nach Amputation irrelevanter Konfigurationen

damit partiell ausgewertet wird. Falls die Symbole der Teilverbände nur orthogonal, aber nicht disjunkt sind, können Konfigurationen verloren gehen; in diesem Fall ist eine manuelle Prüfung der Zerlegbarkeit notwendig. Bei Interferenzen können spezielle, „problematische“ Versionen erzeugt werden, indem alle Symbole, die nicht oberhalb der Interferenz liegen, als „nicht gesetzt“, und alle oberhalb der Interferenz als „gesetzt“ definiert werden. Da ja stets Interferenzen minimaler Konnektivität berechnet werden, garantiert die Zerlegung anhand des Verbandes hohe Kohäsion und schwache Kopplung.

Da die Zerlegung anhand automatisch bestimmter Interferenzen nicht immer befriedigend ist, bietet NORA/RECS noch eine „manuelle“ Möglichkeit: Der Restrukturierer kann - wie bereits erwähnt - Unterverbände auswählen, indem das Top-Element bzw. die maximalen Atome angeklickt werden. Zu jedem Unterverband kann mit partieller Auswertung ein entsprechender Quelltext erzeugt werden, indem alle Symbole außerhalb des Unterverbandes als „nicht gesetzt“ definiert werden.

Als Beispiel betrachten wir den Quelltext aus Abbildung 8. Wählt man im Verband einerseits die beiden Betriebssystem-CPP-Symbole, andererseits das X-Window-CPP-Symbol, so erhält man zwei Unterverbände, denen die linken beiden Quelltexte in Abbildung 17 entsprechen. Man erhält zwei „Module“, jeweils für den betriebssystemspezifischen bzw. fensterspezifischen Code. Da X-Window und die beiden anderen CPP-Symbole disjunkt sind, bleiben alle Konfigurationspfade erhalten. Anhand der Interferenz kann eine spezielle „problematische Version“ erzeugt werden (rechts in der Abbildung), die aber in unserem Beispiel wenig Sinn macht.

Das Beispiel zeigt, daß für die verbesserte Kohäsion innerhalb der neuen Quelltexte ein Preis bezahlt werden muß: Codestück I und VI kommen in beiden Dateien vor, und es ist bekannt, daß Codeduplizierung fehleranfällig ist: man ändert aus Versehen leicht den Code in einer Datei, nicht aber in der anderen. Zukünftige Forschung muß zeigen, ob man z.B. durch subdirekte Zerlegung von Begriffsverbänden redundanzfreie Module erhalten kann.

Oft weiß der Restrukturierer (oder sieht anhand des Verbandes), daß bestimmte Konfigurationen nicht mehr benötigt werden, und möchte den Quelltext entsprechend vereinfachen. Auch dies kann mit partieller Auswertung geschehen, indem die entsprechenden CPP-Symbole als „nicht gesetzt“ definiert werden. Im Grunde wird auch hier ein Unterverband ausgewählt, jedoch wird keine vollständige Zerlegung angestrebt. Parnas nennt eine derartige Vereinfachung deshalb *Amputation* [Pa94]. Der vereinfachte Code kann dann nur noch auf einer Teilmenge der ursprünglichen Plattformen installiert werden.

Als Beispiel betrachten wir eine Vereinfachung von „xload.c“. Amputiert werden sollen die Konfigurationen apollo, X_NON_POSIX, sony, CRAY, mips, umips, att, LOADSTUB, alliant, sequent, UTEK, hcx, sgi. Der resultierende Quelltext hat nurmehr 501 Zeilen, sein Verband hat nur noch 48 Begriffe (Abbildung 18 zeigt nicht nur den Verband, sondern auch alle Markierungen). Durch Auswahl von C3 kann man nun z.B. noch alle „not System V“ Konfigurationen selektieren, was zu einem noch wesentlich kleineren Quelltext führt. Dieser kann dann natürlich nur noch auf “not System V“ Plattformen installiert werden!

Abschließend sei noch einmal auf die Möglichkeit hingewiesen, die regierenden Ausdrücke mittels irreduzibler Elemente zu vereinfachen. Bekanntlich kann man sich bei den Attributen (CPP-Symbole) bzw. den durch sie markietren Begriffen auf die *meet-irrduziblen Begriffe* beschränken, also auf die, von denen nur eine Kante im Verband nach oben weist. Hat ein meet-irreduzibler Begriff keine Markierung mit einem (einfachen oder negierten) CPP-Symbol, so muß für diesen Begriff ein neues CPP-Symbol eingeführt werden. Nachdem dies geschehen ist, lassen sich alle anderen CPP-Symbole als Konjunktion von Irreduziblen darstellen. Dies ver-

einfacht die regierenden Ausdrücke u.U. erheblich; man braucht weniger CPP-Symbole, und insbesondere *verschwinden alle Disjunktionen*. Die regierenden Ausdrücke eines Codestücks enthalten nur noch Konjunktionen einfacher oder negierter CPP-Symbole, nämlich jene, die im Verband oberhalb des mit dem Codestück markierten Begriffs liegen. Allerdings können darunter neu eingeführte CPP-Symbole sein, die für elementare Disjunktionen stehen. Dies ist ein Hinweis, daß die Wahl der ursprünglichen Symbole nicht optimal war; Nora/Recs liefert den Verbesserungsvorschlag gleich mit.

4 Ausblick

Nora/Recs eignet sich hervorragend zur *Analyse* von Konfigurationsabhängigkeiten; als Werkzeug zur *Restrukturierung* steckt der Ansatz aber noch in den Kinderschuhen. Der Nutzen der formalen Begriffsanalyse ist allerdings schon jetzt deutlich geworden. Wir wollen als nächstes darangehen, subdirekte Zerlegungen von Begriffsverbänden zu untersuchen: wenn der Verband sich subdirekt so zerlegen läßt, daß in den Faktoren nur orthogonale CPP-Symbole stehen, so kann dies Grundlage wesentlich verbesserter Restrukturierungsverfahren sein.

Neben der Analyse von Konfigurationsstrukturen sind ohne weiteres andere Anwendungen der Begriffsanalyse im Software-Reengineering vorstellbar. So sind Software-Architekturen ganz allgemein durch Beziehungen zwischen Komponenten definiert und können mithin Gegenstand von Begriffsanalyse sein; hierbei können auch skalierte Kontexte eingesetzt werden, da es verschiedene Arten von Beziehungen gibt. Überhaupt kommen Relationen zwischen Objekten und Attributen im Software Engineering andauernd vor, so daß neben Reengineering und Reuse noch ganz andere Anwendungen der formalen Begriffsanalyse denkbar sind.

Danksagung. Maren Krone implementierte das Frontend von Nora/Recs, Anke Lewien implementierte die Verbandszerlegung. Andreas Zeller implementierte den Graph-Editor und -Layouter und entwickelte Nora/Ice. Peter Dettmer implementierte Nora/Focs. Martin Skorsky lieferte wertvolle theoretische Unterstützung.

Nora wird von der DFG unter den Kennzeichen Sn11/1-1, Sn11/1-2, Sn11/2-1, Sn11/2-2, Sn11/4-1 gefördert.

Nora/Recs, Nora/Ice und Nora/Focs sind von den Autoren erhältlich.

Literatur

Zwischenzeitlich sind die bereits angesprochenen Arbeiten [Sn95, ZS95, Ze95] international publiziert worden. Darüberhinaus sind zwei neue Anwendungen [LS97, Li98] formaler Begriffsanslyse dazugekommen, die die im Ausblick formulierte Richtung einschlagen.

[FKSt95] Fischer, B., Kievernagel, M., Struckmann, W.: VCR: A VDM-Based Software Component Retrieval Tool. Proc. ICSE-17 Workshop on Formal Methods in Software Engineering.

[FKSn95] Fischer, B., Kievernagel, M., Snelting, G. : Deduction-Based Software Component Retrieval. Proc. IJCAI-95 workshop on Reuse of Plans, Proofs, and Programs.

[Gr95a] Grosch, F.-J.: Eine typisierte, rein funktionale Modulsprache für das Programmieren im Großen. Proc. 12. Workshop der GI FG 2.1.4, Bad Honnef 1995.

[Gr95b] Grosch, F.-J.: No Type Stamps and No Structure Stamps – a Fully Applicative Higher-Order Module Language. Informatik-Bericht 95–05, TU Braunschweig 1995.

[Kr93] Krone, M.: Reverse Engineering von Konfigurationsstrukturen. Diplomarbeit, TU Braunschweig, Abteilung Softwaretechnologie, September 1993.

[KS94] Krone, M., Snelting, G.: On the Inference of Configuration Structures from Source Code. Proc. 16th International Conference on Software Engineering, IEEE 1994, pp. 49-58.

[Le95] Lewien, A.: Restrukturierung von Konfigurationen mit formaler Begriffsanalyse. Diplomarbeit, TU Braunschweig, Abteilung Softwaretechnologie, September 1995.

[Li95a] Lindig, C.: Concept Based Component Retrieval. Proc. IJCAI-95 Workshop on Reuse of Plans, Proofs, and Programs, Montréal, August 1995.

[Li95b] Lindig, C.: Komponentensuche mit Begriffen. Proc. Softwaretechnik '95, Softwaretechnik-Trends, September 1995.

[Li98] C. Lindig: *Analyse von Softwarevarianten.* Informatik-Bericht, 98-04, TU Braunschweig, Januar 1998.

[LS97] C. Lindig, G. Snelting: Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. *Proc. International Conference on Software Engineering (ICSE 97),* Boston, USA, May 1997, pp. 349-359.

[Pa94] Parnas, D.: Software Aging. Proc. 16th International Conference on Software Engineering, IEEE 1994, pp. 279–290.

[Sn95] G. Snelting: *Reengineering of Configurations Based on Mathematical Concept Analysis. ACM Transactions on Software Engineering and Methodology* 5(2), 146-189, April 1996.

[SGS91] Snelting, G., Grosch, F.-J., Schroeder, U.: Inference-Based Support for Programming in the Large. Proc. 3rd European Software Engineering Conference, Milano 1991. LNCS 550, pp. 396 – 408.

[SFGKZ94] Snelting, G., Fischer, B., Grosch, F.-J., Kievernagel, M., Zeller, A.: Die inferenzbasierte Softwareentwicklungsumgebung NORA. Informatik – Forschung und Entwicklung 9(3). pp. 116 – 131, September 1994.

[Sun90] Sun Microsystems. *SunOS Reference Manual.* Sun Microsystems, Inc., SunOS 4.1.1, 1990.

[ZS95] A. Zeller, G. Snelting: *Handling Version Sets through Feature Logic.* W. Schäfer, P. Botella (Eds.), *Proc. 5th European Software Engineering Conference*, Sitges, Spain, September 1995, vol. 989 of LNCS, pp. 191-204.

[Ze95] A. Zeller: *A Unified Version Model for Configuration Management.* Gail Kaiser (Ed.), *Proc. ACM SIGSOFT '95 Symposium on the Foundations of Software Engineering (FSE-3),* Washington, DC, October 1995, pp. 151-160.