Die inferenzbasierte Softwareentwicklungsumgebung NORA

Gregor Snelting,
Bernd Fischer, Franz-Josef Grosch,
Matthias Kievernagel, Andreas Zeller

Informatik-Bericht Nr. 93–09
Oktober 1993
überarbeitet Februar 1994

Technische Universität Braunschweig
Gaußstraße 17
D-38092 Braunschweig
Germany

Die inferenzbasierte Softwareentwicklungsumgebung NORA

Gregor Snelting, Bernd Fischer, Franz-Josef Grosch, Matthias Kievernagel, Andreas Zeller
Arbeitsgruppe Softwaretechnologie
Technische Universität Braunschweig
Gaußstraße 17, D-38092 Braunschweig
Telefon: 0531/391–7577
Telefax: 0531/391–8111

Zusammenfassung. Wiederverwendung von Altsoftware – ebenso wie frühzeitige Qualitätssicherung bei Neuentwicklungen – erfordert komplexe Analysen und intelligente Werkzeuge. Die experimentelle Softwareentwicklungsumgebung NORA strebt deshalb die Nutzbarmachung neuer Erkenntnisse im Bereich Unifikationstheorie und Deduktionsverfahren an. Gruppiert um eine Bibliothek wiederverwendbarer Softwarekomponenten bietet NORA interaktive Werkzeuge zur inferenzbasierten Schnittstellenprüfung, zum Komponentenretrieval mit Spezifikationen in Form von Vor-/Nachbedingungen nebst Typschemata, zum unifikationsbasierten Konfigurationsmanagement sowie zur Inferenz von Varianten- und Konfigurationsstrukturen aus existierenden Quelltexten. NORA ist mit sprachspezifischem Wissen parametrisiert und kann unvollständige oder inkonsistente Information handhaben. Der Aufsatz beschreibt die Werkzeuge und die verwendeten Inferenzverfahren; abschließend werden die Systemarchitektur und die Kommunikation zwischen den Werkzeugen skizziert.

Schlüsselworte: Module und Schnittstellen, Softwarebibliotheken, Restrukturierung, Versionskontrolle, Konfigurationsmanagement, Deduktion und automatisches Beweisen, Wissensrepräsentation, Komponentenretrieval.

Abstract. Reuse of old software, as well as early quality assurance during new developments, requires complex analysis methods and intelligent tools. Therefore, the experimental software development environment NORA aims to realize recent achievements from unification theory and deduction methods. NORA presents a set of interactive tools grouped around a library of reusable components. This set of tools includes inference-based interface analysis, component retrieval using pre- and post-conditions and type signatures as search keys, software configuration management based on feature unification, and inference of configuration structures from existing source code. NORA is parametrized with language-specific knowledge and can handle incomplete and inconsistent information. The paper describes the tools and the inference methods involved; finally, the system architecture and tool communication are outlined.

Key words: Modules and Interfaces, Software Libraries, Restructuring, Version Control, Software Configuration Management, Deduction and Theorem Proving, Knowledge Representation Formalisms and Methods, Information Search and Retrieval.

Computing Reviews Classification: D.2.2, D.2.6, D.2.7, D.2.9, I.2.3, I.2.4, H.3.3p.

1 Einleitung

Das Erstellen von neuen Software-Systemen wird zunehmend durch den wirtschaftlichen Zwang zum Wiederverwenden und Weiterentwickeln existierender Software geprägt. Diese Software ist jedoch häufig unter anderen Prämissen und ohne die dafür nötige Systematik entwickelt worden. Es ist deshalb notwendig, ihr Wiederverwendungspotential erst zu erschließen; dazu sind aufwendige Analysen notwendig. Aber auch in der eigentlichen Entwicklungsphase sind Analyse- und Entwicklungswerkzeuge hilfreich, die nicht nur den Entwickler entlasten, sondern auch eine frühzeitige Qualitätssicherung erlauben.

Für solche Analyseverfahren ist jedoch eine gewisse Intelligenz der eingesetzten Werkzeuge nötig. Denn schließlich sollen sie auch bei *unvollständiger* oder *widersprüchlicher* Information sinnvolle Ergebnisse liefern – dies ist gerade im Entwurfsstadium die typische Situation. Aus dieser Motivation heraus entwickelt die Arbeitsgruppe Softwaretechnologie an der TU Braunschweig die inferenzbasierte Softwareentwicklungsumgebung NORA. NORA erschließt Wiederverwendungspotential durch

- *Erhöhen der Granularität:* Zu große oder unstrukturierte Komponenten können durch das Inferieren von Schnittstellen in beliebige Unterkomponenten aufgebrochen werden.
- *Verbessern der Einsatzmöglichkeiten:* Ursprünglich monomorph konzipierte Komponenten können sehr wohl in verschiedenen Kontexten verwendbar sein; der Polymorphismus von Komponenten wird gleichfalls durch die Schnittstelleninferenz erkannt und kontrolliert.
- *Verbessern des Zugriffs:* Durch leistungsfähige Retrieval-Werkzeuge, die Signaturen und Spezifikationen als Suchmuster erlauben, kann man gewünschte Komponenten finden, ohne auf das starre Begriffsgerüst konventioneller Klassifikationsverfahren Rücksicht nehmen zu müssen.
- *Erkennen von Irregularitäten:* Aus dem Quelltext kann die Varianten- und Konfigurationsstrukturen eines Systems inferiert und übersichtlich dargestellt werden; Abhängigkeiten und Interferenzen zwischen Konfigurationspfaden werden aufgedeckt.

NORA unterstützt die Entwicklung durch

- *frühzeitige Entwurfsprüfung:* Systementwürfe können als beliebig unvollständige Module oder Programme bereits in der Implementierungssprache formuliert und dann verfeinert werden; die Schnittstelleninferenz entdeckt Inkonsistenzen trotz der Unvollständigkeit und liefert so sehr früh Aussagen über die Realisierbarkeit bzw. Schwächen des Entwurfes.
- *automatische Wiederverwendungsvorschläge:* Die für die unvollständigen Komponenten inferierten Schnittstellen können als Suchschlüssel für das Komponenten-Retrieval verwendet werden; dadurch werden mögliche Wiederverwendungskandidaten bereits zu einem Zeitpunkt in der Bibliothek gefunden, zu dem die weitere Planung noch darauf reagieren kann.
- *vereinfachte Konfiguration:* NORAs interaktives Konfigurationsmanagement kann viele Konfigurationsmerkmale selbst erschließen, ohne daß der Benutzer eine Spezifikation angeben muß. Inkonsistenzen werden entdeckt, auch wenn eine Konfiguration noch unvollständig ist, wodurch schon während der Entwicklung Fehler in der Konfigurationsstruktur aufgedeckt werden.

NORA stellt damit nicht nur Werkzeuge zur Verfügung, sondern transferiert auch Verfahren aus dem Bereich der Deduktionssysteme oder der Unifikationstheorie in die Praxis der Software-Entwicklung und trägt so dazu bei, die Kluft zwischen Inferenzforschern und Softwareingenieuren zu überwinden.

NORA ist ein junges Projekt, das bei weitem noch nicht abgeschlossen ist. Dementsprechend hat dieser Artikel Übersichtscharakter und soll in erster Linie die grundlegenden Ideen vermitteln; mehr technische Beschreibungen sind anderswo zu finden [29, 30, 13, 16, 15, 34].

2 Softwarekomponenten in NORA

NORAs Werkzeuge gruppieren sich um eine Bibliothek wiederverwendbarer Softwarekomponenten; dabei streben wir eine möglichst einfache Integration in die bestehende UNIX-Landschaft an. NORA ist sprachunabhängig intendiert, d.h. alles, was eventuell an sprachspezifischer Information benötigt wird, wird als Parameter in die einzelnen Werkzeuge gesteckt. Unser Ziel ist es, Softwareentwicklung sowohl mit klassisch-prozeduralen als auch mit objektorientierten und funktionalen Sprachen zu unterstützen; im Moment beschränken wir uns allerdings auf Modula-2 als Studienobjekt.

In NORA können nicht nur Module, sondern auch andere syntaktische Einheiten wie Prozeduren oder Anweisungen eine eigenständige Komponente bilden, die von anderen Komponenten verwendet werden kann. Der "Genotyp" einer Softwarekomponente ist dabei stets ihr Quelltext, der als gewöhnliche UNIX-Datei vorliegt. "Phänotypen" wie abstrakter Syntaxbaum, inferierte Information wie Abhängigkeitsgraphen, Signaturen oder zusätzliche Spezifikationen etwa in Form von Vor-/Nachbedingungen werden separat gespeichert.

Abbildung 1 zeigt ein Beispiel einer Bibliothek wiederverwendbarer Softwarekomponenten, wie sie von NORAs Systemeditor dargestellt wird. Softwarekomponenten werden als Kästchen dargestellt; Abhängigkeiten zwischen ihnen werden durch Linien dargestellt. Der Abhängigkeitsgraph wird mit Hilfe sprachspezifischer Regeln errechnet, so daß keine Spezifikation notwendig ist. Durch Anklicken einer Komponente wird ein Menü der verfügbaren Werkzeuge aktiviert. Eine Komponente kann editiert werden (Konstruktion und Modifikation von Komponenten fällt nicht in NORAs Aufgabenbereich, so daß jeder seinen Lieblingseditor verwenden kann). Eine Komponente kann auch syntaktisch oder semantisch analysiert werden. Zu einer Komponente kann Schnittstelleninformation angezeigt werden; Schnittstelleninkonsistenzen werden durch spezielle Darstellung von Kanten im Graphen angedeutet. Komponenten können gesucht werden, indem etwa aus Verwendungen einer Funktion inferierte Typschemata als Suchschlüssel verwendet werden. Zu jeder Komponente kann ein Varianten-Panel aktiviert werden, das die verfügbaren Varianten anzeigt und die Konfiguration eines Systems erlaubt. Schließlich ist es möglich, die Konfigurationsstruktur existierender Quelltexte zu inferieren.

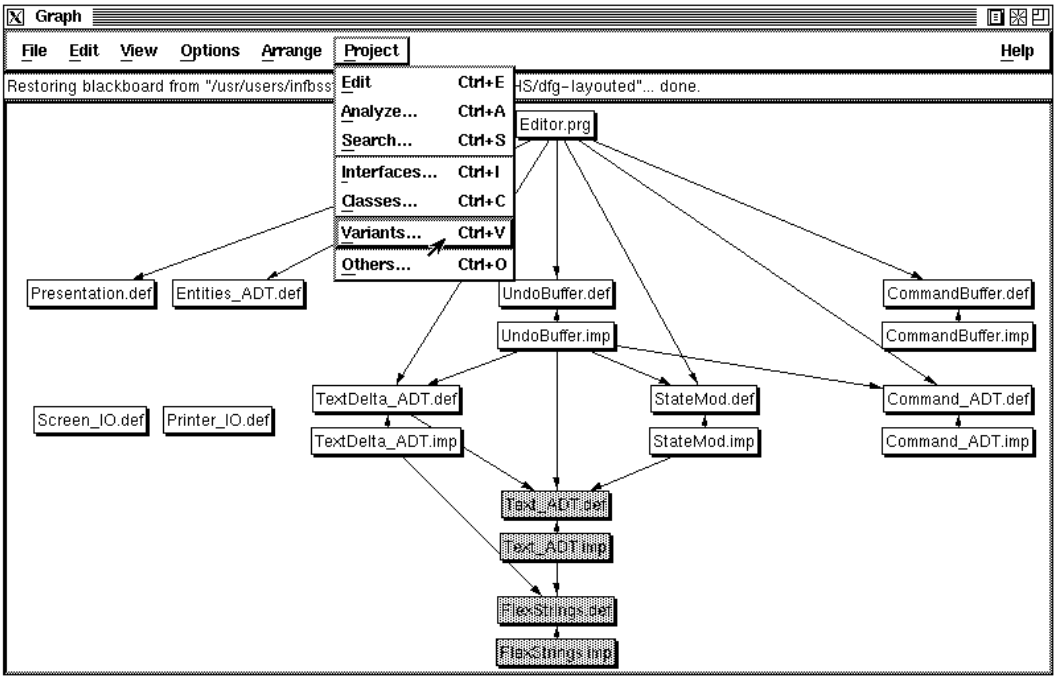


Abbildung 1: NORA Systemeditor – Modulabhängigkeitsgraph

3 Schnittstelleninferenz und polymorphe Komponenten

In prozeduralen Sprachen werden Softwarekomponenten und Übersetzungseinheiten oft gleichgesetzt. NORA geht über diese von den Fähigkeiten der Compiler bestimmte Sichtweise hinaus. Softwarekomponenten in NORA sind beliebige syntaktisch korrekte Code-Stücke; sie dienen zusätzlich zur Repräsentation von Programmgerüsten, generischen Datentypen, speziellen Algorithmen und Systementwürfen in der angestrebten Zielsprache. Solche Softwarekomponenten sind häufig unvollständig und daher nicht übersetzbar. Trotzdem wird die statische Korrektheit dieser unvollständigen Komponenten von NORA garantiert: Sie sind mehr als syntaktische korrekte Templates und viel mehr als reine Text-Makros.

NORA analysiert einzelne Bibliothekskomponenten nach dem Verfahren der Kontextrelationen [29]; der abstrakte Syntaxbaum muß dafür vorhanden sein. Syntaxanalyse und Baumaufbau werden in NORA aus Sprachbeschreibungen generiert, die im Rahmen des PSG-Projektes entwickelt wurden [2]. Kontextrelationen, gleichfalls für PSG entwickelt, können als Verfahren zur generischen, inkrementellen Typinferenz charakterisiert werden. Syntaktisch korrekte Komponenten werden trotz fehlender Deklarationen analysiert, interne Inkonsistenzen werden erkannt und angezeigt; das Ergebnis ist eine Signatur (Schnittstelle), die die nach außen sichtbaren Bezeichner enthält zusammen mit den inferierten statischen Eigenschaften wie Typ und Art des Objekts.

Softwarekomponenten in NORA können nicht-deklarierte Bezeichner enthalten; diese freien Bezeichner erlauben die Entwicklung *polymorpher* Komponenten, die bei Verwendung in verschiedenen Kontexten verschieden instantiiert werden können. Der mögliche Polymorphismus wird in der Signatur repräsentiert, so daß keine mehrfachen Neuanalysen notwendig sind, um mehrfache Verwendungen auf ihre Korrektheit zu überprüfen. Dieses Vorgehen ist vergleichbar mit dem parametrischen Polymorphismus [20], wie er in allen modernen funktionalen Sprachen benutzt wird. Man erschließt damit das beträchtliche Wiederverwendungspotential polymorpher Softwarekomponenten für monomorphe prozedurale Sprachen, ohne die Sicherheit strenger Typisierung aufzugeben [13].

Anhand eines kleinen Beispiels wollen wir nun Schnittstelleninferenz und polymorphe Komponenten erläutern. Art und Typ eines Objekts werden für die Schnittstelleninferenz in Termen repräsentiert. Der Term:

$$object(procedure, func([val\_parm(\alpha), val\_parm(\alpha)], bool))$$

beschreibt eine Funktionsprozedur, die zwei Wertparameter eines Typs α nimmt und einen booleschen Ergebniswert zurückgibt; dabei steht α für eine Variable, die mit passenden Typen instantiiert werden kann.

Ein typisches Beispiel für eine Softwarekomponente in NORA ist die folgende, angedeutete `quicksort`-Prozedur. Ein Compiler kann diese isolierte Prozedur nicht übersetzen; eine statische Analyse ist dennoch möglich:

```
PROCEDURE quicksort(VAR a: ARRAY OF elemtype);
... BEGIN  ... IF greater(a[i],a[j]) THEN ...  ... END;
```

Die Komponente benutzt die beiden nicht-deklarierten Bezeichner `elemtype` und `greater` und ist daher für verschiedene Elementtypen benutzbar, solange eine geeignete Vergleichsprozedur zur Verfügung steht. Die Analyse inferiert für diese Komponente folgende Signatur:

$$\begin{aligned} \forall\alpha : type.\ [elemtype :\ & object(type, \alpha), \\ greater :\ & object(procedure, func([parm(\alpha), parm(\alpha)], bool)), \\ quicksort :\ & object(procedure, func([ref\_parm(array\_type(int, \alpha))], \_))] \end{aligned}$$

Tatsächlich handelt es sich bei dieser Signatur um ein *Signaturschema*: Es enthält die allquantifizierte Variable α, die in verschiedenen Anwendungskontexten verschieden instantiiert werden kann. Diese Prozedur ist insofern polymorph, als der Code für *jeden* Elementtyp benutzt werden kann (allerdings muß zusätzlich eine passende Vergleichsprozedur vorhanden sein).

Um ein System zu montieren, unterstützt NORA die Mechanismen der Zielsprache, darüberhinaus steht orthogonal dazu ein sprachunabhängiger Makromechanismus zur Verfügung. In unserem Studienobjekt Modula-2 gibt es ein einfaches Modulsystem: Sowohl die korrekte Verwendung eines importierten Moduls als auch die Konsistenz zwischen Implementierungs- und Definitionsmodul werden anhand der inferierten Signaturen überprüft. Der Makromechanismus erlaubt es, Komponenten, die keine Übersetzungseinheiten sind, zusammenzusetzen. Zur Überprüfung eines Systems ist keine Makroexpansion notwendig, die Analyse erfolgt ausschliesslich anhand der inferierten Signaturen.

Die `quicksort`-Komponente unseres Beispiels kann nun durch ein einfaches `#include ...` in anderen Komponenten verwendet werden. Nicht-deklarierte Bezeichner, die den Polymorphismus der Komponente induzieren, können dabei von außen instantiiert werden, um die Bezeichnerbindung genau zu kontrollieren. In der folgenden Komponente wird `quicksort` benutzt, um ein Feld von `INTEGER`-Werten zu sortieren:

```
MODULE sortInt;
  TYPE
    value = INTEGER;
  VAR
    a: ARRAY [1 .. 117] OF value;
  PROCEDURE greater(VAR res: BOOLEAN; x, y: elemType);
  BEGIN res := x > y;
  END greater;
#include quicksort.comp (elemType = value)
BEGIN ...  quicksort(a); ...
END sortInt.
```

Im `#include` wird dem freien Bezeichner `elemType` der aktuelle Bezeichner `value` zugeordnet; der freie Bezeichner `greater` wird direkt gebunden. Die Analyse meldet jetzt einen Fehler, da die Vergleichsprozedur `greater` keine Funktionsprozedur ist und daher nicht zur Signatur der benutzten Komponente paßt – dieser Fehler ist allerdings vom Programmierer leicht zu beheben, indem die Verwendung der Deklaration angepaßt wird.

Eine völlig andere Aufgabe wird durch das Sortieren eines “Waldes” gelöst:

```
MODULE sorttrees;
  TYPE
    tree = POINTER TO treeElem;
    treeElem = ... ;
  VAR
    forest: ARRAY [0 .. 44] OF tree;
  PROCEDURE bigger(tree1, tree2: tree): BOOLEAN;
    ...
#include quicksort.comp (elemType = tree, greater = bigger)
BEGIN ... quicksort(forest); ...
END sorttrees.
```

Der Elementtyp wird hier mit dem Typ `tree` und die Vergleichsprozedur mit der Prozedur `bigger` instantiiert. Es sei nochmals bemerkt, daß lediglich die Signaturschemata benutzt werden, um korrekte Verwendung von Komponenten sicherzustellen. Mehrfache Verwendungen einer Komponente erfordern deshalb keine mehrfachen Neuanalysen. Der für die Analyse notwendige Abhängigkeitsgraph wird automatisch berechnet. Signaturschemata dokumentieren die Verwendungsmöglichkeiten auch für den Programmierer und unterstützen das Retrieval von Komponenten.

Die Analyse einzelner Komponenten mit Kontextrelationen und das Berechnen der Signaturen ist sehr ähnlich zur Typinferenz in funktionalen Sprachen. Die Generalisierung von Signaturen zu Signaturschemata ist eine Übertragung der Vorgehensweise beim let-Polymorphismus nach Damas und Milner [5]. Falls die inferierte Signatur keine freien Typvariablen enthält, sollte es sogar möglich sein, die Komponente in Abwesenheit des globalen Kontextes zu übersetzen ("Smartest Recompilation", vgl. [28]). Zwar mag es aus methodischen Gründen besser sein, die Deklaration verwendeter (polymorpher) Komponenten zu erzwingen – wie dies ja auch z.B. in ADA verlangt wird. Wir wollen aber gerade zeigen, daß man durch Anwendung von Typinferenz polymorphe Komponenten sogar für solche Sprachen erhalten kann, die Polymorphismus eigentlich nicht kennen.

4 Komponentenretrieval mit Spezifikationen

NORA verwendet zwei neuartige Verfahren zum Komponentenretrieval:

- Komponentensuche mit Typ-/Signaturschemata
- Komponentensuche mit Vor-/Nachbedingungen.

Beide Ansätze wurden bisher überwiegend für funktionale Sprachen angegangen [25, 26], wegen der hohen Komplexität aber kaum für prozedurale Sprachen. Vorhandene Retrieval-Ansätze für prozedurale Sprachen, die über manuelle Klassifikation hinausgehen, erzeugen etwa aus Softwarekomponenten Kontrollflußskelette, die bei der Suche instantiiert werden können [6]; wir streben demgegenüber eine Komponentenbeschreibung an, die keine Realisierungsinterna verrät.

In NORA kann der Benutzer etwa eine nichtdeklarierte Funktion anklicken, von der er hofft, eine verwendbare Implementierung in der Bibliothek zu finden. NORA prüft zunächst, ob es in der Bibliothek Objekte mit gleicher oder ähnlicher Signatur gibt. Dazu werden die für die Verwendungsstelle inferierten Typen mit Typschemata von Bibliotheksobjekten unifiziert; falls dies nicht fehlschlägt, gilt das Objekt als gefunden. Matching reicht zur Suche nicht aus, da Bibliothekskomponenten unvollständig sein können! Da Benutzer oft die Reihenfolge von Parametern vergessen, wird beim Suchen für die Unifikation bestimmter Teilattribute (z.B. Parameterlisten) AC1-Unifikation [8] verwendet; diese betrachtet Listen, die sich nur durch Elementvertauschung unterscheiden, als äquivalent.

Für Sprachen wie Modula-2 ist die Menge der so gefundenen Funktionen meist viel zu groß (z.B. berichtet [17], daß in seiner Bibliothek die Signatur $s \times i \rightarrow s$ 90–mal vorkommt). Der Benutzer kann deshalb ergänzend eine (partielle) *Spezifikation* der Komponente als zusätzlichen Suchschlüssel verwenden. Spezifikationen werden wie üblich durch Vor-/Nachbedingungen angegeben, also etwa in der Form (P, Q). Eine Komponente K, die die (abgespeicherte) Spezifikation $\{P'\}\ K\ \{Q'\}$ erfüllt, gilt als gefunden, falls $(P \Rightarrow P') \wedge (Q' \Rightarrow Q)$. Diese letzte Eigenschaft muß mit leistungsfähigen *Deduktionsverfahren* geprüft werden.

Um konkret die Vorteile aufzuzeigen, die die Hinzunahme der Vor- und Nachbedingungen zur Suchinformation bringt, soll als Beispiel die polymorphe Sortierprozedur 'quicksort' aus dem letzten Abschnitt wieder aufgegriffen werden. Die semantische Analyse lieferte für diese Prozedur das

Typschema (s.o.):

$$object(procedure, func([ref\_parm(array\_type(int, \alpha))], \_))$$

Zu den Ergebnissen, die man bei der Suche nach diesem Typ erhält, gehören allerdings sehr viele irrelevante Komponenten, wie etwa 'to_lower_case' und 'to_upper_case' mit dem spezielleren Typschema

$$object(procedure, func([ref\_parm(array\_type(int, char))], \_))$$

welches man durch passendes Instantiieren der Typvariablen α der Anfrage erhält. Erst ein Matching mit einer Nachbedingung der Form

$$\{\forall i, j : i < j \Rightarrow a[i] \leq a[j]\}$$

liefert die tatsächlich interessanten Prozeduren: 'bubblesort', 'shellsort', 'unique_sort', 'quicksort'. Durch die stärkere Nachbedingung

$$\{\forall i, j : i < j \Rightarrow a[i] < a[j]\}$$

läßt sich hieraus noch 'unique_sort' selektieren, welches zusätzlich Duplikate eliminiert. Die anderen Prozeduren sind auch bezüglich ihrer Vor- und Nachbedingungen ununterscheidbar.

Wie zu Anfang des Abschnitts bereits angedeutet, verwenden wir zwei Suchphasen. Als schneller grober Filter wird das Suchen mit Typschemata eingesetzt (Phase 1). Auch die Suche nach Bedingungen (Phase 2) kann noch einmal gestaffelt werden, indem die Suchbedingungen sukzessive stärker gemacht werden. Die konkrete Notation zur Formulierung von Anfragen muß nicht abstrakte Terme benutzen, sondern kann auch an die Programmiersprache (Modula-2) angelehnt sein:

$$\texttt{PROCEDURE}\ (\texttt{REF PARAMETER} : \texttt{ARRAY OF}\ \alpha : type) : \_$$

Als Beschreibungsmittel für Spezifikationen verwenden wir die modellorientierte Spezifikationssprache VDM [14]. VDM bietet eine direkte Kennzeichnung der Vor- und Nachbedingungen zu spezifizierten Komponenten. Als Studienobjekt dient uns eine allgemein verbreitete Bibliothek von Modula-2–Komponenten [17]. Diese wurde nachträglich mit VDM spezifiziert. Aus den Komponentenspezifikationen wird der zur Suche verwendete Katalog extrahiert, wobei zusätzlich die Vor- und Nachbedingungen in die Sprache des verwendeten Deduktionsbeweisers übersetzt werden müssen. Als Beweiser wird im Moment das zur Zeit weltweit leistungsfähigste System OTTER [19] verwendet.

Was ist nun der Vorteil unseres Ansatzes gegenüber einfacheren Verfahren, die mit starren Klassifikationsschemata auskommen, etwa Facettenklassifikation [23]? Der Hauptvorteil ist, daß der Benutzer völlige Freiheit in der Formulierung seiner Anfrage hat und sich nicht an ein vorgegebenes Klassifikationsschema halten muß. Diese Freiheit ist aber auch die Quelle hoher Komplexität des Verfahrens. In der Tat wurde Komponentensuche mit Vor- und Nachbedingungen schon früher vorgeschlagen (z.B. in [26]), scheiterte aber in der Praxis an der mangelnden Leistungsfähigkeit vorhandener Deduktionssysteme: OTTER konnte in von uns durchgeführten Experimenten nur einfachste Retrievalaufgaben bewältigen. Der Grund ist, daß der Benutzer Spezifikationen angeben kann, die sich syntaktisch von der abgespeicherten stark unterscheiden, wenngleich sie semantisch äquivalent sein mögen. In solchen Fällen wird es für den Beweiser schwierig.

Das zur Zeit laufende DFG-Schwerpunktprogramm "Deduktion" [3] zielt u.a. auf die Entwicklung von konfigurierbaren, dedizierten Hochleistungstheorembeweisern. Es ist zu erwarten, daß durch den Einsatz dieser Beweiser die Schwierigkeiten mit klassischen Deduktionsbeweisern überwunden werden können [15].

5 Interaktives, inferenzbasiertes Konfigurationsmanagement

5.1 Überblick

Die meisten der "klassischen" Verfahren zum Konfigurationsmanagement sind Erweiterungen von *MAKE* [10] oder *RCS* [31]. Das *shape*-System [18] etwa erlaubt es, Varianten in erweiterten MAKE-Regeln anzugeben. All diese Arbeiten legen ihren Schwerpunkt auf Verwaltung (d.h. Speicherung und Retrieval) sowie das tatsächliche Zusammensetzen einer Konfiguration und sind weder inferenzbasiert noch interaktiv. Jüngere Systeme favorisieren logikbasierte Modelle, insbesondere zum Planen [24] oder Zusammenstellen [21] einer Konfiguration, unterstützen aber keine inkrementelle oder interaktive Vorgehensweise und erfordern vollständige Spezifikationen. Das Konfigurationsmanagement von NORA hingegen

- erlaubt die *Klassifikation* von Komponenten anhand ihrer Eigenschaften,
- erlaubt *unvollständige* und *mehrdeutige* Spezifikationen,
- vereinheitlicht *Revisions-* und *Variantenmanagement*,
- weist den Benutzer frühzeitig auf *Konfigurations-Konflikte* hin,
- läßt den Benutzer *interaktiv* und *inkrementell* aus der Menge der möglichen Konfigurationen seine Auswahl treffen,
- verschont den Benutzer von Spezifikationen, die vom System selbst inferiert werden können – so etwa der Abhängigkeitsgraph oder bestimmte Konfigurationsmerkmale.

5.2 Features und Feature-Terme

Klassischerweise werden Varianten durch individuelle *Attribute* unterschieden. Ein Attribut ist dabei ein Paar $(name, wert)$. Jede Variante wird eindeutig durch eine Menge von Attributen identifiziert, was die Verwaltung wesentlich vereinfacht. Solche Ansätze finden wir im C-Präprozessor oder im *attributierten Dateisystem* von Shape [18], die, dem einfachen Modell entsprechend, den Schwerpunkt auf das (platzeffiziente) Speichern und das spätere Retrieval legen.

Die in der Wissensrepräsentation entwickelte *Feature-Logik* [1, 27] erlaubt es, Aussagen über Attribute zu treffen, die *Feature-Terme* genannt werden. Ein Feature ist (wie auch ein Attribut) ein Name/Wert-Paar; ein Feature-Term ist ein Ausdruck, der eine Menge von Features beschreibt. Wir unterscheiden

- *Selektion.* Das Auftreten eines Features wird durch ein Paar $name : wert$ dargestellt. *wert* kann dabei ein Atom, Literal oder auch ein Feature-Term sein. Als Werte sind auch *Variablen* zulässig, die durch Großbuchstaben gekennzeichnet werden. Der besondere Wert $\top$ ("Top") steht für die Menge aller Werte.
- *Konjunktion.* Das gemeinsame Auftreten von Features wird durch eckige Klammern dargestellt: $[operating\text{-}system : dos, concurrent : false]$.
- *Negation.* Ausgeschlossene Features werden durch das Negations-Zeichen $\sim$ gekennzeichnet.[1]
- *Disjunktion.* Alternativ auftretende Features werden in geschweiften Klammern dargestellt, etwa als

$$\left[operating\text{-}system : unix, \left\{ \begin{array}{l} [concurrent : true], \\ [concurrent : false] \end{array} \right\}\right]$$

  Wie das Beispiel zeigt, können Feature-Terme beliebig geschachtelt werden.

[1] Hierbei muß zwischen dem Nicht-Auftreten eines Attributs $(\sim attribut : wert)$ und dem Nicht-Auftreten eines Wertes $(attribut : \sim wert)$ unterschieden werden. Der erste Ausdruck subsumiert alle Feature-Terme, in denen das Attribut mit dem gegebenen Wert nicht auftritt; der zweite Ausdruck steht für die Terme, in denen das Attribut einen anderen als den gegebenen Wert aufweist.

Zwei Eigenschaften von Feature-Termen sollten besonders hervorgehoben werden:

- *Jedem Feature kann nur ein Wert zugeordnet sein.* Der Term $[\mathit{state}: \mathit{buggy}, \mathit{state}: \mathit{fixed}]$ ist daher ungültig. Wir verwenden diese Eigenschaft zur Konsistenzprüfung.
- *Feature-Terme sind immer unvollständige Beschreibungen.* Der Term $[\mathit{farben}: 2]$ sagt nichts über das Auftreten oder Nicht-Auftreten weiterer Eigenschaften neben *farben* aus. Dies erlaubt uns inkrementelle Verfeinerung der Konfigurations-Möglichkeiten.

5.3 Features von Varianten, Komponenten und Systemen

Im folgenden gehen wir davon aus, daß jede Variante v mit einem Feature-Term $\mathit{conf}(v)$ versehen ist, der die Eigenschaften (oder Nicht-Eigenschaften) der Variante beschreibt, und daß jede Variante über $\mathit{conf}(v)$ eindeutig identifiziert werden kann. Die Eigenschaften von Varianten bestimmen die Eigenschaften von Komponenten und von Systemen wie folgt: Liegt eine Komponente k in verschiedenen Varianten $V = \{k', k'', k''', \ldots\}$ vor, werden alle möglichen Konfigurationen durch *Generalisierung* über V beschrieben:

$$\mathit{conf}(k) = \bigsqcup_{v \in V} \mathit{conf}(v)$$

Die Generalisierung $S \sqcup T$ liefert die Disjunktion von S und T; jede neue Variante erweitert die Menge der möglichen Konfigurationen.

Wird ein System aus einzelnen Komponenten $K = \{k_1, k_2, \ldots\}$ zusammengesetzt, erhält man die möglichen Konfigurationen durch die *Feature-Unifikation* über K:

$$\mathit{conf}(K) = \bigsqcap_{k_i \in K} \mathit{conf}(k_i)$$

Die Unifikation $S \sqcap T$ liefert die Konjunktion von S und T, wobei $S \sqcap T = \bot$, wenn die Unifikation fehlschlägt. Jede hinzukommende Komponente schränkt die Menge der möglichen Konfigurationen ein. Ist $\mathit{conf}(K) = \bot$, existiert keine gültige Konfiguration.

Optionale Komponenten werden modelliert, indem für eine optionale Komponente k neben $\mathit{conf}(k)$ die Negation $\sim\mathit{conf}(k)$ für die Abwesenheit von k zur Auswahl zugelassen ist. Hiermit kann die Komponentenmenge in Abhängigkeit der Eigenschaften eindeutig bestimmt werden.

Da wir eine beliebige Untermenge der Komponentenmenge wählen können, erlaubt Feature-Logik *inkrementelle Konfiguration*, bei der Stück für Stück die Menge der Möglichkeiten eingeschränkt wird. Insbesondere beeinflußt die Konfiguration einer Komponente die Konfigurationsmöglichkeiten an anderer Stelle, wie wir in Abschnitt 5.5 zeigen werden.

5.4 Ein Beispiel

Als Beispiel betrachten wir ein modernes Textverarbeitungsprogramm, das Grafik und Text kombinieren kann. Das Programm unterstützt verschiedene Betriebssysteme und verschiedene Bildschirmtypen. Wir können wählen zwischen zwei Betriebssystemen (*dos* und *unix*), vier Bildschirmtypen (*ega*, *tty*, *x11* und *news*) sowie zwei Bildschirmtreibern (*dumb* und *ghostscript*). Der *dumb*-Bildschirmtreiber nimmt an, daß der Bildschirm die anfallenden Daten direkt verarbeiten kann; der *ghostscript*-Bildschirmtreiber kommuniziert mit einem separaten Prozeß, der *postscript*-Daten in *bitmap*-Daten umwandeln kann. Alle Komponenten mit ihren jeweiligen Eigenschaften sind in Abbildung 2 dargestellt.

Als Beispiel wollen wir nun ein System zusammensetzen. Wir beginnen mit der Auswahl des Betriebssystems und wählen *dos* aus. Das bedeutet, daß die *x11*- und *news*-Bildschirmtypen wegfallen,

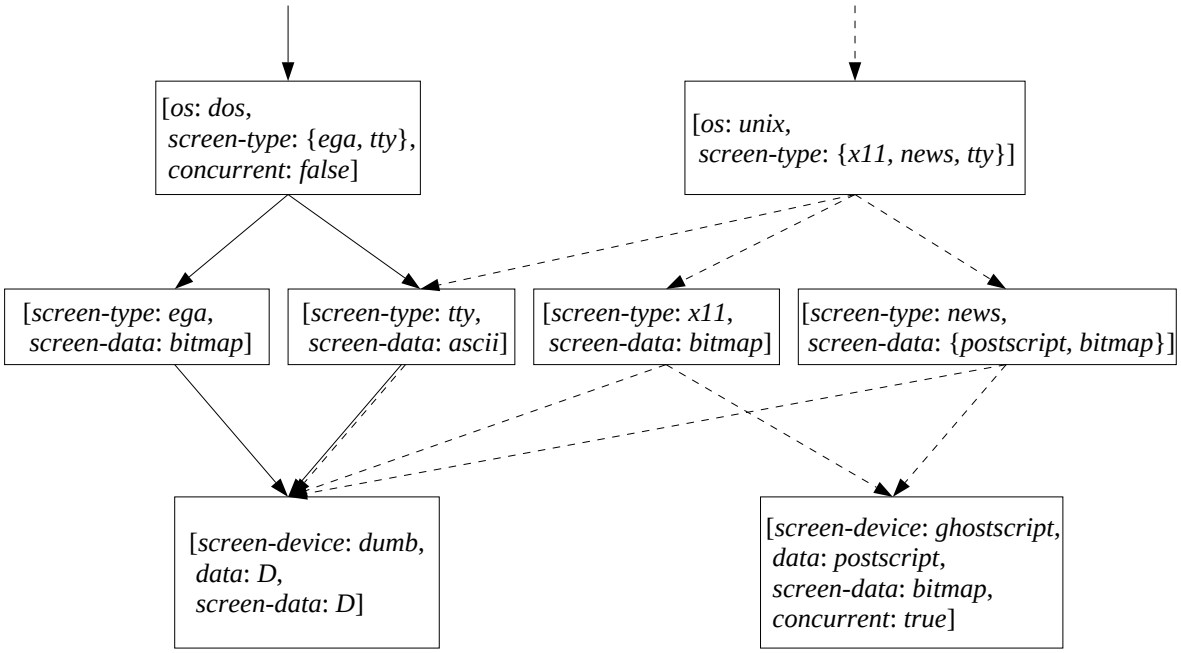


Abbildung 2: Möglichkeiten der Systemkonfiguration.
Jeder Pfad stellt eine mögliche Konfiguration dar.

da sie von *dos* (in unserem Beispiel) nicht unterstützt werden.[2] Es verbleiben *ega* und *tty*, die von *dos* unterstützt werden (und im Beispiel durch gewöhnliche Pfeile markiert sind). Nun folgt die Wahl des Bildschirmtreibers. *ghostscript* kann nicht ausgewählt werden, da er $[concurrent : true]$ voraussetzt, was *dos* aber nicht liefern kann. Es verbleibt der *dumb*-Bildschirmtreiber, wobei D je nach Bildschirmtyp zu *ascii* oder *bitmap* instantiiert wird. Die Gesamtmenge der Konfigurationen unter *dos* läßt sich damit beschreiben durch

$$\left[os : dos, \left\{ \begin{array}{l} [screen\text{-}type : ega, screen\text{-}data : bitmap, screen\text{-}device : dumb, data : bitmap], \\ [screen\text{-}type : tty, screen\text{-}data : ascii, screen\text{-}device : dumb, data : ascii] \end{array} \right\} \right]$$

Die Auswahl hätte statt mit *dos* auch mit *unix* beginnen können; die entsprechende Auswahl ist mit gestrichelten Linien dargestellt. Jeder Pfad ist eine gültige Konfiguration. Die Auswahl hätte auch mit einer anderen Komponente als dem Betriebssystem beginnen können, so z.B. mit einer Menge möglicher Bildschirmtypen (etwa $[screen\text{-}type : \{x11, tty\}]$) oder auch einer Negation (etwa $[screen\text{-}device : \sim ghostscript]$). Jede weitere Auswahl schränkt die Menge der möglichen Konfigurationen ein, bis ein einzelner Pfad gefunden ist. Dieses *inkrementelle Vorgehen* ist nicht nur von der Benutzerseite her vorzuziehen. Es hilft auch, die Komplexität zu reduzieren, denn je früher wir Alternativen ausschließen, um so weniger müssen wir sie bei der späteren Auswahl berücksichtigen.

5.5 Interaktive Auswahl von Konfigurationen

Die Terme der Feature-Logik können sehr schnell zu komplex werden, um sie vollständig anzuzeigen. Deshalb werden in NORA Konfigurationen durch *Feature-Panels* bearbeitet. Hierbei wird ausgehend von den möglichen Konfigurationen ein interaktives Panel erzeugt, aus dem dann Konfigurationsmengen per Menü oder Texteingabe ausgewählt werden (Abbildung 3).

Nach der Selektion einzelner Features werden weitere Optionen verfügbar (schwarze Schrift) oder ausgeblendet (graue Schrift). Nicht benötigte Subterme können einfach ausgeblendet werden – etwa

[2] Formal fallen *x11* und *news* heraus, da $[os : dos, screen\text{-}type : \{ega, bitmap\}] \sqcap [screen\text{-}type : \{x11, news\}] = \bot$)

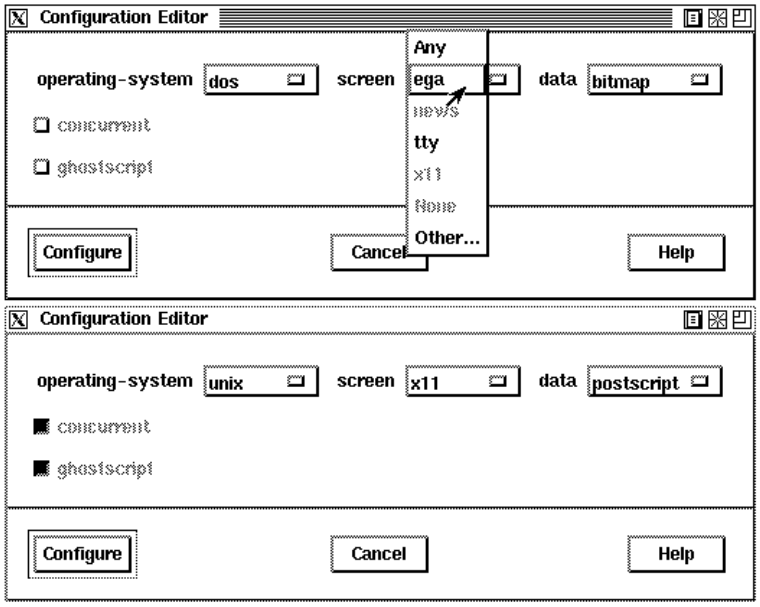


Abbildung 3: Ein Feature-Panel in zwei verschiedenen Zuständen. Bei der Variante *operating-system*:*dos* werden Optionen für *screen* ausgeblendet

Optionen für [*screen*:*news*] und [*screen*:*x11*], wenn der Benutzer bereits [*operating-system*:*dos*] gewählt hat.

Solche Panels können ebenfalls für beliebige Komponentenmengen erzeugt werden, um diese Komponenten zu konfigurieren. In Abbildung 4 unseres Beispiels hat der Benutzer den Bildschirmtreiber

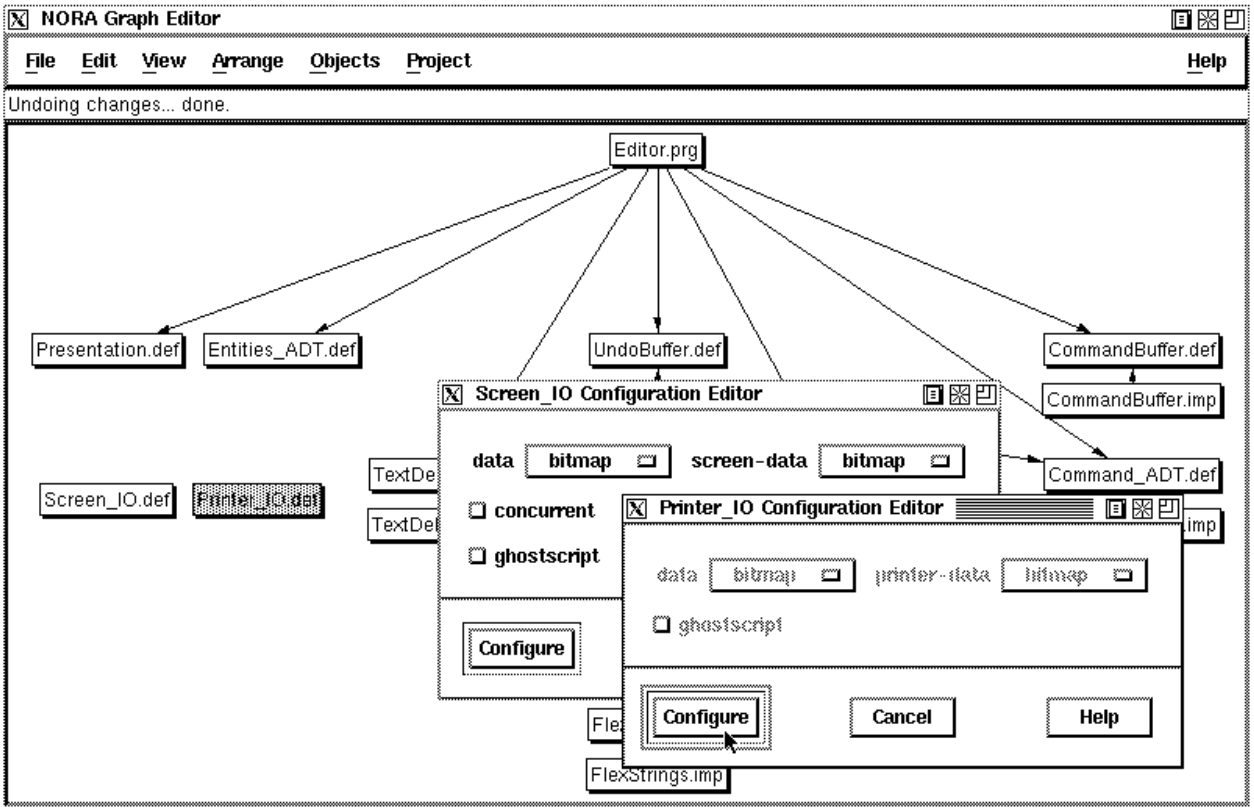


Abbildung 4: Nicht-lokale Implikationen bei der Auswahl von Konfigurationen. Die Konfiguration von *Screen-IO* schränkt die Konfiguration von *Printer-IO* auf eine einzige Variante ein

Screen-IO so konfiguriert, daß er die GhostScript-Variante ausgeschlossen hat. Das hat unmittelbare Konsequenzen auf die Konfiguration des Druckertreibers *Printer-IO*, der damit nur noch in der Variante $[\mathit{data}:\mathit{bitmap}, \mathit{printer\text{-}data}:\mathit{bitmap}]$ widerspruchsfrei zu konfigurieren ist. Mit Hilfe der Feature-Logik lassen sich, wie hier gezeigt, zahlreiche solcher nicht-trivialen, nicht-lokalen Implikationen aufdecken.

5.6 Schnittstelle zu traditionellen Techniken

Da Feature-Logik eine Obermenge der klassischen Attributierung ist, lassen sich Feature-Terme leicht aus attributierten Darstellungen gewinnen. NORA verwaltet Variantenmengen mit Hilfe von C-Präprozessor-Anweisungen, die als Feature-Terme interpretiert werden. Neben dem Vorteil der Kompatibilität zu weitverbreiteten Standards gewinnt man die Möglichkeit, individuelle *Sichten* auf Variantenmengen zu generieren.

Als Beispiel betrachten wir Abbildung 5. Das gezeigte Programmstück entstammt dem `xload`-Programm, das die Rechnerauslastung auf dem Bildschirm anzeigt. Konfigurationsspezifische Programmteile sind in `#if ... #endif` eingeklammert; beim Übersetzen werden durch das Setzen von *Präprozessorsymbolen* die richtigen Programmstücke zusammengesetzt. Das Programmstück ist hochgradig systemabhängig, wie die hohe Zahl von Präprozessor-Anweisungen zeigt – und auch reichlich unübersichtlich.

Wir können nun Präprozessor-Ausdrücke mit Hilfe der Feature-Logik *partiell auswerten* und die Variantenmenge schrittweise reduzieren.[3] Jedes Auftreten eines Präprozessorsymbols v mit dem Wert w wird als Feature-Term $[v:w]$ aufgefaßt, der dem betreffenden Textabschnitt zugeordnet ist.

[3] Das Problem geht über die partielle Auswertung Boolescher Formeln weit hinaus, da Symbole auch *undefiniert* sein können.

```
#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 5: Ausschnitt aus `xload`. Insgesamt werden 43 Präprozessorsymbole verwendet

Die in C üblichen logischen Operatoren werden in die entsprechenden Operatoren der Feature-Logik umgesetzt.

Zu Beginn finden wir den Programmtext

```
#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
    ...
```

Der Ausdruck in der ersten Zeile entspricht dem Feature-Term

$$T = [\{\sim svr4 : \top, \sim stdc : \top\}, \sim sgi : \top, \sim motorola : \top]$$

Wir stellen uns nun vor, daß wir diesen Programmtext für einen bestimmten *Kontext* betrachten wollen, etwa für *svr4*[4]. Wir können in unserem Kontext also *aixv3*[5] sowie die *sgi*- und *motorola*-Varianten ausschließen. Unser Kontext sieht dann so aus:

$$S = [svr4 : \top, \sim aixv3 : \top, \sim sgi : \top, \sim motorola : \top]$$

Die Variantenmenge wird nun mittels *Unifikation* eingeschränkt, indem wir $T \sqcap S$ für alle gefundenen Ausdrücke T bilden und das Ergebnis als neue Bedingung einsetzen. Ist $T \sqcap S = \bot$, wird der Text unterdrückt; ist $T \sqcap S = S$, ist keine besondere Präprozessor-Anweisung notwendig. Der resultierende Programmtext, der dann nur noch die Variantenmenge für den Kontext S repräsentiert, sieht dann so aus:

```
#if !defined(__STDC__)
    extern void alist();
#endif
    nlist( KERNEL_FILE, namelist);
    ...
```

Als nächstes könnte der Anwender einen geeigneten Wert für `__STDC__` wählen, um nur noch diese Variante zu bearbeiten. Hier zeigt sich inkrementelle Konfiguration von einer anderen Seite: Der Anwender kann sukzessive seine Sicht auf die Variantenmenge einengen, bis er sie genügend reduziert hat, um den entsprechenden Programmtext überschauen zu können.

5.7 Ausblick

Neben Variantenmanagement lassen sich auch *Revisionen*, d.h. Versionen, die bestehende Versionen *ersetzen* sollen, mit Hilfe von Feature-Logik verwalten. Hierzu wird die Durchführung einer Änderung als Feature (sogenanntes *Delta-Feature*) repräsentiert. Einzelne Versionen werden ausgewählt, indem die letzte Änderung, die zu dieser Version geführt hat, angegeben wird und spätere Änderungen ausgeschlossen werden. Mehr hierzu findet sich in [34].

Sehr vielversprechend ist die Integration von Informationen der Schnittstellenanalyse (Kapitel 3). Grob gesagt, werden hierzu die Signaturen als Feature-Terme kodiert, so daß jedem Objekt ein Feature zugeordnet wird, dessen Wert Art und Typ des Objektes beschreibt. Der Unifikationsmechanismus

[4] System V Release 4, eine UNIX-Variante

[5] AIX Version 3 von IBM, eine andere UNIX-Variante

wird dann um Unifikation der Typen erweitert. Auf diese Art und Weise schlägt die Unifikation bei Komponenten mit inkompatiblen Annahmen über Art und Typ der Objekte fehl; statisch inkorrekte Konfigurationen werden somit frühzeitig ausgeschlossen.

Feature-Logik kann neben Signaturschemata und Vor-/Nachbedingungen als drittes, universelles Verfahren zum Komponentenretrieval eingesetzt werden; dabei werden Feature-Terme als Suchmuster verwendet. In jüngerer Zeit sind Retrieval-Verfahren populär geworden, die ein *Ähnlichkeitsmaß* zwischen einzelnen Attributen vorsehen [23]. Wir planen, solche Verfahren auf Feature-Logik anzuwenden ("Fuzzy Feature Forms") und so Komponenten zu finden, die gewünschten Eigenschaften möglichst nahe kommen.

Zur *Konstruktion* und zur *Systemmodellierung* planen wir, spezielle Programmiersprachen einzusetzen, die auf Feature-Logik und *Feature-Resolution* [11] beruhen. Hiermit kann dann die Komponentenmenge in Abhängigkeit von den gewünschten Eigenschaften beschrieben werden. Weiterhin wird das System Anfragen der Art "Was muß neu übersetzt werden, wenn sich das Feature f der Komponente k nach f' ändert?" beantworten können und so *smart recompilation* realisieren.

Ein letztes Problem ist auch im `xload`-Beispiel deutlich geworden: Woher weiß der Anwender, daß bestimmte Eigenschaften (etwa *aixv3* und *svr4*) einander ausschließen? Wo solches Wissen nicht explizit zur Verfügung steht, muß es inferiert werden. Dies ist Thema des folgenden Kapitels.

6 Reverse Engineering von Konfigurationsstrukturen

Reverse Engineering befaßt sich mit der Rekonstruktion der Systemarchitektur bzw. allgemein mit dem Inferieren von Abstraktionen aus dem Programmtext. Mit NORA wollen wir etwas Neues angehen: die Inferenz von Varianten- und Konfigurationsstrukturen aus existierenden Quelltexten.

Eine häufig benutzte Standardtechnik zum Konfigurationsmanagement ist die Verwendung des C-Präprozessors, wie in Abschnitt 5.6 geschildert. Obwohl es inzwischen wesentlich ausgefeiltere Konfigurationssysteme gibt (siehe etwa [9]), wird eine ganze Menge existierender Software mit Hilfe des Präprozessors konfiguriert. Es wäre deshalb überaus nützlich, wenn man Software, die nach diesem Konzept erstellt wurde, wiederum mit NORA behandeln kann, um die Konfigurationsstruktur zu analysieren, zu modifizieren oder weiterzuentwickeln. Ein solches Werkzeug wäre außerdem nicht nur zur Analyse alter Software, sondern auch zur Qualitätssicherung in laufenden Projekten verwendbar.

In der Tat gibt es ein Verfahren, mit dem man aus Rohdaten wie den in unserem Falle vorliegenden tieferliegende Strukturen zurückinferieren kann, nämlich die *formale Begriffsanalyse* [33, 12]. Diese Methode wurde in zehnjähriger Arbeit am Lehrstuhl für allgemeine Algebra der TH Darmstadt entwickelt und auf so unterschiedliche Dinge wie die Analyse der Goldfischretina, das Verhalten Drogensüchtiger, die Klassifizierung endlicher Verbände, die Analyse von Werken Rembrands, und die Musikwahrnehmung von Fernsehzuschauern angewendet. Das Verfahren beruht darauf, aus Rohdaten, die in Form einer Relation zwischen Objekten und Attributen vorliegen, einen vollständigen Verband von sog. *Begriffen* zu konstruieren. Die gewonnenen Strukturen stimmen bemerkenswert mit den intuitiven Konzepten des zugrundeliegenden Problembereichs überein. Aus Platzgründen können wir auf die mathematischen Grundlagen im folgenden nur kurz eingehen.

In Programmen wie `xload` aus Abbildung 5 kann man jede Anweisung danach charakterisieren, welche Präprozessorvariablen definiert sein müssen, damit sie in einen *"configuration thread"* übernommen wird. Dies führt zu einer Tabelle wie in Abbildung 6: die Zeilen sind mit Programmabschnit-

ten (Zeilennummern) markiert, die Spalten mit Präprozessorvariablen[6]. Die Spalten sind konjunktiv verknüpft: ein Codestück wird nur dann Teil eines Configuration Thread, wenn alle angekreuzten Präprozessorsymbole definiert sind. In der Praxis tauchen nicht nur einfache Bedingungen, sondern auch Konjunktionen, Diskunktionen und Negationen von Bedingungen auf, außerdem können #ifdefs geschachtelt werden, was zu Implikationen führt. Hier müssen besondere Regeln angewandt werden, deren Darstellung wir an dieser Stelle übergehen (siehe jedoch die detaillierte Darstellung in [16]).

Die entstehende Tabelle ist eine Relation, die in diesem Zusammenhang als *formaler Kontext* bezeichnet wird. Ein formaler Kontext $C = (O, A, P)$ ist charakterisiert durch eine Menge von *Objekten* O, eine Menge von *Attributen* A und eine Relation $P \subseteq O \times A$ zwischen Objekten und Attributen; falls $(o, a) \in P$ sagt man: Objekt o hat Attribut a[7].

Für eine Menge von Objekten $X \subseteq O$ sind die *gemeinsamen Attribute* gegeben durch $\sigma(X) = \{a \in A \mid \forall o \in X : (o, a) \in P\}$. Umgekehrt sind zu einer Attributmenge $Y \subseteq A$ die *gemeinsamen Objekte* definiert durch $\tau(Y) = \{o \in O \mid \forall a \in Y : (o, a) \in P\}$. Die Abbildungen σ und τ bilden eine *Galoisverbindung* zwischen Objekten und Attributen, mit allen dafür charakteristischen Eigenschaften.

Ein (formaler) *Begriff* ist nun ein Paar von Objektmengen und Attributmengen (X, Y) derart, daß $Y = \sigma(X)$, $X = \tau(Y)$. Die Menge aller Begriffe zu einem Kontext $B(O, A, P)$ bildet eine Halbordnung vermöge $(X_1, Y_1) \leq (X_2, Y_2) \Longleftrightarrow X_1 \subseteq X_2$ ($\Longleftrightarrow Y_2 \subseteq Y_1$). Es gilt sogar das *Theorem: Die Begriffe zu einem Kontext bilden einen vollständigen Verband* [32]. Dieser wird *Begriffsverband* genannt und kann effektiv konstruiert werden.

Ein Begriff ist in unserer Anwendung charakterisiert ist durch eine Menge von Programmstückchen (Objekte) und eine Menge von definierten Präprozessorvariablen (Attribute), so daß jedes Objekt alle Attribute hat und jedes Attribut auf alle Gegenstände des Begriffs paßt. Intuitiv kann man sich einen Begriff als maximales markiertes Rechteck in der Tabelle vorstellen, wobei es allerdings auf Zeilen- und Spaltenvertauschungen nicht ankommt.

Am Begriffsverband kann man die Begriffstaxonomie direkt ablesen. Abbildung 7 zeigt das Resultat der Analyse für unser fiktives Beispiel. Die einzelnen Punkte entsprechen den errechneten Begriffen; die Linien entsprechen der Beziehung Oberbegriff-Unterbegriff. Ein oben an einen Begriff geschriebenes Attribut besagt, daß alle Objekte, die zu darunterliegenden Begriffen gehören, dieses Attribut haben. Ein unten an den Begriff geschriebenes Objekt besagt, daß alle Attribute, die zu darüberliegenden Begriffen gehören, auf das Objekt passen. Wenn man im Verband abwärts geht, erhält man also stärkere Aussagen über kleinere Objektmengen. Im Beispiel ist der mit CRAY markierte Begriff

[6] Die Tabelle entspricht nicht `xload`, sondern ist eine isomorphe Kopie eines in [33] präsentierten Beispiels, das wegen der Prägnanz der entstehenden Begriffshierarchie gewählt wurde.

[7] Der hier verwendete Attributbegriff kommt aus der Theorie der Begriffsanalyse und ist nicht identisch zu den Attributen aus Kapitel 5.

	SYSV	SYSV386	macII	i386	ultrix	sun	AIX	CRAY	apollo	sony	sequent	alliant
1 - 10		X		X	X	X	X			X	X	
11 - 20	X		X		X	X	X	X	X	X		
21 - 28	X		X					X	X			
29 - 40	X		X			X		X	X	X		
41 - 100	X				X	X						
101 - 106		X		X	X	X	X			X		
107 - 115	X		X			X						
116 - 125			X			X				X		
126 - 200	X		X		X	X			X			
201 - 207	X		X		X	X		X	X			

Abbildung 6: Klassifikation der Quelltextzeilen nach definierten Präprozessorvariablen

tatsächlich ({11-20, 21-28, 29-40, 201-207},{CRAY, apollo, macII, SYSV}), und dieser Begriff besagt, daß die Zeilen 11-20, 21-28, 29-40, und 201-207 die Konfigurationen CRAY, apollo, macII und SYSV eindeutig charakterisieren (und umgekehrt); insbesondere enthält die CRAY-Konfiguration gerade die Zeilen 11-20, 21-28, 29-40, 201-207. Der Begriff ist ein Unterbegriff des mit apollo markierten, der für ({11-20, 21-28, 29-40, 126-200, 201-207},{apollo, macII, SYSV}) steht – die Apollo-Konfiguration enthält zusätzlich die Zeilen 126–200. Man sieht, daß der Begriffsverband eine hierarchische Gruppierung von Objektmengen liefert.

Wichtiger ist aber die Tatsache, daß der Verband alle *Implikationen* zwischen Attributmengen liefert. Für zwei Attributmengen A und B sagen wir "A impliziert B", wenn $\tau(A) \subseteq \tau(B)$. Dies wird gelesen als "jedes Objekt, daß alle Attribute aus A hat, hat auch alle Attribute aus B". Wenn nun A und B zwei Begriffe $C = \big(\tau(A), A\big)$ und $D = \big(\tau(B), B\big)$ konstituieren, und wenn $C \leq D$, dann impliziert A offenbar B. Da Implikationen auch als Abhängigkeiten interpretiert werden können, liefert der Begriffsverband also eine feinkörnige Darstellung aller *Abhängigkeiten zwischen Konfigurationspfaden*.

In unserem Beispiel kann man am Diagramm erkennen, daß es drei Hauptkonfigurationsparameter gibt (macII, SYSV, sun), die durch andere ergänzt und verfeinert werden. Man kann sehen, daß die Zeilen 21–28 für CRAY, apollo, maxII und SYSV spezifisch sind. Man kann sehen, daß die Zeilen 11–20 in allen Konfigurationen bis auf alliant, sequent und i386 vorkommen, und man kann erkennen, daß alle Zeilen, die sony- oder ultrix-spezifisch sind, auch in der sun-Konfiguration vorkommen usw. Die apollo- und ultrix-Konfigurationen haben die Zeilen 126–200, 201–207, 11–20 gemeinsam (Infimum im Verband). Die Zeilen 126–200 und 101–106 werden beide von ultrix (und sun) "regiert" (Supremum im Verband). Derartige Information läßt sich aus einem Programm wie `xload` nicht leicht von Hand exzerpieren!

Der Begriffsverband liefert aber nicht nur alle feinkörnigen Abhängigkeiten zwischen "configuration threads", er erlaubt es auch, Verstöße gegen Prinzipien des Software Engineering unmittelbar zu er-

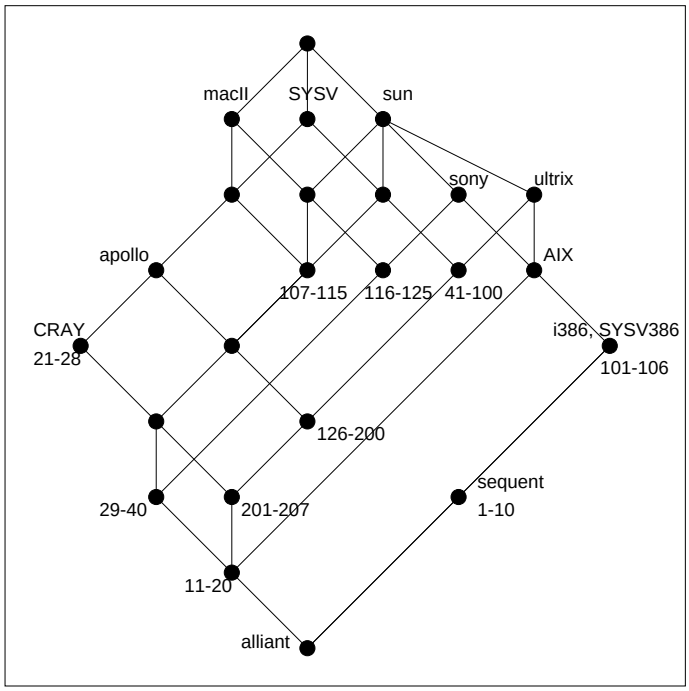
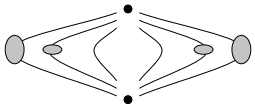


Abbildung 7: Berechneter Begriffsverband

kennen. So besagt das Prinzip der *schwachen Kopplung* ganz allgemein, daß es keine Anhängigkeiten zwischen Dingen geben sollte, die nichts miteinander zu tun haben. In unserem Fall bedeutet das: es darf keine Abhängigkeiten zwischen "semantisch verschiedenen" Präprozessorsymbolen geben. Hat man zum Beispiel verschiedene Varianten für Betriebssysteme und andere Varianten für Benutzerschnittstellen, so sollten diese nicht interferieren. Dies kann man im Begriffsverband daran erkennen, daß der Begriffsverband sich als horizontale Summe darstellen läßt:  . Die Unterverbände dürfen oben und unten "verklebt" sein, aber Querverbindungen zeigen stets Verstöße gegen das Prinzip der schwachen Kopplung auf. Auf diese Weise kann man sogar Programmierfehler entdecken, wie das abschließende Beispiel zeigen wird.

Komplementär besagt das Prinzip der *hohen Kohäsion*, angewandt auf Konfigurationsmanagement, daß "semantisch benachbarte" Präprozessorvariablen zusammen behandelt werden sollten. Hat man etwa verschiedene Betriebssystemvarianten und –untervarianten, so sollten diese in systematischer Weise zusammen auftreten. Für den Begriffsverband bedeutet das, daß entsprechende Unterverbände ein Gitter bilden sollten:  . Fehlende Kanten deuten auf nicht behandelte Kombinationen zwischen Symbolen (die etwa die verschiedenen Betriebssystemeigenschaftern charakterisieren), was stets verdächtig ist.

Zum Abschluß soll unser Analysewerkzeug für ein reales Beispiel demonstriert werden (Abbildung 8). Es handelt sich um den Stream-Editor des RCS-Systems [31]. Dieses 1656 Zeilen lange Programm verwendet 21 Präprozessorsymbole zum Konfigurationsmanagement. Die Abbildung zeigt den Begriffsverband nebst Erläuterung der Begriffe sowie einen Ausschnitt des Quelltextes. Die Begriffe unterhalb C6 beschäftigen sich mit verschiedenen Dateizugriffsverfahren; der Begriff C8 ist

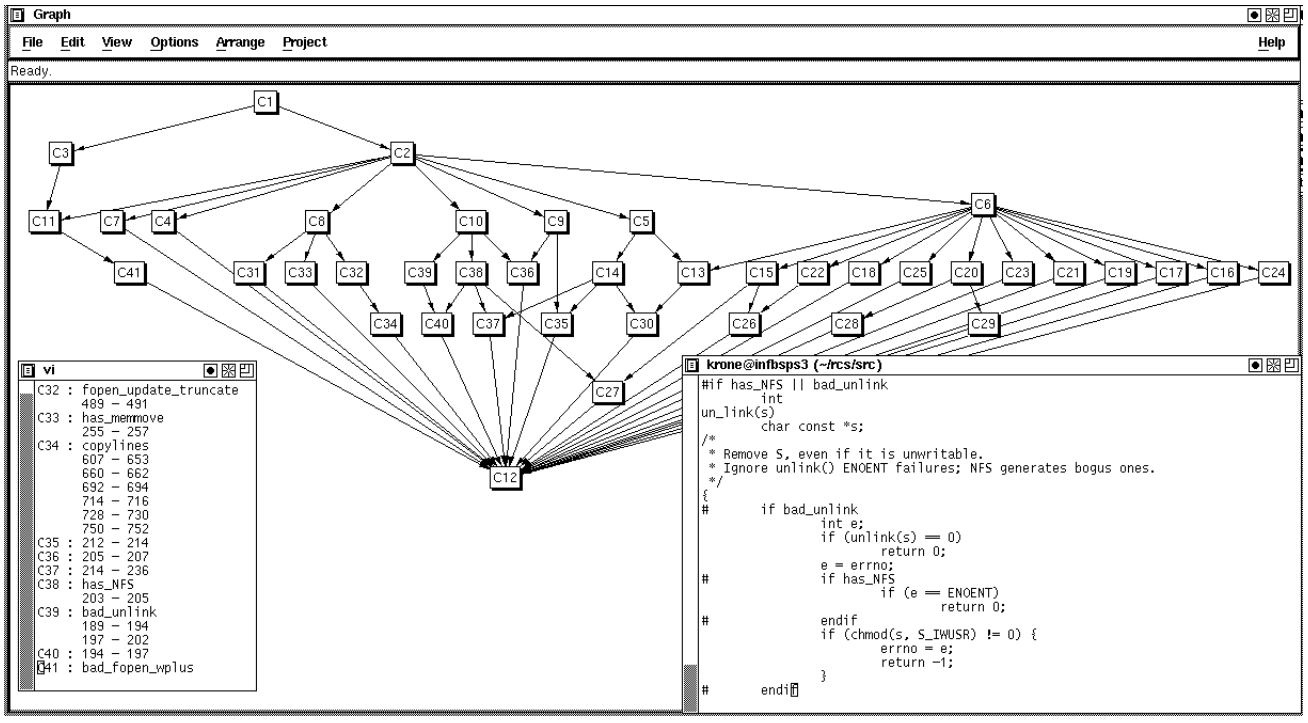


Abbildung 8: Konfigurationsstruktur des RCS Stream-Editors `rcsedit`

mit “large_memory” markiert. Die Teilverbände unter beiden Begriffen haben eine flache Struktur. C9/C10 befassen sich mit Netzwerkvarianten; der darunterliegende Teilverband hat gitterähnliche Struktur. Mithin ist hohe Kohäsion und schwache Kopplung im Prinzip gegeben. Es gibt allerdings eine Interferenz, die in C27 manifest wird: C27 ist mit 1425 – 1427 markiert, und diese beiden Zeilen werden sowohl durch has_NFS (C38) als auch durch has_rename (C15) “regiert”. Eine ähnliche Interferenz zeigt sich in C35/C37.

Obwohl die Konfigurationsstruktur also insgesamt gut ist, wird nicht klar zwischen Netzwerkvarianten und Dateizugriffsvarianten unterschieden. Dies ist verdächtig und veranlaßt uns, einen Blick in den Quelltext zu tun. Und nun zeigt sich, daß wir einen Bug wieder-entdeckt haben, der sogar als Kommentar im Programm vermerkt ist: eine spezifische Konfiguration von NFS und dem Dateisystem kann aufgrund eines NFS-Fehlers zur Zerstörung der RCS-Datei führen! Der interessierte Leser mag sich den Quelltext selbst ansehen, wir zitieren hier nur den abschließenden Satz des genannten Kommentars: “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”.

Das Beispiel zeigt, daß Interferenzen auch von Nicht-Programmautoren erkannt werden können und tatsächlich auf reale Probleme hinweisen. [16] präsentiert weitere Beispiele, die darlegen, daß nicht nur Abhängigkeiten und Interferenzen zwischen Konfigurationspfaden abgelesen werden können, sondern auch die Gesamtqualität der Konfigurationsstruktur beurteilt werden kann.

Es ist ohne weiteres möglich, unser Verfahren auf modernere Techniken des Konfigurationsmanagements anzuwenden, z.B. *shape* [18]. Hierzu ist lediglich ein neues Frontend erforderlich; der Begriffsanalyse-Kern und die graphische Ausgabe nebst Grapheditor und Layouter bleiben unverändert.

7 Die NORA-Architektur

Obwohl der Schwerpunkt unserer Arbeit keineswegs auf der Architektur von Softwareentwicklungsumgebungen liegt, soll doch die Architektur von NORA kurz beschrieben werden. Strebte man früher integrierte Softwareentwicklungsumgebungen an, die sich leider nur allzuoft als monolithische Monster entpuppten, so setzt man heute auf eine Sammlung dedizierter, kooperierender Werkzeuge. NORAs Konzept der autonomen, kommunizierenden Agenten trägt dem Rechnung.

7.1 Agenten, Blackboards und Ereignisse

NORA ist in Form einer *Blackboard-Architektur* [22] realisiert. Ein Blackboard ist eine Wissensbasis, die einzelnen Werkzeugen, den Wissensquellen oder *Agenten*, zur Verfügung steht. Im Blackboard legen die Agenten ihre Sicht der Welt ab. Alle Interaktion zwischen Agenten erfolgt ausschließlich über das Blackboard. Dieser Ansatz ähnelt bekannten Architekturen von Softwareentwicklungsumgebungen wie z.B. SoftBench [4] oder Software-Bus [7]. In NORA besitzt das Blackboard jedoch eigene Intelligenz und stellt einen einheitlichen Wissensrepräsentations- und Kommunikationsmechanismus in Form der Feature-Terme zur Verfügung.

Stellt ein Agent eine Änderung der Welt fest, teilt er dies dem Blackboard als *Ereignis* (Event) mit. Ereignisse sind zunächst einmal *Fakten*, die als *Feature-Terme* (s.o.) kodiert werden. So wird etwa ein Agent die inferierte Abhängigkeit zwischen zwei Komponenten `foo` und `bar` als `uses[client: foo, server: bar]` an das Blackboard mitteilen.

Ereignisse sind aber nicht nur auf Fakten beschränkt, sondern können sich aber auch auf andere Ereignisse beziehen. So werden als Ereignisse aufgefaßt:

- Das *Interesse* (Query) eines Agenten an einem bestimmten Ereignis *x*, gekennzeichnet durch "*x*?". Hat ein Agent sein Interesse dem Blackboard mitgeteilt, wird er in Zukunft vom Blackboard mit entsprechenden Ereignissen versorgt. Soll etwa der Graph-Editor Abhängigkeiten darstellen, kann er sein Interesse als `uses[client: C, server: S]?` äußern, um dann vom Blackboard mit passenden Fakten versorgt zu werden.
- Der *Wunsch* (Request) eines Agenten, ein bestimmtes Ereignis *x* möge eintreten, gekennzeichnet durch "*x*!". So könnte ein Compiler-Agent auf den Wunsch `is-compiled-from[target: foo, source: foo.mod]!` mit der Übersetzung von `foo.mod` reagieren. Mit `is-compiled-from[target: foo, source: foo.mod]` meldet der Compiler-Agent schließlich den erfolgreichen Abschluß.
- Die *Rücknahme* (Retract) eines zuvor gültigen Ereignisses *x*, gekennzeichnet durch "*x*@". Sollte irgendwann die Datei `foo.mod` geändert werden, ist der Fakt `is-compiled-from[target: foo, source: foo.mod]` nicht mehr gültig. Dies wird dem Blackboard mit dem Ereignis `is-compiled-from[target: foo, source: foo.mod]@` mitgeteilt.

Interesse, Wunsch und Rücknahme können zu beliebig komplexen *Meta-Ereignissen* zusammengesetzt werden, die vor allem dazu dienen, Informations-Anbieter und Informations-Kunden zusammenzubringen. Der Compiler-Agent etwa wird seine Fähigkeiten mit `is-compiled-from[target: T, source: S]!?` anpreisen, damit er auch passende Wünsche erhält. Dieses Ereignis selbst kann aber von anderen Agenten verarbeitet werden – etwa im Graph-Editor, der daraufhin einen Menüpunkt "Übersetzen" verfügbar macht. Hierfür muß der Graph-Editor sein Interesse als `is-compiled-from[target: T, source: S]!??` bekunden. Neben solchen Rendezvous-Mechanismen können aber auch komplexe Verhandlungen modelliert werden. Ein Agent kann etwa auf einen Wunsch `do-it!` von anderen Agenten Unterstützung `do-it!!` oder Ablehnung `do-it!@!` einholen, bevor er den Wunsch mit `do-it` erfüllt.

Insgesamt steht es jedem Agenten frei, wann und wie er auf Ereignisse reagiert. Insbesondere besteht keine Garantie, daß eine Query beantwortet wird, und es besteht auch keine Möglichkeit, herauszufinden, daß zu einem gegebenen Zeitpunkt alle Antworten vorliegen. Diese *asynchrone Kommunikation* fördert die Entwicklung von dynamischen Agenten, die auf jede Änderung der Welt unmittelbar reagieren. So kann etwa der Graph-Editor automatisch Knoten und Kanten hinzufügen oder löschen, wenn sich entsprechende Änderungen ergeben haben.

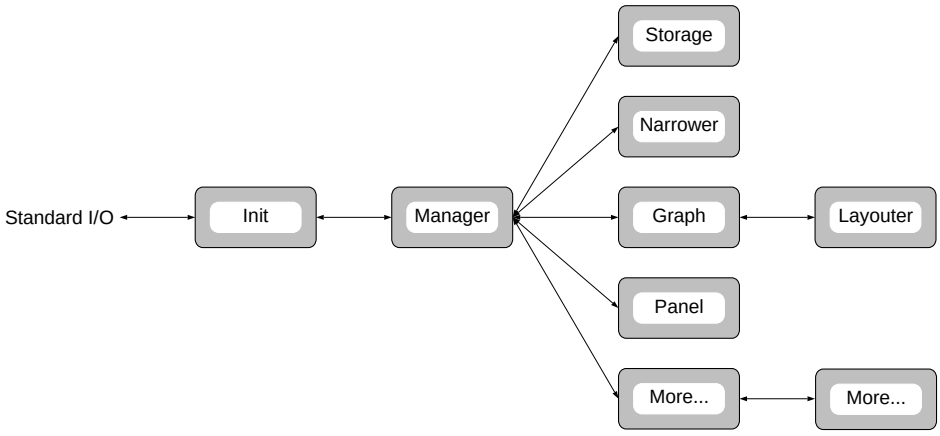


Abbildung 9: Ein Netzwerk aus NORA-Agenten

7.2 Eine verteilte Wissensbasis

In NORA sind Agenten als separate Prozesse realisiert. Dies erlaubt auf einfachem Weg parallele und verteilte Ausführung von Agenten; auch das Testen und Warten von Agenten wird erheblich erleichtert.

Einfache Agenten sprechen das Blackboard über ihre Standard-Ein/Ausgabe an. Einfache Agenten können als kurze Shell-Skripte realisiert werden, die nur eine kleine Anzahl von Ereignissen verarbeiten. So kann der Benutzer einen Texteditor-Agenten schreiben, der auf das Ereignis `edit[file: foo.mod]!` seinen Lieblings-Texteditor aktiviert. Noch einfachere Agenten beginnen gleich bei ihrem Start mit der Arbeit und geben ihre Ergebnisse als Fakten heraus, unabhängig vom Interesse der anderen Agenten.

Komplexe Agenten unterhalten ein eigenes Blackboard, können dort selbst ankommende Ereignisse nachvollziehen und automatisch an angeschlossene Agenten weitergeben. Auf diese Art und Weise ist das Blackboard auf zahlreiche vernetzte Agenten verteilt (Abbildung 9). Da jeder Agent genau den Inhalt des Blackboards hält, den er oder angeschlossene Agenten benötigen, wird die Kommunikation auf ein Minimum reduziert.

8 Schluß

NORA zielt auf die Nutzbarmachung von Inferenzverfahren in Softwarewerkzeugen. Dies führt in vielen Fällen zu leistungsstärkeren Werkzeugen. Aus Benutzersicht besteht der größte Vorteil von inferenzbasierten Werkzeugen darin, daß viele Dinge nicht explizit spezifiziert werden müssen, sondern vom System erschlossen werden können; dies ermöglicht auch eine frühere Fehlererkennung bei der Softwareentwicklung. Die Architekur bietet leichte Integration in die bestehende UNIX-Welt und leichte Erweiterbarkeit.

Die Implementierung von NORA ist noch nicht vollständig. Graph-Editor, Agenten, Blackboards, syntaktische und semantische Analyse sind fertiggestellt. Gleichfalls fertiggestellt ist das Werkzeug zur Begriffsanalyse von Konfigurationsstrukturen. Beide stehen über *anonymous ftp* allen Interessenten zur Verfügung (`ftp.ips.cs.tu-bs.de:/pub/local/softech/nora` und `ftp.ips.cs.tu-bs.de:/pub/local/softech/xplain`). Unvollständig sind noch das Konfigurationssystem und das Komponentenretrieval; ferner sind die Teile von NORA noch nicht vollständig integriert. Parallel zur Implementierung untersuchen wir, wie inferenzbasierte Verfahren auch für andere Bereiche des Software Engineering nutzbar gemacht werden können.

Danksagung. NORA wird von der Deutschen Forschungsgemeinschaft unter den Kennzeichen Sn11/1–2 und Sn11/2–1 gefördert. Lars Düning, Erich Gode, Maren Krone, Christian Lindig und Thorsten Sommer unterstützen die Implementierung von NORA.

9 Literatur

1. Aït-Kaci, H.: An algebraic semantics approach to the effective resolution of type equations. Theoretical Computer Science 45, 293–351 (1986)
2. Bahlke, R., Snelting, G.: The PSG system: From formal language definitions to interactive programming environments. ACM Transactions on Programming Languages and Systems 8 (4), 547-576 (1986)
3. Bibel, W.: DFG-Schwerpunktprogramm Deduktion. Künstliche Intelligenz 6 (3), 71 – 74 (1992)

4. Cagan, M.: The HP SoftBench environment: An architecture for a new generation of software tools. Hewlett-Packard Journal 41 (3), 36–47 (1990)
5. Damas, L., Milner, R.: Principal type schemes for functional programs. Proc. 9th Principles of Programming Languages. ACM, 207 – 212 (1982)
6. Doberkat, E.: Zur Wiederaufbereitung von Software. Informatik – Forschung und Entwicklung 4, 14 – 24 (1989)
7. The ESF Software Bus subproject: The ESF Software Bus – An Overview. ESF Technical Report. ESF, Hohenzollerndamm 152, Berlin (1991).
8. Fages, F.: Associative-commutative unification. Proc. 7th International Conference on Automated Deduction, Lecture Notes in Computer Science 170, pp. 194 – 208 Berlin, Heidelberg, New York: Springer 1984
9. Feiler, P. (ed.): Proc. of the 3rd international workshop on software configuration management. ACM (1991)
10. Feldmann, S. I.: Make — a program for maintaining computer programs. Software Practice and Experience, Vol. 9, 255 – 265 (1979)
11. Fischer, B.: Resolution for feature logic. Workshop der GI-FG 2.1.4, Alternative Konzepte für Sprachen und Rechner, Bad Honnef (1993)
12. Ganter, B., Wille, R., Wolff, K.E. (Hsg.): Beiträge zur Begriffsanalyse. BI Wissenschaftsverlag (1987)
13. Grosch, F.J., Snelting, G.: Polymorphic components for monomorphic languages. Proc. 2nd International Workshop on Software Reuse. IEEE, 47 – 55 (1993)
14. Jones, C.: Systematic software development using VDM. Englewood Cliffs: Prentice Hall 1990
15. Kievernagel, M., Snelting, G.: Software-Komponentenretrieval mit Deduktionsverfahren. Kolloquium des DFG-Schwerpunktprogramms “Deduktion”. TH Darmstadt (1994)
16. Krone, M., Snelting, G.: On the inference of configuration structures from source code. Proc. 16th International Conference on Software Engineering, IEEE 1994 (erscheint demnächst).
17. Lins, C.: The Modula-2 software component library. Vol. 1 – 4. Berlin, Heidelberg, New York: Springer 1989
18. Mahler, A., Lampen, A.: An integrated toolset for engineering software configurations. Proc. ACM Symposium on Practical Software Development Environments, SIGSOFT Notices 13 (5), 191 –200 (1988)
19. McCune, W.: OTTER 2.0 users guide. Argonne National Laboratory (1990)
20. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17 (3), 348 – 375 (1978)
21. Nicklin, P.: Managing multi-variant software configurations. Proc. 3rd International Workshop on Software Configuration Management, Trondheim, 53–57 (1991)
22. Nii, H.P.: Blackboard systems (part 1 & 2). AI Magazine, 7 (2), 38 – 53 und 7 (3), 82 – 106 (1986)
23. Prieto-Diaz, R.: Classifying software for reusability. IEEE Software 4, 1 (1987)
24. Rich, A., Solomon, M.: A logic-based approach to system modelling. Proc. 3rd International Workshop on Software Configuration Management, Trondheim, 84–93 (1991)
25. Rittri, M.: Retrieving library identifiers via equational matching of types. Proc. 10th Conference on Automated Deduction, LNCS 449, 603 – 617 (1990)
26. Rollins, E., Wing, J.: Specifications as search keys for software libraries. Proc. International

Conference on Logic Programming, Paris (1991)
27. Smolka, G.: Feature-constrained logics for unification grammars. Journal of Logic Programming 12, 51–87 (1992)
28. Shao, Z., Appel, A.: Smartest recompilation. Proc. 20th Symposium on Principles of Programming Languages, ACM, 439 – 450 (1993)
29. Snelting, G.: The calculus of context relations. Acta Informatica Vol. 28, S. 411–445 (1991)
30. Snelting, G., Grosch, F.-J., Schroeder, U.: Inference-based support for programming in the large. Proc. 3rd European Software Engineering Conference, Milano 1991. Lecture Notes in Computer Science 550, 396–408 (1991)
31. Tichy, W. F.: RCS — A system for version control. Software Practice and Experience 15 (7), 637 – 654 (1985)
32. Wille, R.: Restructuring lattice theory: An approach based on hierarchies of concepts. In: I. Rival (ed.) Ordered Sets, pp. 445–470. Dordrecht-Boston: Reidel 1982
33. Wille, R.: Concept lattices and conceptual knowledge systems. Computers & Mathematics with Applications 23, 493–515 (1992)
34. Zeller, A.: Configuration management with feature logics. Informatik-Bericht 94–01. TU Braunschweig (1994).