Inferenzbasierte Werkzeuge in NORA

Gregor Snelting, Andreas Zeller
Arbeitsgruppe Softwaretechnologie
Technische Universität Braunschweig
Gaußstraße 17, D-38106 Braunschweig

Zusammenfassung

Die experimentelle Softwareentwicklungsumgebung NORA strebt die Nutzbarmachung neuer Ergebnisse im Bereich Unifikationstheorie und Deduktionsverfahren für Softwarewerkzeuge an. Gruppiert um eine Bibliothek wiederverwendbarer Softwarekomponenten bietet NORA interaktive Werkzeuge zum Komponentenretrieval mit Spezifikationen und Verwendungsmustern, 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 gibt eine Übersicht über die Werkzeuge und die verwendeten Inferenzverfahren; abschließend wird die Systemarchitektur und die Kommunikation zwischen den Werkzeugen skizziert.

Stichworte: Inferenzverfahren, Wiederverwendung, Konfigurationsmanagement, Reverse Engineering

1 Einleitung

Im Bereich der Softwarewerkzeuge gibt es ein noch unausgeschöpftes Potential, nämlich die Verwendung von Inferenzverfahren. Es gibt eine Fülle neuerer Ergebnisse im Bereich Deduktionssysteme und Unifikationstheorie, deren Anwendbarkeit für das Software Engineering jedoch noch keineswegs ausgeschöpft ist – im Gegenteil, Softwareingenieure und Inferenzforscher pflegen sich meistenteils mit Mißvergnügen zu betrachten.

Um diese Lücke zu schließen, entwickeln wir die experimentelle Softwareentwicklungsumgebung NORA. NORA zielt auf die Nutzbarmachung von Inferenzverfahren für Softwarewerkzeuge und behandelt die folgenden Aspekte der Softwareentwicklung:

- Schnittstelleninferenz in einer Bibliothek wiederverwendbarer Softwarekomponenten
- Komponentenretrieval, das Spezifikationen und Verwendungsmuster als Suchschlüssel erlaubt
- Interaktives Konfigurationsmanagement, basierend auf einer Inferenzmaschine für Feature-Logik
- Unterstützung für das Reverse Engineering durch Inferenz von Varianten- und Konfigurationsstrukturen aus existierender Software.

Im Gegensatz zu existierenden Ansätzen sollen interaktive Werkzeuge entwickelt werden, die unvollständige Information handhaben können (*Ambiguitätstoleranz*), inkonsistente Information tolerieren (*Fehlertoleranz*), schnelle und bequeme Interaktion ermöglichen (*Inkrementalität*), und mit sprachspezifischem Wissen parametrisiert sind (*Generierbarkeit*). Diese Eigenschaften ermöglichen frühere Fehlererkennung und unterstützen evolutionäre Systementwicklung.

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; die theoretischen Grundlagen und weitere Details werden wir an anderer Stelle beschreiben.

2 Softwarekomponenten in NORA

NORAs Werkzeuge gruppieren sich um eine Bibliothek wiederverwendbarer Softwarekomponenten. 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 Signaturschema (s.u.) oder zusätzliche Spezifikationen etwa in Form von Vor-/Nachbedingungen werden separat gespeichert. Die Verwendung einfacher Dateien erlaubt einfache Integration in die bestehende UNIX-Landschaft; weitergehende Ansätze wie Datenbanken für Softwarewerkzeuge usw. sind nicht unser Thema.

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

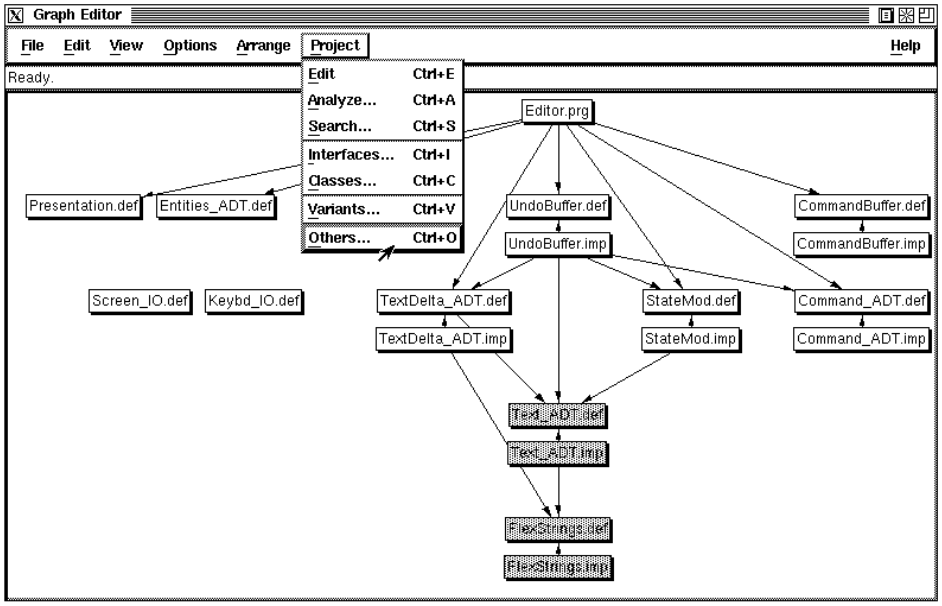

Abbildung 1: NORA Graph Editor – Modulabhängigkeitsgraph

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 (s.u.) aktiviert werden, das die verfügbaren Varianten anzeigt und die Konfiguration eines Systems erlaubt. Schließlich ist es möglich, die Konfigurationsstruktur existierender Quelltexte zu inferieren.

3 Schnittstelleninferenz und polymorphe Komponenten

Die meisten prozeduralen Sprachen erlauben aufgrund ihrer eingeschränkten Typsyteme nicht die Verwendung *polymorpher* Softwarekomponenten im Stil funktionaler Programmierung. Der in modernen funktionalen Sprachen angebotene parametrische Polymorphismus [11] erhöht aber das Wiederverwendungspotential beträchtlich, ohne die Sicherheit strenger Typisierung aufzugeben. NORA erlaubt nun die Verwendung polymorpher Komponenten für “gewöhnliche” monomorphe Sprachen wie Modula-2. Die Quelle des Polymorphismus ist die Verwendung freier (undeklarierter) Bezeichner, wie z.B. in der folgenden Sortierprozedur, die zwei nichtdeklarierte Bezeichner enthält:

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

Ein Compiler kann diese isolierte Prozedur nicht übersetzen; in NORA kann sie aber eine eigenständige Komponente sein. In der Tat ist die Prozedur insofern polymorph, daß der Code für *jeden* Elementtyp korrekt ist.

NORA analysiert Bibliothekskomponenten nach dem Verfahren der Kontextrelationen [2,22]. Für jede Komponente wird ein *Signaturschema* inferiert, das analog zu Typschemata in funktionalen Sprachen ist: es beschreibt exakt die Komponentenschnittstelle bzw. die möglichen Verwendungen. Die Beispielkomponente etwa hat das Signaturschema

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

Komponenten wie die obige können durch eine einfache “#include ... “ Anweisung (die allerdings von NORA, nicht vom Präprozessor bearbeitet wird) in anderen Komponenten verwendet werden. Nichtdeklarierte Bezeichner, die den Polymorphismus einer Komponente induzieren, können dabei von außen instantiiert werden, um die Variablenbindungen genau zu kontrollieren.

Nachdem der Abhängigkeitsgraph berechnet ist, werden die Signaturschemata benutzt, um korrekte Verwendung von Komponenten sicherzustellen. Mehrfache Verwendungen einer Komponente erfordern dehalb keine mehrfachen Neuanalysen [8]. Falls die inferierte Signaturschnittstelle keine freien Typvariablen enthält, ist es sogar möglich, die Komponente in Abwesenheit der importierten Definitionsmodule zu übersetzen ("Smartest Recompilation", vgl. [21]).

4 Komponentenretrieval mit Verwendungsmustern

NORA verwendet zwei neuartige Verfahren zum Komponentenretrieval:

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

Beide Ansätze wurden bisher nur für funktionale Sprachen angegangen [18,19], nicht aber für prozedurale Sprachen. Vorhandene Retrieval-Ansätze für prozedurale Sprachen, die über manuelle Klassifikation (wie sie etwa bei der Facettenklassifikation [16] verwendet wird) hinausgehen, erzeugen etwa aus Softwarekomponenten Kontrollfluß-skelette, die bei der Suche instantiiert werden können [4]; 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 Typcharakteristik gibt. Dazu werden die für die Verwendungsstelle inferierten Typschemata mit Typschemata von Bibliotheksobjekten unifiziert; falls dies nicht fehlschlägt, gilt das Objekt als gefunden. Da Benutzer oft die Reihenfolge von Parametern vergessen, wird beim Suchen für die Unifikation bestimmter Teilattribute (z.B. Parameterlisten) AC1-Unifikation [5] 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. 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. Das zur Zeit laufende DFG-Schwerpunktprogramm "Deduktion" zielt u.a. auf die Entwicklung effizienter Deduktionssysteme, die wir verwenden wollen [3].

5 Interaktives, inferenzbasiertes Variantenmanagement

Die meisten der "klassischen" Verfahren zum Variantenmanagement sind Erweiterungen von *MAKE* [6] oder *RCS* [24]. Das *shape*-System [10] 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 [17] oder Zusammenstellen [13] einer Konfiguration, unterstützen aber keine inkrementelle oder interaktive Vorgehensweise und erfordern vollständige Spezifikationen. NORA hingegen

- läßt den Benutzer interaktiv und inkrementell aus der Menge der möglichen Konfigurationen seine Auswahl treffen;
- weist den Benutzer frühzeitig auf Konfigurations-Konflikte hin;
- ermöglicht die Spezifikation von Variantenmerkmalen durch Feature-Terme, einschließlich Variablen und logischen Verknüpfungen;
- verschont den Benutzer von Spezifikationen, die vom System selbst inferiert werden können – so etwa der Abhängigkeitsgraph oder bestimmte Konfigurationsmerkmale.

In NORA kann jede Komponente in mehreren *Varianten* vorliegen. Varianten werden durch ihre *Features* (Merkmale) unterschieden. Ein Feature ist ein Paar der Form *attribut: wert*. So können etwa zwei Varianten für verschiedene Betriebssysteme durch die Features `operating-system: unix` und `operating-system: dos` unterschieden werden.

Features können zu *Feature-Termen* [1,20] kombiniert werden. Wir unterscheiden:

1. Das gleichzeitige Auftreten von Features *(Konjunktion)*. Konjunktion wird durch eckige Klammern dargestellt, etwa
   ```
   [operating-system: dos,
    concurrent: false] .
   ```
   Hierbei darf jedes Attribut nur einmal auftreten.
2. Das alternative Auftreten von Features *(Disjunktion)*. Alternativen werden in geschweiften Klammern aufgezählt, etwa
   ```
   [operating-system: unix,
    {[concurrent: true],
     [concurrent: false]}]
   ```
   Vereinfachungen wie
   ```
   [operating-system: unix,
    concurrent: {true, false}]
   ```
   sind zulässig.
3. Das Nicht-Auftreten von Features *(Negation)*, gekennzeichnet durch eine vorangestellte Tilde (~).[1]

[1] Hierbei muß zwischen dem Nicht-Auftreten eines Attributs

4. *Variablen*, dargestellt durch Großbuchstaben. Variablen werden verwendet, um zu spezifizieren, daß bestimmte Subterme gleich sein müssen.

Jeder Variante v einer Komponente ist genau ein Feature-Term *conf*(v) zugeordnet, der die möglichen Konfigurationen beschreibt. Diese werden vom Benutzer anhand der Varianten-Attribute spezifiziert oder aus bestehenden Beschreibungen inferiert.

Die möglichen Konfigurationen *conf*(k) einer Komponente k mit einer Varianten-Menge $V = \{k', k'', k''', \ldots\}$ erhält man durch *Generalisierung* über V:

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

Die Generalisierung *conf*(k) besteht aus einer Aufzählung der Einzelterme und beschreibt damit alle möglichen Konfigurationen von k. Jede neue Variante erweitert die Menge der möglichen Konfigurationen.

Die möglichen Konfigurationen *conf*(K) einer Menge von Komponenten $K = \{k_1, k_2, \ldots\}$ erhält man durch *Feature-Unifikation* über K:

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

Bei der Unifikation von zwei Termen wird versucht, Einträge mit identischen Attributnamen gleich zu machen, indem Variablen instantiiert bzw. atomare Werte verglichen werden: $[a : X, b : p, c : q] \sqcap [a : [f : t, g : u], b : Y, d : Y] = [a : [f : t, g : u], b : p, c : q, d : p]$. Zusammengesetzte Terme werden rekursiv unifiziert, bei Disjunktionen müssen alle Kombinationsmöglichkeiten probiert werden, und die Negation wird nach dem bekannten Prinzip "Negation by Failure" behandelt. Falls dasselbe Attribut in den beiden Termen nicht denselben atomaren Wert hat, schlägt die Unifikation fehl (Inkonsistenz): $[os : ibm] \sqcap [os : sun] = \bot$. Sonst liefert die Unifikation einen Feature-Term (sog. Unifikat), der den *gemeinsamen Durchschnitt* der durch die Ausgangsterme repräsentierten Information verkörpert. Das Unifikat beschreibt so die möglichen Konfigurationen der Komponentenmenge. Jede hinzukommende Komponente schränkt die Menge der möglichen Konfigurationen ein. Ist *conf*$(K) = \bot$, existiert keine gültige Konfiguration.

Gewöhnlich wird man K als Gesamtmenge der Komponenten eines Systems wählen. Man kann aber K auch als beliebige Teilmenge wählen, um lokale Konsistenzen zu überprüfen. Da die Feature-Terme einen vollständigen Verband bilden (sog. "subsumption lattice"), erhält man gleichzeitig eine natürliche Taxonomie der Varianten- und Komponentenbeschreibungen.

Ein Beispiel (Abbildung 2): Ein (zugegebenermaßen äußerst frugaler) Compiler bestehe aus drei konfigurierbaren Komponenten k_1, k_2, k_3. Der Command Line Interpreter (k_1) stellt unterschiedliche Schnittstellen zur Verfügung, wenn die Verfahren des Constant-Folding oder Peephole-Optimizing verfügbar sind. Constant-Folding wird vom Optimizer (k_2) realisiert; es ist nur dann verfügbar, wenn Zielmaschine und übersetzende Maschine identisch sind. Der Code-Generator (k_3) erzeugt Code für verschiedene Architekturen; auf der Sun-Architektur steht Peephole-Optimizing zur Verfügung.

Aus der Unifikation $K = k_1 \sqcap k_2 \sqcap k_3$ erhalten wir den Term K, der die möglichen Konfigurationen des Gesamt-Systems beschreibt. Je nach Wahl des Benutzers können jetzt die Möglichkeiten sukzessive eingeschränkt werden. Wählt er etwa als Ziel-Architektur `ibm`, so ergibt sich als neue Menge $K \sqcap$ `[target-arch: ibm]` =

```
[target-arch: ibm,
 peephole-optimizing: off,
{[host-arch: ibm,
   constant-folding: {on, off}],
 [host-arch: ~ibm,
   constant-folding: off]}] .
```

(~attribut: wert) und dem Nicht-Auftreten eines Wertes *(attribut: ~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.

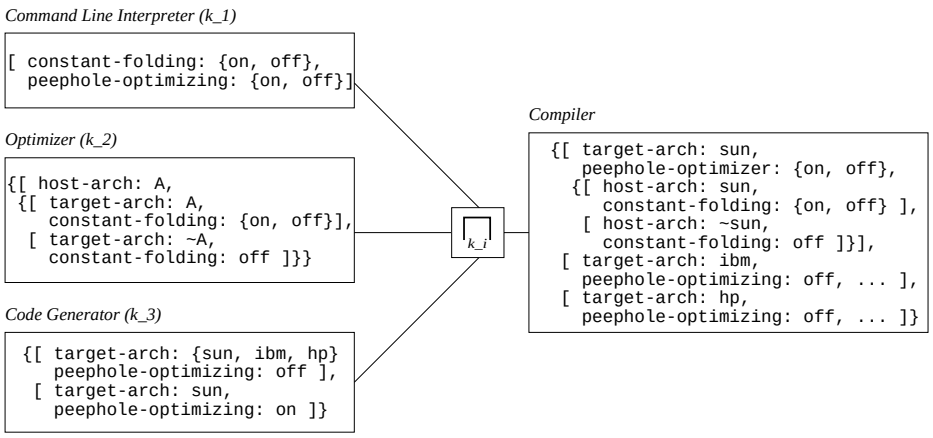

Abbildung 2: Unifikation von Variantenbeschreibungen

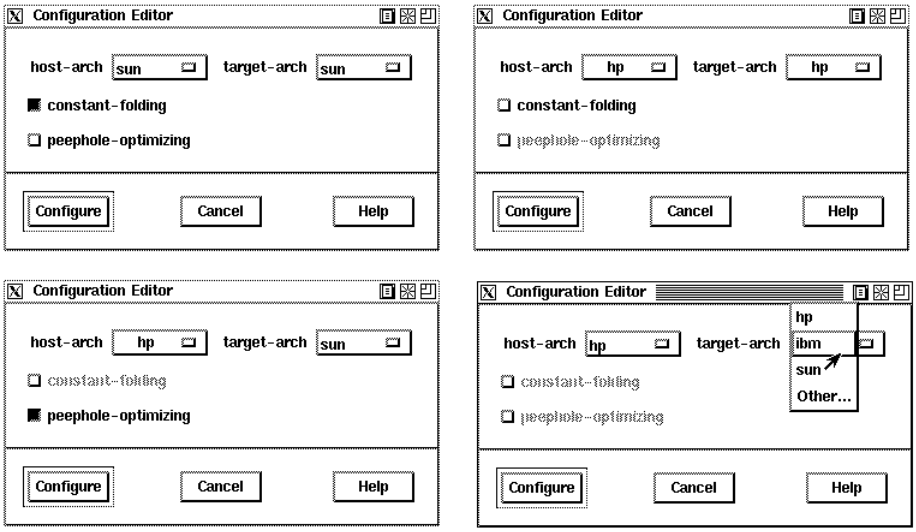

Abbildung 3: Ein Feature-Panel in vier verschiedenen Zuständen.

Die bei solchen Verfahren entstehenden Terme können trotz Minimierung 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). Je nach Auswahl von Host- und Target-Architektur werden weitere Optionen verfügbar (schwarze Schrift) oder ausgeblendet (graue Schrift). Nicht benötigte Subterme können einfach ausgeblendet werden – etwa Optionen für `sun`, wenn der Benutzer bereits `ibm` gewählt hat.

Konfigurations-Terme können gespeichert und auf Wunsch in späteren Konfigurationen wiederverwendet werden. Sei c eine solche "eingefrorene" Konfiguration. Dann kann der Benutzer statt $conf(x)$ die Konfiguration $conf(x) \sqcap c$ betrachten, wodurch nur solche Attribute zur Auswahl gestellt werden, die (noch) nicht in c angegeben wurden. Dieses Verfahren kann auch genutzt werden, um Wissen über Konfigurations-Abhängigkeiten zu kodieren – etwa Betriebssysteme mit ihren spezifischen Eigenschaften. In Zukunft wird es möglich sein, Konfigurations-Regeln in *Feature-Logik* (einem PROLOG-ähnlichen Stil) anzugeben, etwa

```
[ os-long-filenames: true,
  host-arch: X ] :-
      [ os-family: bsd,
        host-arch: X ].
```

Das hier vorgestellte Verfahren der Beschreibung von Softwarekomponenten durch Feature-Terme kann als eine abstrakte und verallgemeinerte Form der Facettenklassifikation [16] betrachtet werden. Facettenklassifikation erlaubt die Beschreibung von Komponenten anhand gewisser Attribute, wobei – anders als in unserem Ansatz – auch Ähnlichkeitsmaße zwischen Merkmalen definiert werden können; entsprechend werden bei der Komponentensuche auch "ähnliche" und nicht nur gleiche Beschreibungen gefunden. Facettenklassifikation bietet aber weder geschachtelte Terme noch Disjunktion und Negation, von Unifikation, Subsumption und Inferenz ganz zu schweigen.

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 Präprozessors: konfigurationsspezifische Programmteile werden in `#ifdef ... #endif` eingeklammert, und beim Übersetzen werden durch das Setzen von Präprozessorvariablen die richtigen Programmstücke zusammengesetzt. Abbildung 4 zeigt ein Beispiel, nämlich einen Ausschnitt des X-Window-Auslastungsmeßprogramms "x_load.c". Hier werden Präprozessorvariablen ganz extensiv genutzt, um in Abhängigkeit von der Plattform, auf der X-Windows installiert ist, bestimmte Dinge zu tun. Das Beispiel ist (wie die Abbildung zeigt) schier unverständlich; das Programm umfaßt 724 Zeilen, von denen die meisten konfigurationsspezifisch ein- oder ausgeblendet werden.

Es wäre nun überaus nützlich, wenn man existierende Software, die nach diesem Konzept erstellt wurde, wiederum mit NORA behandeln kann, um die Konfigurationsstruktur zu analysieren, zu modifizieren oder weiterzuentwickeln. 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* [7,25]. Diese Methode wurde in zehnjähriger Arbeit am Lehrstuhl für allgemeine Algebra

```
#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 4: Auslastungsmeßwerkzeug “x_load.c”

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. Ohne auf die Details dieses Vorgangs eingehen zu wollen, möchten wir doch anmerken, daß die gewonnenen Strukturen bemerkenswert mit den intuitiven Konzepten der zugrundeliegenden Miniwelt übereinstimmen. In einer Analogie gesprochen, entspricht der Begriffsverband einer Relation der Fouriertransformierten einer reellen Funktion, und die Begriffsanalyse ist das Pendant zur Spektralanalyse kontinuierlicher Signale. Aus Platzgründen können wir auf die mathematischen Grundlagen nicht eingehen; einige grundlegende Konzepte sind im folgenden erläutert.

In “x_load.c” 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 5: die Zeilen sind mit Programmabschnitten (Zeilennummern) markiert, die Spalten mit Präprozessorvariablen[2]. In der Praxis tauchen außerdem nicht nur einfache Bedingungen, sondern auch Konjunktionen, Diskunktionen und Negationen von Bedingungen auf, außerdem können #ifdefs geschachtelt werden; all dies ist unproblematisch, wird hier aber ignoriert.

Der Begriffsanalysealgorithmus errechnet aus der Tabelle den sog. *Begriffsverband*. Ein Begriff ist charakterisiert ist durch eine Menge von Objekten (Programmstückchen) und eine Menge von Attributen (definierten Präprozessorvariablen), 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 6 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. Am Diagramm kann man 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. Derartige Information läßt sich aus einem Programm wie “xload.c” nicht leicht von Hand exzerpieren!

Aus Platzgründen müssen wir auf eine detailliertere Beschreibung nebst Bewertung aus softwaretechnischer Sicht verzichten; diese wird in [9] gegeben.

[2] Die Tabelle entspricht nicht “x_load.c”, sondern ist eine isomorphe Kopie eines in [25] präsentierten Beispiels, das wegen der Prägnanz der entstehenden Begriffshierarchie gewählt wurde.

	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 5: Klassifikation der Quelltextzeilen nach definierten Präprozessorvariablen

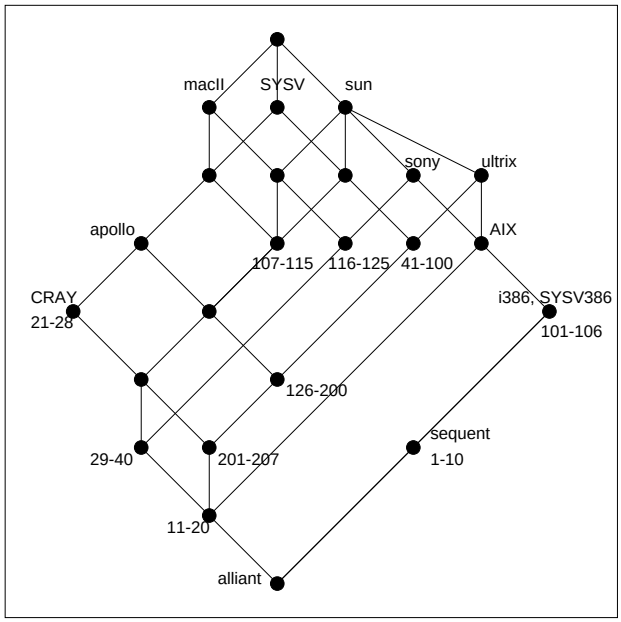

Abbildung 6: berechneter Begriffsverband

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.

NORA ist in Form einer *Blackboard-Architektur* [12,14,15] realisiert. Ein Blackboard ist eine Wissensbasis, die einzelnen Komponenten, 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.

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:

1. 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.
2. 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.
3. 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. Auf diese Art und Weise ist das Blackboard auf zahlreiche vernetzte Agenten verteilt.

Das Entwickeln neuer Agenten wird nicht zuletzt dadurch beschleunigt, daß jeder Agent die Dienste der bereits existierenden nutzen kann. Aber auch bestehende Programme können leicht in eine Agenten-Schnittstelle verpackt und so in NORA zugänglich gemacht werden.

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 keineswegs abgeschlossen. Während Agenten, Blackboards, syntaktische und semantische Analyse bereits fertiggestellt sind und die Inferenz von Konfigurationsstrukturen ihrem Abschluß entgegengeht, fehlen noch Teile des Retrieval– und des Konfigurationssystems. 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. Bernd Fischer, Franz-Josef Grosch und Matthias Kievernagel entwickelten wesentliche Konzepte und Teile des NORA-Systems. Eduard 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, S. 293–351 (1986).

[2] Bahlke, R., Snelting, G.: The PSG System: From Formal Language Definitions to Interactive Programming Environments. ACM TOPLAS 8 (4), S. 547-576 (Oktober 1986).

[3] Bibel, W.: DFG-Schwerpunktprogramm Deduktion. KI 6 (3), S. 71 – 74 (September 1992).

[4] Doberkat, E.: Zur Wiederaufbereitung von Software. Informatik – Forschung und Entwicklung 4 (1989), S. 14 – 24.

[5] Fages, F.: Associative-Commutative Unification. Proc. 7th International Conference on Automated Deduction, Lecture Notes in Computer Science 170, S. 194 – 208.

[6] Feldmann, S. I.: Make - A Program for Maintaining Computer Programs. Software Practice and Experience, Vol. 9 (April 1979).

[7] Ganter, B., Wille, R., Wolff, K.E. (Hsg.): Beiträge zur Begriffsanalyse. BI Wissenschaftsverlag 1987.

[8] Grosch, F.J., Snelting, G.: Polymorphic Components for Monomorphic Languages. Proc. 2nd International Workshop on Software Reuse, IEEE 1993, S. 47 – 55.

[9] Krone, M., Snelting, G.: On the Inference of Configuration Structures from Source Code. Informatik-Bericht 93–05, TU Braunschweig, Juli 1993. Zur Publikation eingereicht.

[10] Mahler, A., Lampen, A.: An Integrated Toolset for Engineering Software Configurations. Proc. ACM Symposium on Practical Software Development Environments, SIGSOFT Notices 13 (5), S. 191 – 200 (November 1988).

[11] Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, 17 (3), S. 348 – 375 (1978).

[12] Newell, A.: Some problems of the basic organization in problem-solving programs. Proc. Second Conference on Self-Organizing Systems, S. 393 – 423 (1962).

[13] Nicklin, P.: Managing Multi-Variant Software Configurations. Proc. 3rd International Workshop on Software Configuration Management, Trondheim, S. 53–57 (1991).

[14] Nii, H.P.: Blackboard Systems (Part 1). AI Magazine, 7 (2), S. 38 – 53 (1986).

[15] Nii, H.P.: Blackboard Systems (Part 2). AI Magazine, 7 (3), S. 82 – 106 (1986).

[16] Prieto-Diaz, R.: Classifying Software for Reusability. IEEE Software 4, 1 (Januar 1987).

[17] Rich, A., Solomon, M.: A Logic-Based Approach to System Modelling. Proc. 3rd International Workshop on Software Configuration Management, Trondheim, S. 84–93 (1991)

[18] Rittri, M.: Retrieving Library Identifiers via Equational Matching of Types. Proc. 10th Conference on Automated Deduction, LNCS 449, S. 603 – 617.

[19] Rollins, E., Wing, J.: Specifications as Search Keys for Software Libraries. Proc. International Conference on Logic Programming, Paris 1991.

[20] Smolka, G.: Feature-Constrained Logics for Unification Grammars. Journal of Logic Programming 12, S. 51–87 (1992).

[21] Shao, Z., Appel, A.: Smartest Recompilation. Proc. 20th Principles of Programming Languages, ACM 1993, S. 439 – 450.

[22] Snelting, G.: The Calculus of Context Relations. Acta Informatica Vol. 28, S. 411–445 (Mai 1991).

[23] 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, S. 396–408.

[24] Tichy, W. F.: RCS - A System for Version Control. Software Practice and Experience 15 (7), S. 637 – 654 (Juli 1985).

[25] Wille, R.: Concept Lattices and Conceptual Knowledge Systems. Computers & Mathematics with Applications 23 (1992), S. 493–515.