Rechnergestützte Erstellung von Prototypen für Programme auf relationalen Strukturen

Rudolf Berghammer, Thorsten Hoffmann, Barbara Leoniuk

Bericht Nr. 9905
Juli 1999

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Preusserstraße 1–9, D-24105 Kiel

Inhaltsverzeichnis

1 Einleitung

Seit vielen Jahren gewinnen auch im industriellen Umfeld die Korrektheit und Zuverlässigkeit von Software und die Verwendung von formalen Methoden zu ihrer Erstellung immer stärkere Bedeutung. Die Gründe hierfür sind vielfältig. Man denke etwa an den Einsatz von Software in äußerst sicherheitskritischen Bereichen. Dieser ist oft nur dann noch zu verantworten, wenn eine weitgehende Risikofreiheit gegeben werden kann. Aber auch Wirtschaftlichkeit ist mittlerweile ein Grund für Korrektheit, da die Folgekosten für die nachträgliche Beseitigung von Fehlern die Gesamtkosten eines Produkts oft unangemessen erhöhen. Korrektheit ist natürlich auch unabdingbar für häufig wiederverwendete Software. Zuletzt sei noch auf den Zusammenhang von Korrektheit, Zuverlässigkeit und Schutz vor unbefugten Manipulationen hingewiesen, denn solche nutzen im Normalfall Fehler in den Programmen aus.

Es ist allgemein anerkannt, daß Prototypen sehr wichtige Bestandteile bei der Entwicklung von Software insbesondere im Hinblick auf Korrektheit, Qualitätssicherung und Risikominderung sind; siehe [20, 40]. Das Ziel von Prototyping beim Software-Entwurf ist die schnelle und billige Realisierung eines geplanten Programms durch einen ausführbaren Prototyp. So eine Erstimplementierung ist beispielsweise ein wichtiges Hilfsmittel zur Ermittlung und zum Studium des funktionalen Verhaltens beim Erstellen der Anforderungsdefinition, etwa anhand von ausgewählten Tests. Prototypen sollen mit sehr geringem Aufwand erstellbar und leicht veränderbar sein, denn Experimentieren spielt in dieser Phase eines Software-Projekts oft eine große Rolle. Dies zeigt, daß ökonomisches Prototyping nicht ohne entsprechende Werkzeuge möglich ist.

In diesem Bericht befassen wir uns mit der formalen und rechnergestützten Erstellung von Prototypen für Programme auf diskreten Strukturen, die in einfacher Weise durch Relationen modelliert werden können. Beispiele für solche Strukturen findet man sehr häufig sowohl an den Hochschulen als auch in industriellen Anwendungen. Man denke nur an Graphen, die man oft durch Relationen auf Knoten und/oder Pfeilen beschreibt. Sie werden in den beiden genannten Bereichen häufig zur Visualisierung abstrakter Konzepte oder komplex strukturierter und vielfach verzahnter Sachverhalte verwendet, aus der sich dann Einsichten und Ideen zu Problemlösungen gewinnen lassen. Graphentheoretische Probleme sind an den Hochschulen auch sehr beliebt in Vorlesungen über Algorithmen und Datenstrukturen, da man an ihnen nicht nur sehr schön die verschiedenen Methoden zum Entwurf von effizienten Algorithmen demonstrieren kann, sondern auch die Unmöglichkeit, gewisse Probleme effizient zu lösen. In Verbindung mit Relationen sollten weiterhin Petri-Netze und Ereignis-Strukturen (event structures) genannt werden, die häufig bei der Modellierung von konkurrierenden Prozessen Verwendung finden. Die statische Struktur eines Petri-Netzes ist im Grunde ja nichts anderes als die graphentheoretische Verkleidung eines Paars von Relationen von den Stellen (Bedingungen) zu den Hürden (Ereignissen) und umgekehrt. Auch eine Ereignis-Struktur ist abstrakt ein Paar von Relationen, nämlich einer Kausalitäts- und einer Konfliktrelation auf Ereignissen.

Der Bericht ist wie folgt gegliedert. Zuerst stellen wir in Abschnitt 2 kurz die benötigten relationenalgebraischen Grundlagen zusammen. In Abschnitt 3 beschreiben wir dann das an der Universität Kiel entwickelte Computersystem RelView zum prototypischen Manipulieren von Relationen. Die folgenden drei Abschnitte 4, 5 und 6 sind der Kern dieses Berichts. Hier zeigen wir anhand von ausgewählten Beispielen, wie Relationen es erlauben, grundlegende Datenstrukturen zu modellieren, wie man aus formalen prädikatenlogischen Problembeschreibungen schnell und einfach zu relationalen Spezifikationen als Prototypen kommen kann, und wie es durch Anwendung des relationalen Kalküls in Kombination mit Programmentwicklungstechniken schließlich möglich ist, daraus effiziente und per Konstruktion korrekte relationale Programme zu gewinnen, die ohne großen Aufwand in eine gängige imperative Programmiersprache (wie C oder Modula-2) übertragbar sind. Darüberhinaus demonstrieren wir in diesen Abschnitten

auch Anwendungen von RelView. Abschnitt 7 enthält einige weitere RelView-Programme, die geeignet sind, die vielfältige Verwendbarkeit des Systems zu demonstrieren. Im Gegensatz zu den Abschnitten 5 und 6 verzichten wir jedoch auf formale Herleitungen. In Abschnitt 8 zeigen wir auf, wie durch eine Implementierung von Relationen mittels geordneter binärer Entscheidungsdiagramme die Effizienz der relationalen Spezifikationen und Programme teils beträchtlich erhöht werden kann. Schließlich gehen wir in Abschnitt 9 noch auf die bisher mit dem Ansatz und dem System gemachten Erfahrungen und zukünftige Arbeiten ein.

2 Relationenalgebra

Eine Relation R mit Urbildbereich X und Bildbereich Y ist eine Teilmenge des direkten Produkts $X \times Y$. Im folgenden bezeichnen wir mit $[X \leftrightarrow Y]$ die Menge (den Typ) dieser Relationen und schreiben $R : X \leftrightarrow Y$ statt $R \in [X \leftrightarrow Y]$. Falls Urbild- und Bildbereich nicht leer und endlich sind (ersteres nehmen wir im folgenden immer an, um pathologische Fälle zu vermeiden) und Kardinalität m bzw. n besitzen, so kann man R als Boolesche Matrix mit m Zeilen und n Spalten auffassen. Diese Matrixinterpretation ist sowohl zur graphischen Darstellung von Relationen als auch zur Beschreibung von relationalen Eigenschaften bzw. Manipulationen sehr gut geeignet. Sie wird auch in RelView verwendet. Aus diesem Grund schreiben wir im folgenden wie bei den Matrizen R_{xy} statt $\langle x, y\rangle \in R$ und übernehmen auch für Relationen die Matrix-Sprechweisen Zeilen und Spalten.

Wir setzen voraus, daß der Leser mit den folgenden Basisoperationen auf den Relationen vertraut ist: Vereinigung $R \cup S$, Durchschnitt $R \cap S$, Produkt (Komposition) $R\,S$, Negation (Komplement) $\overline{R}$ und Transposition R^T. Die Ordnung auf den Relationen $[X \leftrightarrow Y]$ gleichen Typs ist durch die Inklusion $R \subseteq S$ gegeben. Schließlich bezeichnen, polymorph getypt, O die leere Relation, L die Allrelation und I die identische Relation. Neben der eben aufgezählten Basis werden wir uns in diesem Bericht noch auf R^* (reflexiv-transitive Hülle) und R^+ (transitive Hülle) beziehen, sowie einige weitere relationale Konstruktionen an geeigneten Stellen einführen.

Für die eben aufgezählten Operationen und Konstanten gelten eine Fülle von komponentenfreien algebraischen Gesetzen, die man als den relationalen Kalkül bezeichnet. Wir setzen im folgenden fundamentale Gesetze als bekannt voraus, z. B.

$$\begin{array}{ll}
R^{\mathsf{T}\mathsf{T}} = R & R \subseteq S \Longrightarrow R^\mathsf{T} \subseteq S^\mathsf{T} \\
(R\,S)^\mathsf{T} = S^\mathsf{T}\,R^\mathsf{T} & \overline{R^\mathsf{T}} = \overline{R}^\mathsf{T} \\
R \subseteq S \Longrightarrow Q\,R \subseteq Q\,S & R \subseteq S \Longrightarrow R\,Q \subseteq S\,Q \\
Q\,(R \cap S) \subseteq Q\,R \cap Q\,S & Q\,(R \cup S) = Q\,R \cup Q\,S \\
(R \cap S)^\mathsf{T} = R^\mathsf{T} \cap S^\mathsf{T} & (R \cup S)^\mathsf{T} = R^\mathsf{T} \cup S^\mathsf{T}\,.
\end{array} \tag{1}$$

Weiterhin setzen wir voraus, daß dem Leser die grundlegendsten Eigenschaften von Relationen und ihre algebraischen Beschreibungen vertraut sind, wie beispielsweise Reflexivität ($\mathsf{I} \subseteq R$), Irreflexivität ($R \subseteq \overline{\mathsf{I}}$), Symmetrie ($R = R^\mathsf{T}$), Transitivität ($R\,R \subseteq R$), Eindeutigkeit ($R^\mathsf{T}\,R \subseteq \mathsf{I}$) und Totalität ($R\,\mathsf{L} = \mathsf{L}$). Speziellere Gesetze für und Eigenschaften von Relationen, die dem Leser vielleicht nicht geläufig sind, werden wir erst dort angegeben, wo sie benötigt werden. Bezüglich der Axiomatisierung des relationalen Kalküls, d.h. des Begriffs der (abstrakten) Relationenalgebra, müssen wir auf die Literatur verweisen, etwa auf [44, 23] oder die Bücher [41, 19].

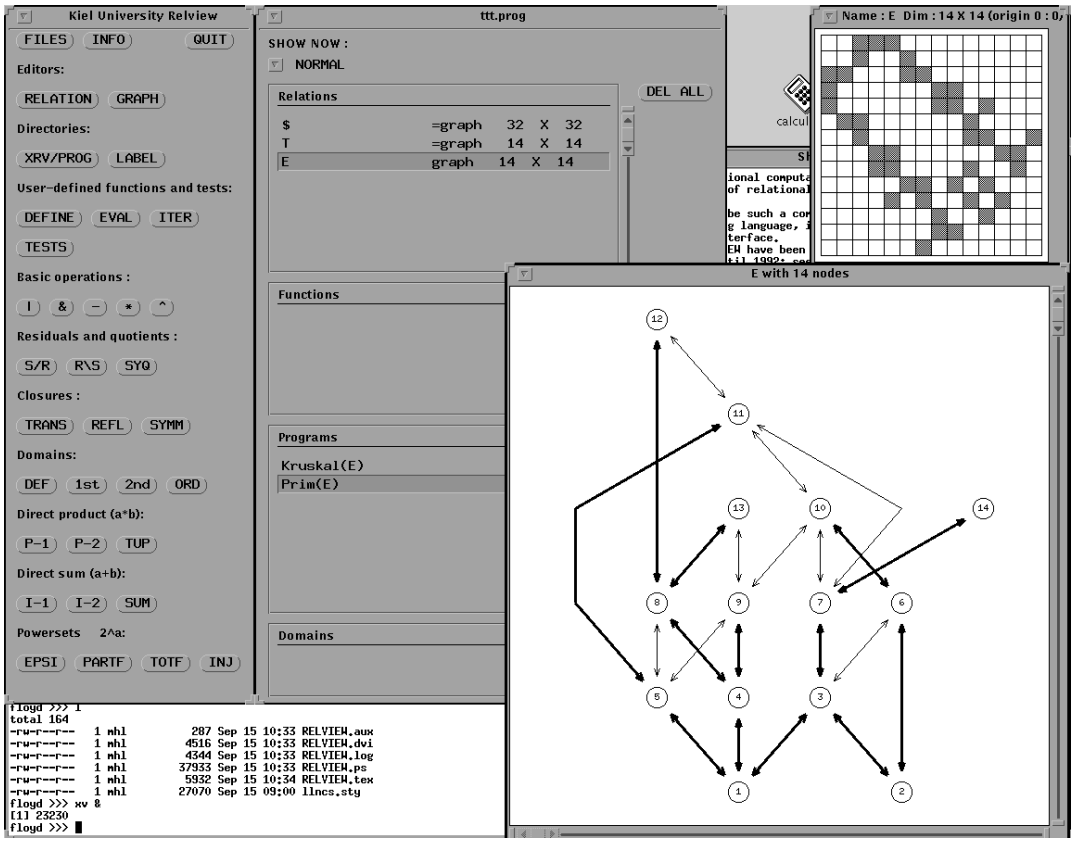
Relationenalgebra in ihrer Grundform kennt keinerlei mengentheoretische Operationen wie $x \in X$ und $X \subseteq Y$. Sie können jedoch einfach eingeführt werden, indem man Mengen durch *Vektoren* darstellt, das sind spezielle Relationen $v : X \leftrightarrow Y$, für die $v\,\mathsf{L} = v$ gilt. In der Matrixinterpretation heißt $v = v\,\mathsf{L}$, daß die Boolesche Matrix zeilenkonstant ist, d.h. jede Zeile nur

Nullen oder nur Einsen enthält. Damit wird der Bildbereich irrelevant. Im folgenden betrachten wir deshalb oft Vektoren $v : X \leftrightarrow \mathbf{1}$ mit einem speziellen einelementigen Bildbereich $\mathbf{1}$ und unterdrücken dann in der oben eingeführten Indexschreibweise den zweiten (konstanten) Index. So ein Bildbereich kann algebraisch durch $v^\mathsf{T}\, v = \mathsf{I}$ charakterisiert werden, denn $v^\mathsf{T}\, v = \mathsf{L}$ gilt immer und eine nichtleere Menge ist einelementig genau dann, wenn auf ihr L und I zusammenfallen. In der Matrixinterpretation heißt dies, daß wir uns auf Boolesche Spaltenvektoren beschränken. Ein Vektor $v : X \leftrightarrow \mathbf{1}$ stellt somit die Menge $\{x \in X : v_x\}$ dar.

3 Das Computersystem RelView

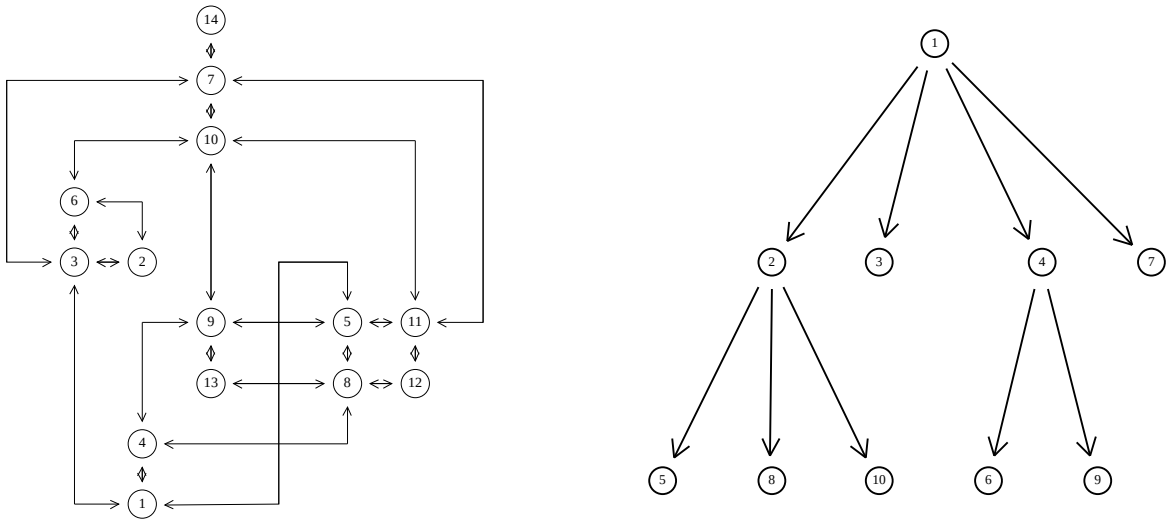
RelView ist ein in den letzten sechs Jahren an der Universität Kiel entwickeltes interaktives und bildschirmorientiertes Computersystem zum prototypischen Manipulieren von diskreten Strukturen, die auf Relationen basieren. Es ist in der Programmiersprache C geschrieben und läuft unter X-Windows mit intensiver Verwendung der graphischen Benutzeroberfläche. Das Kieler RelView System stellt eine Neuimplementierung und wesentliche Erweiterung des ursprünglich an der Universität der Bundeswehr München entwickelten Systems (siehe [11]) dar. Es ist weltweit in Gebrauch und wurde mehrmals auf anerkannten internationalen Tagungen vorgeführt, zuletzt auf der FASE '98 im Rahmen von ETAPS '98; siehe [6].

In RelView werden alle Daten als Relationen dargestellt, für die es wiederum zwei Visualisierungen gibt. Einerseits kann man Relationen auf dem Bildschirm als Boolesche Matrizen anzeigen und mit der Maus bzw. entsprechenden Kommandoknöpfen editieren. Stimmen Urbild- und Bildbereich überein, so ist für diese sogenannten homogenen Relationen auch eine Darstellung durch gerichtete Graphen möglich. Um einen Eindruck von der Benutzeroberfläche von RelView zu vermitteln, ist nachfolgend ein Bildschirm-Abzug angegeben.



In dieser Abbildung sehen wir links das Menüfenster des Systems, welches die Bedienungselemente zur Steuerung und die Kommandoknöpfe für benutzerdefinierte relationale Funktionen und vordefinierte relationale Operationen enthält. Daneben befindet sich das Verzeichnisfenster, das den jeweiligen Zustand des Systems angibt, also die definierten Relationen (derzeit `T` und `E`), Funktionen (derzeit keine), Programme (derzeit `Kruskal` und `Prim`) und Bereiche (derzeit keine). Die rechten beiden Fenster dienen schließlich der oben erwähnten Darstellung von Relationen als Graphen bzw. Boolesche Matrizen. Alle Fenster sind in ihrer Position und Größe zu verändern und die entsprechenden Voreinstellungen können über eine Datei den eigenen Vorstellungen angepaßt werden.

Für die Darstellung von Relationen durch gerichtete Graphen sind im RelView System eine Reihe von verschiedenen Verfahren zum „schönen“ Zeichnen von sowohl beliebigen als auch speziellen Graphen (wie DAGs, planaren Graphen oder Bäumen) implementiert. In der obigen Abbildung ist zu sehen, was einer der Zeichenalgorithmen von RelView, der schichtenweise vorgeht, als Graphdarstellung für die Relation `E` produziert. Dabei ist, um eine zusätzliche Eigenschaft des Systems hervorzuheben, in dem dargestellten Graphen noch ein gerichteter spannender Baum als Teilgraph durch die fetten Pfeile hervorgehoben. Das linke der nachfolgenden zwei Bilder zeigt eine andere Graphdarstellung der obigen Relation `E`, die in der Literatur auch orthogonale Zeichnung genannt wird, und das rechte Bild zeigt die Relation `T` des spannenden Baums, wie sie durch den Baumzeichen-Algorithmus von RelView dargestellt wird.



Das RelView-System kann gleichzeitig soviele Relationen als Boolesche Matrizen und Graphen verwalten, wie die Größe des Arbeitsspeichers erlaubt. Wie schon erwähnt, können Relationen mit der Maus und durch Kommandos manipuliert werden. Die Kommandos beinhalten insbesondere alle in Abschnitt 2 erwähnten Operationen auf den Relationen. Darüber hinaus ist es möglich, die den Kommandos entsprechenden Basisfunktionen des Systems zu relationalen Termen zu komponieren, aus denen man durch Abstraktion dann relationale Funktionen erhält. Durch Hinzunahme der grundlegendsten Kontrollstrukturen imperativer Programmiersprachen erreicht man schließlich relationale Programme, die ähnlich den Funktionsprozeduren von Pascal oder Modula-2 sind. Ein Beispiel für eine relationale Funktion ist nachfolgend angegeben:

$$tik(R) : [X \leftrightarrow X] \to [X \leftrightarrow X] \qquad tik(R) = R \cap \overline{R\,R^+}\,. \tag{2}$$

Ist R in der Graphinterpretation kreisfrei, was man relationenalgebraisch durch $R^+ \subseteq \bar{\mathsf{I}}$ ausdrücken kann, so berechnet diese Funktion den transitiv-irreduzible Kern von R, das ist die kleinste Relation S mit $S \subseteq R$ und $S^+ = R^+$. In der Syntax des RelView-Systems (mit den

Bezeichnungen `&`, `-` und `trans` für die Bildung von Durchschnitt, Negation und transitive Hülle) sieht die relationale Funktion *tik* aus (2) wie folgt aus:

```
tik(R) = R & -(R*trans(R)).
```

Das nachfolgend angegebene Beispiel für ein einfaches relationales Programm in RelView verwendet den senkrechten Strich `|` für die Vereinigungsoperation und das Zeichen `^` für die Operation zum Transponieren. Weiterhin benutzt es eine Basisfunktion `empty`, die testet, ob eine Relation leer ist. Durch das Programm `reach` werden für einen endlichen gerichteten Graphen mit Relation $R : X \leftrightarrow X$ und eine durch den Vektor $s : X \leftrightarrow \mathbf{1}$ dargestellte Knotenmenge die von dieser Menge aus erreichbaren Knoten in der Vektordarstellung $(R^{\mathsf{T}})^* s : X \leftrightarrow \mathbf{1}$ berechnet.

```
reach(R,s)
  DECL u, v
  BEG  u = s;
       v = -u & R^*u;
       WHILE -empty(v) DO
         u = u | v;
         v = -u & R^*v OD
       RETURN u
  END.
```

An dieser Stelle muß auf eine Besonderheit von RelView hingewiesen werden. Wie schon erwähnt, werden alle Daten als Relationen dargestellt. Insbesondere entsprechen die beiden einzigen Relationen O, L von $[\mathbf{1} \leftrightarrow \mathbf{1}]$ den Wahrheitswerten „falsch“ und „wahr“. Diese Darstellung ist motiviert durch ein Resultat aus [12]. Hier wird, nur auf die Basisoperationen von Abschnitt 2 aufbauend, in expliziter Weise eine einfache relationale Funktion zum Testen von Relationeninklusion angegeben, d.h.:

$$incl : [X \leftrightarrow Y] \times [X \leftrightarrow Y] \to [\mathbf{1} \leftrightarrow \mathbf{1}] \qquad R \subseteq S \iff incl(R,S) = \mathsf{L}. \tag{3}$$

Diese Funktion ist in RelView als Basisfunktion gleichen Namens vorhanden. Als eine Konsequenz kann man im System alle aussagenlogischen Formeln über Inklusionen ausdrücken, beispielsweise einen Gleichheitstest durch die RelView-Funktion

```
eq(R,S) = incl(R,S) & incl(S,R),
```

da auf den zwei Relationen L und O von $[\mathbf{1} \leftrightarrow \mathbf{1}]$ die Operationen $\cup, \cap, \overline{\ }$ und $\subseteq$ genau den logischen Junktoren $\vee, \wedge, \neg$ und $\to$ entsprechen.

Eine Beschreibung von RelView, die auch ein Benutzermanual und eine Sammlung von Anwendungsbeispielen enthält, ist [5]. Für weitere Informationen, insbesondere weitere Beispiele, sei auf die Web-Seite `http://www.informatik.uni-kiel.de/~progsys/relview.html` zum RelView-System an der Universität Kiel verwiesen.

4 Relationale Modellierung von Datenstrukturen

Wir haben schon erwähnt, daß es mittels Relationen einfach möglich ist, die beiden Wahrheitswerte und auch Teilmengen eines Universums zu modellieren. Letzteres führt unmittelbar auch zur Modellierung von einzelnen Elementen einer Menge, da diese genau den einelementigen Teilmengen entsprechen. In der Literatur werden die speziellen, einelementige Teilmengen

darstellende Vektoren auch *(relationale) Punkte* genannt. Sie entsprechen in der Matrixinterpretation denjenigen zeilenkonstanten Booleschen Matrizen, die genau eine Zeile besitzen, die nur Einsen enthält. Die relationenalgebraische Charakterisierung eines Punkts p ist gegeben durch $p = p\,\mathsf{L}$ (Vektoreigenschaft), $p\,p^{\mathsf{T}} \subseteq \mathsf{I}$ (Injektivität) und $p^{\mathsf{T}}\,\mathsf{L} = \mathsf{L}$ (Surjektivität); daraus herleitbare Rechenregeln findet man beispielsweise in [41].

Verwendet man Vektoren und Punkte zur Modellierung von Mengen und Elementen, so entsprechen sich genau die Booleschen Operationen der relationalen und der mengentheoretischen Seite. Stellt also beispielsweise der Vektor $v : X \leftrightarrow \mathbf{1}$ eine Teilmenge Y von X dar und der Punkt $p : X \leftrightarrow \mathbf{1}$ ein Element x von X, so modellieren trivialerweise $v \cup p$ das Einfügen von x in Y und $v \cap \overline{p}$ das Entfernen von x aus Y. Neben den Booleschen Operationen haben sich noch zwei weitere Operationen auf Mengen als sehr bedeutsam für die Formulierung von relationalen Spezifikationen und Programmen herausgestellt. Die erste betrifft die Auswahl eines Elements aus einer nichtleeren Menge. Relational wird diese modelliert durch eine Funktion

$$\mathit{point} : [X \leftrightarrow \mathbf{1}] \to [X \leftrightarrow \mathbf{1}]\,, \tag{4}$$

die zu einem nichtleeren Vektor $v : X \leftrightarrow \mathbf{1}$ einen Punkt $\mathit{point}(v)$ mit $\mathit{point}(v) \subseteq v$ liefert. Die zweite Operation betrifft die injektive Einbettung einer nichtleeren Menge Y in eine Obermenge X mittels der identischen Funktion $\mathit{id} : Y \to X$. Ist Y durch einen nichtleeren Vektor $v : X \leftrightarrow \mathbf{1}$ beschrieben, so ergibt sich diese Funktion als eindeutige und totale Relation $\mathit{inj}(v) : Y \leftrightarrow X$, indem man aus der identischen Relation $\mathsf{I} : X \leftrightarrow X$ all jene Zeilen entfernt, die in v einen 0-Eintrag besitzen. Sowohl *point* aus (4) als auch die relationale Funktion

$$\mathit{inj} : [X \leftrightarrow \mathbf{1}] \to [Y \leftrightarrow X] \tag{5}$$

sind in RelView vorhanden. Ihre relationenalgebraische Charakterisierung findet man in der Beschreibung [5] des Systems.

Das Gegenstück zur relationalen Funktion *point* auf der Ebene der allgemeinen Relationen ist gegeben durch

$$\mathit{atom} : [X \leftrightarrow Y] \to [X \leftrightarrow Y]\,. \tag{6}$$

Mittels dieser, auch in RelView vorhandenen, Funktion wird zu einer nichtleeren Relation R ein in R enthaltenes Atom $\mathit{atom}(R)$ berechnet, also eine Relation $\mathit{atom}(R) \subseteq R$, die genau ein Paar enthält. Diese letzte Eigenschaft kann man beispielsweise dadurch beschreiben, daß man sowohl den Definitionsbereich $\mathit{atom}(R)\,\mathsf{L}$ als auch den Wertebereich $\mathit{atom}(R)^{\mathsf{T}}\,\mathsf{L}$ von $\mathit{atom}(R)$ als Punkt fordert. In der graphentheoretischen Interpretation von Relationen wird durch den Aufruf $\mathit{atom}(R)$ ein einzelner Pfeil des nichtleeren gerichteten Graphen $g = (X, R)$ geliefert, was in zahlreichen Algorithmen Anwendung findet.

Weiterhin als sehr bedeutend bei relationaler Spezifikation und Programmentwicklungen haben sich die Bildung von direkten Produkten und direkten Summen herausgestellt. Ihre relationale Modellierung erfolgt im ersten Fall durch die beiden Projektionen $\pi_1 : X \times Y \leftrightarrow X$ und $\pi_2 : X \times Y \leftrightarrow Y$ und im zweiten Fall durch die beiden Injektionen $\imath_1 : X \leftrightarrow X + Y$ und $\imath_2 : Y \leftrightarrow X + Y$. RelView erlaubt die Deklaration von direkten Produkten und Summen und stellt auch Basisfunktionen zur Berechnung der Projektionen, Injektionen und einigen weiteren in diesem Zusammenhang wichtigen Konstruktionen zur Verfügung. Bezüglich der Details und auch einer relationenalgebraischen Charakterisierung der Projektionen und Injektionen verweisen wir wiederum auf die Beschreibung [5] des RelView-Systems.

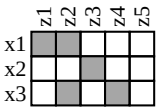
Durch die Verwendung von direkten Produkten wird es insbesondere möglich, jede beliebige Formel der Logik erster Stufe in einen gleichwertigen relationalen Term zu überführen. Dies ist sehr bedeutend zur Gewinnung von relationalen Spezifikationen. Mittels direkter Summen ist es

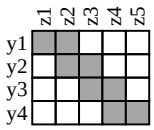
beispielsweise möglich, bipartite Graphen zu modellieren, wie sie etwa Petrinetzen zugrundeliegen. Eine weitere Anwendung ist die relationale Modellierung von Sequenzen von Mengen, die wir nun in der sehr einfachen Form angeben, wie wir sie später brauchen werden.

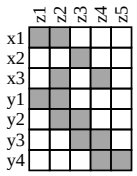
Wir benötigen im folgenden das Konzept der relationalen Summe. Sind $R : X \leftrightarrow Z$ und $S : Y \leftrightarrow Z$ zwei Relationen mit gleichem Bildbereich, so kann mit Hilfe der beiden Injektionen $\imath_1 : X \leftrightarrow X + Y$ und $\imath_2 : Y \leftrightarrow X + Y$ ihre Summe $R + S : X + Y \leftrightarrow Z$ definiert werden durch die Gleichung

$$R + S = \imath_1{}^{\mathsf{T}}\, R \cup \imath_2{}^{\mathsf{T}}\, S\,. \tag{7}$$

Die mittels (7) definierte relationale Summe verhält sich wie die Relation R für alle Elemente, die aus X kommen, und wie die Relation S für alle Elemente, die aus Y kommen. Nachfolgend sind in der Darstellung des RelView-Systems, in welchem natürlich auch die relationale Summe als Basisfunktion + vorhanden ist, zwei Relationen R und S sowie ihre Summe $R+S$ angegeben. Zum besseren Verständnis der Konstruktion sind in den drei Bildern zusätzlich noch die Zeilen und Spalten der Booleschen Matrizen mit den Elementen der Urbildbereiche bzw. des Bildbereichs markiert.







Kehren wir nun zur relationalen Modellierung von Sequenzen von Mengen zurück. Man kann eine Relation $R : X \leftrightarrow Y$ mit $|Y| = n$ auch als eine Sequenz von Vektoren $v_i : X \leftrightarrow \mathbf{1}$ auffassen, wobei der Vektor v_i als die i-te Spalte von R definiert ist ($1 \le i \le n$). In dieser Sichtweise modelliert R die nichtleere Sequenz der durch ihre Spalten dargestellten Mengen. Der Konkatenationsoperation auf den Sequenzen entspricht dann auf relationaler Ebene die Funktion

$$\mathit{conc} : [X \leftrightarrow Y] \times [X \leftrightarrow Z] \to [X \leftrightarrow Y + Z] \qquad \mathit{conc}(R, S) = (R^{\mathsf{T}} + S^{\mathsf{T}})^{\mathsf{T}}\,. \tag{8}$$

Die RelView-Version von (8) werden wir später nur für den Spezialfall benötigen, daß wir einen Vektor $v : X \leftrightarrow \mathbf{1}$, der eine Menge von Knoten eines gerichteten Graphen $g = (X, R)$ darstellt, von rechts als zusätzliche Spalte an eine Relation anfügen, was der bekannten Sequenzoperation „postfix“ entspricht. Damit werden wir schließlich eine Relation $S : X \leftrightarrow \{1, \ldots, n\}$ erhalten, die eine Sequenz $Y_1, \ldots, Y_n$ von Knotenmengen beschreibt.

Der eben beschriebene Ansatz kann aber auch verwendet werden, um beispielsweise Sequenzen von Pfeilmengen eines gerichteten Graphen $g = (X, R)$ zu modellieren. Dies ist dadurch begründet, man vergleiche mit [41], daß jede Pfeilmenge durch eine Teilrelation S von R dargestellt wird und zwischen den Relationen $[X \leftrightarrow X]$ und den Vektoren $[X \times X \leftrightarrow \mathbf{1}]$ eine eineindeutige Beziehung besteht. Die erste Richtung ist gegeben durch die relationale Funktion

$$\mathit{rel2vec} : [X \leftrightarrow X] \to [X \times X \leftrightarrow \mathbf{1}] \qquad \mathit{rel2vec}(R) = (\pi_1\, R \cap \pi_2)\, \mathsf{L}\,, \tag{9}$$

wobei π_1 und π_2 die beiden Projektionen des direkten Produkts $X \times X$ sind. Auch die andere Richtung kann mit Hilfe von π_1 und π_2 wie folgt beschrieben werden:

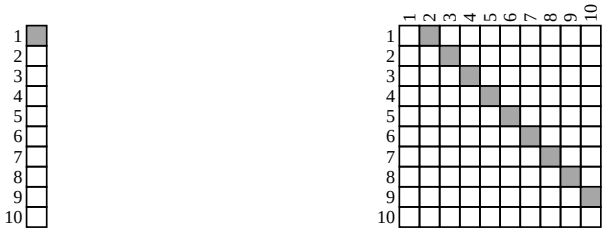
$$\mathit{vec2rel} : [X \times X \leftrightarrow \mathbf{1}] \to [X \leftrightarrow X] \qquad \mathit{vec2rel}(v) = \pi_1{}^{\mathsf{T}}\, (\pi_2 \cap v\, \mathsf{L})\,. \tag{10}$$

Als letzte Datenstruktur modellieren wir nun noch die Erzeugung einer endlichen Menge X durch ein initiales Element und eine Nachfolgeroperation. Diese Konstruktion hat sich ebenfalls als sehr bedeutend für den von uns gewählten relationalen Ansatz zur Programmentwicklung

erwiesen. Wir orientieren uns an der Charakterisierung der natürlichen Zahlen durch eine relationale Variante der Peano-Axiome, wie sie beispielsweise in [9] angegeben ist, modellieren also das initiale Element und die Nachfolgeroperation durch ein Paar

$$\mathit{init} : X \leftrightarrow \mathbf{1} \qquad\qquad \mathit{succ} : X \leftrightarrow X\,, \tag{11}$$

wobei *init* ein Punkt und *succ* eine eindeutige und injektive Relation mit $(\mathit{succ}^\mathsf{T})^* \mathit{init} = \mathsf{L}$ sind. Die Gleichung $(\mathit{succ}^\mathsf{T})^* \mathit{init} = \mathsf{L}$ beschreibt, daß von dem durch *init* dargestellten Element jedes Element von X bezüglich der Nachfolgerrelation *succ* erreichbar ist. Sowohl *init* als auch *succ* sind in RelView mittels Basisfunktionen gleichen Namens realisiert. Im Gegensatz zur Typisierung in (11) benötigen diese jedoch noch einen Vektor als Argument, durch dessen Urbildbereich die zu erzeugende Menge definiert ist. Beispielsweise produziert RelView mittels der Aufrufe `init(v)` und `succ(v)` für einen beliebigen Vektor `v` mit 10 Spalten den folgenden initialen Punkt und die folgende Nachfolgerrelation:



Ist die Menge X durch *init* und *succ* endlich erzeugt und $p : X \leftrightarrow \mathbf{1}$ ein Punkt, der ein Element x von X darstellt, so bezeichnet $\mathit{succ}^\mathsf{T}\, p$ als Punkt den Nachfolger von x, falls ein solcher existiert. Andernfalls ist $\mathit{succ}^\mathsf{T}\, p$ leer. Dies führt zu einer relationalen Nachfolger-Funktion

$$\mathit{next} : [X \leftrightarrow \mathbf{1}] \to [X \leftrightarrow \mathbf{1}] \qquad\qquad \mathit{next}(p) = \mathit{succ}^\mathsf{T}\, p \tag{12}$$

die nur für Punkte definiert ist. Sie ist im RelView-System unter dem gleichen Namen ebenfalls als Basisfunktion enthalten.

5 Von Problemen zu relationalen Spezifikationen

Die Gewinnung einer relationalen Spezifikation ist der erste Schritt beim Prototyping in dem in diesem Bericht gewählten Anwendungsbereich. Man startet dazu oft mit einer prädikatenlogischen Problembeschreibung, die man durch Formalisierung der in der Regel natürlichsprachlich gestellten Aufgabe gewinnt. Dann transformiert man die logische Formel in eine relationale Form. Dies geschieht schrittweise durch die Anwendung von einfachen Beziehungen zwischen den relationalen Basisoperationen und den Junktoren und Quantoren der Logik, wodurch nach und nach logische durch relationale Konstrukte ersetzt werden. Die erreichte relationale Form ist oft ein indizierter relationaler Term, was durch Abstraktion zu einer relationalen Funktion und einem anschließenden Übergang nach RelView zu einem Prototyp in Gestalt einer relationalen Spezifikation führt.

Die Ersetzung der logischen Junktoren Disjunktion, Konjunktion usw. durch Vereinigung, Durchschnitt usw. auf relationaler Seite ist nach Definition der relationalen Operationen offensichtlich. Wir behandeln deshalb nachfolgend nur noch die Ersetzung von Quantoren durch gleichwertige relationale Konstruktionen. Aufgrund von

$$(R\,S)_{xy} \iff \exists\, z : R_{xz} \wedge R_{zy} \tag{13}$$

lassen sich Existenzquantoren mittels Produktbildung entfernen. Die wichtigsten Regeln zur Elimination von Allquantoren stützen sich auf die relationalen *Residuen* $S \,/\, R = \overline{\overline{S}\; R^{\mathsf{T}}}$ (Linksresiduum) und $R \setminus S = \overline{R^{\mathsf{T}}\; \overline{S}}$ (Rechtsresiduum). Aus (13) folgen nämlich die Beziehungen

$$(S \,/\, R)_{yx} \iff \forall\, z : R_{xz} \to S_{yz} \qquad (R \setminus S)_{xy} \iff \forall\, z : R_{zx} \to S_{zy}\,. \tag{14}$$

Eine Kombination der beiden Residuen führt zur Definition des *symmetrischen Quotienten* $syq(R,S) = (R \setminus S) \cap (R^{\mathsf{T}} \,/\, S^{\mathsf{T}})$. Damit erhalten wir aus (14) unmittelbar die Eigenschaft

$$syq(R,S)_{xy} \iff \forall\, z : R_{zx} \leftrightarrow S_{zy}\,. \tag{15}$$

Im folgenden demonstrieren wir die schrittweise Herleitung von relationalen Spezifikationen nach der oben skizzierten Methode anhand von einigen einfachen Beispielen aus der Graphentheorie; für weitere solchgeartete Beispiele vergleiche man mit [12, 14][1]. Beispiele aus anderen Bereichen für dieses Vorgehen findet man in [10] (Verbandstheorie), [46, 13, 7] (Petrinetze) und [35] (endliche Automaten).

5.1 Spezifikation der Quellen. Gegeben seien ein gerichteter Graph $g = (X, R)$ mit Knotenmenge X und Relation (Pfeilmenge) $R : X \leftrightarrow X$ und eine Teilmenge s von Knoten. Wir wollen die *Quellen* von s bestimmen, d.h. diejenigen Knoten von s, die keinen Vorgänger in s besitzen. Um einen Prototyp zu erhalten, starten wir mit der prädikatenlogischen Formalisierung

$$x \in s \wedge \forall\, y : R_{yx} \to y \notin s \tag{16}$$

der Eigenschaft, daß der Knoten x eine Quelle in der Menge s ist. Dann stellen wir die gegebene Teilmenge als Vektor $s : X \leftrightarrow \mathbf{1}$ dar und bekommen die das Problem spezifizierende Formel (16) in der äquivalenten Form $s_x \wedge \forall\, y : R_{yx} \to \overline{s}\, y$. Eine Elimination des Quantors in dieser Formel ist durch (14) möglich und bringt

$$(s \cap (R \setminus \overline{s}))_x\,. \tag{17}$$

Abstrahieren wir in (17) nach dem Index x, so erhalten wir in der Sprache des RelView-Systems, welche auch die beiden Residuen zur Verfügung stellt, schließlich

```
sources(R,s) = s & (R \ -s)
```

als relationale Spezifikation zur Berechnung der Quellen einer Menge von Knoten eines gerichteten Graphen. □

Wir haben im letzten Beispiel aus einer prädikatenlogischen Problembeschreibung eine relationale Funktion zur Berechnung eines Vektors hergeleitet. In graphentheoretischer Sprechweise haben wir also eine Menge von Knoten spezifiziert. Nachfolgend demonstrieren wir nun anhand eines einfachen Beispiels, wie man auf die gleiche Weise eine Relation berechnen kann, was graphentheoretisch der Spezifikation einer Pfeilmenge entspricht.

5.2 Spezifikation der isolierten Pfeile. Wir betrachten das Problem, in einem gerichteten Graphen $g = (X, R)$ die Menge aller isolierten Pfeile zu bestimmen, d.h. aller Pfeile, die mit keinem anderen Pfeil einen Knoten gemeinsam haben. Ein Paar x, y von Knoten bildet genau dann einen isolierten Pfeil, falls die Formel

$$R_{xy} \wedge (\forall\, z : R_{xz} \vee R_{zx} \to z = y) \wedge (\forall\, z : R_{yz} \vee R_{zy} \to z = x) \tag{18}$$

[1] Dabei setzen wir voraus, daß die zu behandelnden Graphen immer endlich sind. Weiterhin nehmen wir an, daß der Leser die grundlegendsten Begriffe der Graphentheorie kennt; ansonsten verweisen wir z.B. auf [2, 31].

gilt. Formen wir die beiden Quantifizierungen von (18) um zu $\forall z : (R \cup R^\mathsf{T})_{zx} \to \mathsf{I}_{zy}$ bzw. zu $\forall z : (R \cup R^\mathsf{T})_{yz} \to \mathsf{I}_{xz}$, so werden die Beziehungen (14) anwendbar und wir bekommen schließlich die gleichwertige Version

$$(R \cap ((R \cup R^\mathsf{T}) \setminus \mathsf{I}) \cap (\mathsf{I} \,/\, (R \cup R^\mathsf{T})))_{xy} \,. \tag{19}$$

Damit sind wir im Prinzip fertig. Denn abstrahieren wir in dem indizierten relationalen Term (19) nach den beiden Indizes x und y und gehen dann zu RelView über, so erhalten wir

```
isolated(R) = R & ((R | R^) \ I(R)) & (I(R) / (R | R^))
```

als einen Prototyp zur Berechnung derjenigen Relation, welche genau die Knotenpaare enthält, die einen isolierten Pfeil bilden. In dieser RelView-Funktion berechnet `I(R)` eine identische Relation mit gleicher Zeilen- und Spaltenanzahl wie `R`. □

Im nächsten Beispiel benötigen wir das mengentheoretische Symbol $\in$ auch als Relation auf der Objektebene. Wir setzen also $\varepsilon : X \leftrightarrow 2^X$ so voraus, daß für alle Elemente x aus X und Teilmengen s von X die Beziehung ε_{xs} genau dann gilt, wenn x ein Element von s ist. Weiterhin verwenden wir noch die reflexiv-transitive Hülle. Für beide Konstrukte stellt RelView entsprechende Basisfunktionen bereit. Zur Berechnung der Potenzmengenrelation $\varepsilon : X \leftrightarrow 2^X$ gibt es im System eine Basisfunktion `epsi`. Diese hat, analog zur Basisfunktion `init` zur Berechnung des Punkts *init* aus (11), einen Vektor als Argument, welcher durch seinen Urbildbereich die Menge X spezifiziert. Die reflexiv-transitive Hülle R^* kann mittels der Basisfunktionen `trans` für die transitive Hülle R^+ und `refl` für die reflexive Hülle $R \cup \mathsf{I}$ berechnet werden.

5.3 Spezifikation der Knotenbasen. In einem gerichteten Graphen heißt eine inklusionsminimale Menge von Knoten, von der aus alle anderen Knoten erreichbar sind, eine *Knotenbasis*. Wir wollen einen RelView-Prototyp zur Berechnung aller Knotenbasen eines gerichteten Graphen $g = (X, R)$ entwickeln. Dazu starten wir mit der Formel

$$\forall\, x \;\exists\, y : y \in b \wedge R^*_{yx} \,, \tag{20}$$

die besagt, daß von der Menge b aus jeder Knoten von g erreichbar ist. Dann verwenden wir die Definition von ε zusammen mit (13) und (14) und transformieren (20) in die gleichwertige Form

$$((\varepsilon^\mathsf{T}\, R^*) \,/\, \mathsf{L})_b \tag{21}$$

mit $\mathsf{L} : \mathbf{1} \leftrightarrow X$, in der Matrixinterpretation also einem Booleschen Zeilenvektor. Abstraktion in (21) nach dem Index b zeigt, daß durch den Vektor $(\varepsilon^\mathsf{T}\, R^*) \,/\, \mathsf{L} : 2^X \leftrightarrow \mathbf{1}$ die Menge aller Teilmengen von X dargestellt wird, von denen aus jeder Knoten erreichbar ist. In RelView führt diese Abstraktion unter Verwendung einer relationalen Funktion

```
rtc(R) = refl(trans(R))
```

zur Berechnung der reflexiv-transitiven Hülle zu der folgenden relationalen Spezifikation:

```
allreach(R) = (epsi(Ln1(R))^*rtc(R)) / L1n(R).
```

In der RelView-Funktion `allreach` berechnet `epsi(Ln1(R))` die passende Potenzmengenrelation ε. Dabei liefert der Aufruf `Ln1(R)` der Basisfunktion `Ln1` einen einspaltigen Allvektor mit gleicher Zeilenanzahl wie `R`, der den Urbildbereich der Potenzmengenrelation festlegt. Weiterhin wird durch `L1n(R)` ein passender transponierten Allvektor `L` (man vergleiche mit (21)) bestimmt.

Nun beachten wir, daß zu einer irreflexiven Ordnungsrelation und einem Vektor den minimalen Elementen der durch den Vektor dargestellten Menge genau die Quellen in einer graphentheoretischen Auffassung der Ordnung entsprechen. Unter Verwendung einer Relation `incl` für die Mengeninklusion, einer passenden identischen Relation `I`, der RelView-Spezifikation `sources` aus dem Beispiel 5.1 und `allreach` erhalten wir also

```
base(R) = sources(incl & -I, allreach(R))
```

als RelView-Funktion zur Berechnung des Vektors der Knotenbasen. Der Rest wird nun durch (14) erledigt, denn diese Beziehung zeigt, daß die Mengeninklusion als Relation auf der Objektebene gleich dem Residuum $\varepsilon \setminus \varepsilon$ ist, was eine Berechnung von `incl` mittels der Basisfunktionen `epsi` und `\` erlaubt. □

Wir haben im letzten Beispiel die relationale Spezifikation des Problems der Knotenbasis in der Form von drei RelView-Funktionen angegeben, um möglichst nahe bei den mathematischen Funktionen zu bleiben. Um Mehrfachberechnungen der Potenzmengerelation zu vermeiden, bietet sich natürlich an, statt der Funktionen ein relationales Programm mit einer Variablen zur Speicherung der Potenzmengenrelation zu verwenden.

Auch im nächsten Beispiel benötigen wir wiederum das mengentheoretische Symbol $\in$ als Potenzmengenrelation $\varepsilon : X \leftrightarrow 2^X$ auf der Objektebene und die reflexiv-transitive Hülle. Weiterhin kommt noch der symmetrische Quotient ins Spiel, der in RelView als Basisfunktion `syq` vorhanden ist.

5.4 Spezifikation der starken Zusammenhangskomponenten. Gegeben sei ein gerichteter Graph $g = (X, R)$. Eine starke Zusammenhangskomponente c von g kann durch die folgende Eigenschaft beschrieben werden: Es gibt einen Knoten x, so daß jeder Knoten y genau dann in c enthalten ist, wenn er von x aus erreichbar ist und x von ihm aus erreichbar ist. Als Formel schreibt sich dies wie folgt:

$$\exists\, x\ \forall\, y : y \in c \leftrightarrow (R^*_{xy} \wedge R^*_{yx}). \tag{22}$$

Unter Verwendung der Definitionen von ε und des symmetrischen Quotienten, siehe (15), können wir (22) gleichwertig durch $\exists\, x : syq(\varepsilon, R^* \cap (R^*)^{\mathsf{T}})_{cx}$ beschreiben. Eine Entfernung des verbleibenden Quantors erlaubt letztendlich (13) in Verbindung mit einem Vektor $\mathsf{L} : X \leftrightarrow \mathbf{1}$. Das Resultat ist der indizierte relationale Term

$$(syq(\varepsilon, R^* \cap (R^*)^{\mathsf{T}})\, \mathsf{L})_c\,. \tag{23}$$

Die Abstraktion von (23) nach dem Index c liefert $syq(\varepsilon, R^* \cap (R^*)^{\mathsf{T}})\, \mathsf{L} : 2^X \leftrightarrow \mathbf{1}$ als Vektor zur Darstellung der starken Zusammenhangskomponenten von g. In RelView führt dies schließlich zur relationalen Spezifikation

```
scc(R) = syq(epsi(Ln1(R)),rtc(R) & rtc(R)^)*Ln1(R),
```

wobei noch, zur Verbesserung der Effizienz, die Multiplikation mit dem Allvektor von rechts durch einen Aufruf der Basisfunktion `dom` ersetzt werden kann, die den Definitionsbereich einer Relation als Vektor berechnet. □

Der bisher vorgestellte Ansatz zur Gewinnung von ausführbaren relationalen Spezifikationen ist natürlich nicht immer so einfach anwendbar. Seine Anwendung kann auch in komplexen und deshalb nur schwer zu durchschauenden Spezifikationen enden. Letzteres ist häufig der Fall bei relationalen Spezifikationen, die Gebrauch von direkten Produkten machen. Beispiele

findet man in [12]. Nachfolgend geben wir nun ein Beispiel an, wo der bisherige Ansatz über eine logische Formel und einen indizierten relationalen Term zur Verwendung von direkten Produkten und zu einer komplexen Spezifikation führen würde, die Verwendung von direkten Summen hingegen eine einfache und auch einigermaßen effizient ausführbare relationale Spezifikation erlaubt. Dieses Beispiel berechnet einen Vektor als n-fache relationale Summe von einzelnen Wahrheitswerten und ist leicht zu verallgemeinern. Es macht für eine Relation $R : X \leftrightarrow X$ und einen Vektor $v : X \leftrightarrow \mathbf{1}$ Gebrauch von der Konstruktion $inj(v)\, R\, inj(v)^\mathsf{T}$, welche wir, zur Verbesserung der Lesbarkeit, mit R_v abkürzen. Ist R die Relation eines Graphen $g = (X, R)$ und kennzeichnet der Vektor v eine Teilmenge Y der Knoten, so ist R_v genau die Relation des von Y erzeugten Untergraphen.

5.5 Spezifikation der Trennknoten. Es sei ein ungerichteter Graph $g = (X, R)$ vorausgesetzt, d.h. $R : X \leftrightarrow X$ ist eine symmetrische und irreflexive Relation. Ein Knoten x heißt ein *Trennknoten* (man findet in der Literatur auch die Bezeichnung *Artikulationsknoten*) von g, wenn der Untergraph von g, der durch das Entfernen von x entsteht, mindestens eine Zusammenhangskomponente mehr als g besitzt. Aus dieser Definition folgt unmittelbar, daß x ein Trennknoten von g genau dann ist, wenn es zwei verschiedene Knoten y und z aus $X \setminus \{x\}$ gibt, so daß x auf jedem Weg von y nach z in g liegt.

Zur Entwicklung einer relationalen Spezifikation für den Vektor der Trennknoten stellen wir den Knoten x durch einen Punkt $p : X \leftrightarrow \mathbf{1}$ dar. Weiterhin betrachten wir die Relation $R_{\overline{p}}$ des durch die Knotenmenge $X \setminus \{x\}$ erzeugten Untergraphen von g. Dann gilt die Inklusion

$$(R_{\overline{p}})^+ \cap \overline{\mathsf{I}} \subseteq (R^+)_{\overline{p}} \cap \overline{\mathsf{I}}\,, \tag{24}$$

denn die linke Seite von (24) setzt zwei Knoten y und z aus $X \setminus \{x\}$ genau dann in Beziehung, wenn $y \neq z$ gilt und ein Weg von y nach z im dem von $X \setminus \{x\}$ erzeugten Untergraphen existiert, während die rechte Seite von (24) zwei Knoten y und z aus $X \setminus \{x\}$ genau dann in Beziehung setzt, wenn $y \neq z$ gilt und ein Weg von y nach z im Originalgraphen g existiert. Diese Überlegungen zeigen auch, daß die Umkehrung der Inklusion (24) exakt ausdrückt, daß es verschiedene Knoten y und z aus $X \setminus \{x\}$ sowie einen Weg von y nach z in g gibt, auf dem x nicht liegt.

Wechseln wir nun von den Knoten des Graphen zu den relationalen Punkten, so erhalten wir durch Kontraposition des bisher erzielten Resultats die Äquivalenz

$$(R_{\overline{p}})^+ \cap \overline{\mathsf{I}} \neq (R^+)_{\overline{p}} \cap \overline{\mathsf{I}} \iff p \text{ beschreibt einen Trennknoten}\,. \tag{25}$$

Mit Hilfe des auf den Inklusionstest *incl* aufbauenden Gleichheitstests *eq*, den wir in Abschnitt 3 in RelView formuliert haben, und einer relationalen Funktion $ip(R) = R \cap \overline{\mathsf{I}}$ zur Berechnung des irreflexiven Anteils von R bekommen wir also aus (25) sofort die relationale Funktion

$$iscut : [X \leftrightarrow X] \times [X \leftrightarrow \mathbf{1}] \to [\mathbf{1} \leftrightarrow \mathbf{1}] \qquad iscut(R, p) = \overline{eq(ip((R_{\overline{p}})^+), ip((R^+)_{\overline{p}}))} \tag{26}$$

zum Testen, ob ein einzelner Punkt einen Trennknoten beschreibt. Den gesuchten Vektor zur Darstellung aller Trennknoten können wir nun als relationale Funktion

$$cuts : [X \leftrightarrow X] \to [X \leftrightarrow \mathbf{1}] \qquad cuts(R) = iscut(R, init) + \ldots + iscut(R, next^n(init)) \tag{27}$$

spezifizieren, wobei $n+1$ die Mächtigkeit von X ist. Wir erzeugen also die endliche Knotenmenge X mittels des initialen Punkts *init* von (11) und der Nachfolger-Funktion *next* von (12) in Form einer Sequenz $next^i(init)$, $0 \leq i \leq n$, von Punkten, wenden die relationale Funktion *iscut* auf jeden dieser Punkte an und bilden schließlich die relationale Summe der erhaltenen

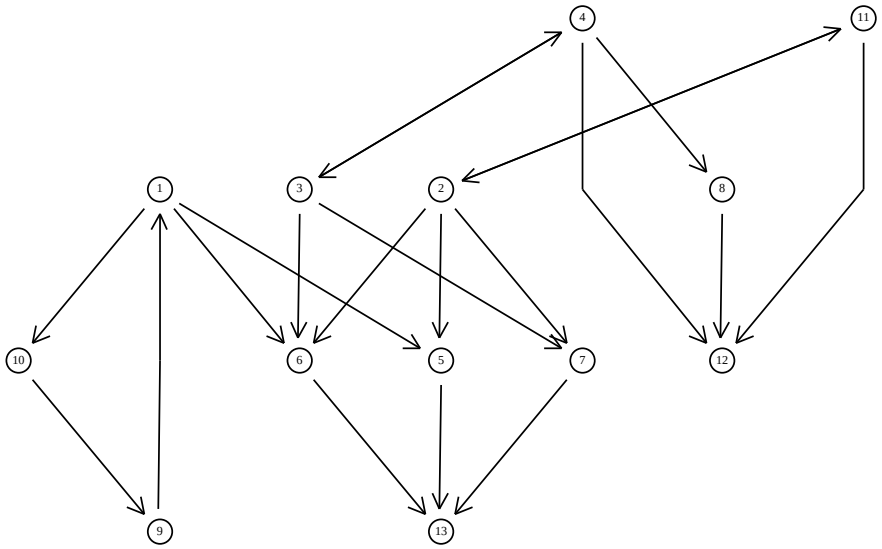
Wahrheitswerte. In RelView ist die Summation von (27) mittels einer Schleife möglich. Auf diese Weise bekommen wir schließlich

```
cuts(R)
  DECL ip(R) = R & -I(R);
       del(R,v) = inj(-v)*R*inj(-v)^;
       iscut(R,p) = -eq(ip(trans(del(R,p))),ip(del(trans(R),p)));
       c, p
  BEG  p = init(Ln1(R));
       c = iscut(R,p);
       WHILE -empty(next(p)) DO
         p = next(p);
         c = c + iscut(R,p) OD
       RETURN c
  END
```

als ein relationales Programm zur Berechnung des Vektors aller Trennknoten eines ungerichteten Graphen. □

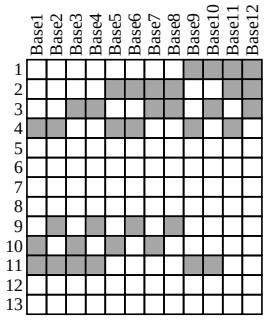
Aufbauend auf das zweite der eben vorgestellten Beispiele behandeln wir nun eine konkrete Anwendung von RelView. Wir verbinden dies damit, zu demonstrieren, wie man mit Hilfe des Systems auch Objekte höherer Stufe, in diesem Fall Vektoren mit einer Potenzmenge als Urbildbereich, auf anschauliche Weise darstellen kann.

5.6 Visualisierung von Potenzmengenvektoren. Wir betrachten den in dem folgenden Bild angegebenen gerichteten Graphen $g = (X, R)$:



Wenden wir die in Beispiel 5.3 entwickelte RelView-Funktion `base` auf die Relation dieses Graphen an, so liefert das System als Resultat auf dem Bildschirm einen die Knotenbasen darstellenden Vektor $v : 2^X \leftrightarrow \mathbf{1}$, also einen Booleschen Spaltenvektor mit 2^{13} Komponenten, in dem jede Komponente für ein Element von 2^X steht und genau die Knotenbasen mit 1 markiert sind. Daraus könnten wir dann die Knotenbasen erhalten, indem wir v komponentenweise mit den 2^{13} Spalten der Matrixdarstellung von $\varepsilon : X \leftrightarrow 2^X$ vergleichen. Praktisch ist das eben beschriebene Verfahren bei der gegebenen Größenordnung jedoch nicht mehr durchführbar. Und

tatsächlich gibt es in RelView auch eine viel einfachere Möglichkeit, Mengen von Mengen zu visualisieren. Im Fall der Knotenbasen berechnen wir zuerst durch die Basisfunktion `inj` die injektive Einbettung $inj(v) : \mathcal{B} \leftrightarrow 2^X$ von der durch v dargestellten Menge $\mathcal{B}$ der Knotenbasen in die Potenzmenge 2^X der Knoten. Dann transponieren wir $inj(v)$ und multiplizieren das Resultat von links mit der Potenzmengenrelation. Das Ergebnis $\varepsilon\, inj(v)^\mathsf{T} : X \leftrightarrow \mathcal{B}$ sieht in RelView, versehen mit einer geeigneten Markierung der Zeilen und Spalten, wie folgt aus:



Diese Boolesche Matrix setzt einen Knoten mit einer Knotenbasis genau dann in Beziehung, wenn der Knoten in der Knotenbasis enthalten ist. Ihre Spalten stellen somit, als Sequenz von einzelnen Vektoren betrachtet, genau die 12 Knotenbasen des obigen Graphen dar. □

Wir wollen zum Ende dieses Abschnitts noch auf die Bedeutung von RelView bei der Gewinnung von relationalen Spezifikation eingehen. Der Hauptanwendungsbereich des Systems ist hier nach unserer Erfahrung die Überprüfung der relationalen Problemspezifikation durch stichprobenartige Tests, wobei insbesondere auch Randfälle wie leere Graphen (Relationen) und Mengen (Vektoren) abgedeckt werden sollten. Prototypische Tests ermöglichen einen Vergleich der relationalen Spezifikation sowohl mit der ihrer Entwicklung zugrundeliegenden formal-logischen Spezifikation als auch mit der ganz ursprünglich in der Intuition verankerten und informellen Problemstellung. Durch den Vergleich der relationalen und der logischen Spezifikation kann man insbesondere die Korrektheit der Herleitung der ersten aus der zweiten überprüfen; man gewinnt also zusätzliche Sicherheit bezüglich der durchgeführten Rechnungen. So ist es etwa bei der Charakterisierung eines Trennknotens x wichtig, daß es zwei *verschiedene* Knoten y und z aus $X \setminus \{x\}$ gibt, so daß x auf jedem Weg von y nach z liegt. Verzichtet man auf die Forderung $y \neq z$, so werden auch Knoten mit nur einem Nachbarn zu Trennknoten, im Widerspruch zur originalen Forderung, daß durch das Entfernen von Trennknoten die Anzahl der Zusammenhangskomponenten sich echt vergrößert. Der stichprobenartige Vergleich der relationalen Spezifikation und der intuitiven Problemstellung hingegen erlaubt es zu überprüfen, ob die relationale Formalisierung die Intuition hinreichend genau beschreibt.

Nach den von uns gemachten Erfahrung ist es wesentlich, daß bei einem relationen-basierten Ansatz mehrere leicht verständliche und intuitive Visualisierungen von Relationen zur Verfügung stehen, die es erlauben, gleiche Sachverhalte auch unter verschiedenen Blickwinkeln zu betrachten. Bei RelView sind drei solche in der Form der Booleschen Matrizen, der gerichteten Graphen und der sogenannten Korrespondenzgraphen (siehe [5]) vorhanden, welche noch durch verschiedene Markierungsmechanismen für Knoten, Pfeile, Zeilen und Spalten erweitert werden. Um als Prototyping-System geeignet zu sein, sind natürlich auch noch andere Eigenschaften notwendig, die unabhängig vom Anwendungsbereich „Relationen“ sind. Dies betrifft insbesondere die enge Integration der einzelnen Komponenten des Systems, um verschränktes Editieren, Ausführen und Analysieren zu erlauben. Auch in dieser Hinsicht ist RelView unserer Meinung nach schon ziemlich ausgereift.

6 Von relationalen Spezifikationen zu relationalen Programmen

Entwickelt man eine relationale Spezifikation wie im letzten Abschnitt vorgestellt, so kann es in günstigen Fällen, wie etwa im Beispiel der Quellen eines gerichteten Graphen, durchaus vorkommen, daß die resultierende relationale Funktion mittels RelView effizient ausgewertet werden kann. Dann ist man im Prinzip fertig und kann den RelView-Prototyp in die gewünschte imperative Programmiersprache übertragen. Es gibt aber auch Fälle, in denen der Prototyp noch sehr ineffizient ist. Dies ist etwa bei den Knotenbasen und den starken Zusammenhangskomponenten der Fall, da hier jeweils die relationale Spezifikation dem Generieren aller Teilmengen mittels einer Potenzmengenrelation gefolgt von der Aussonderung der gewünschten Teilmengen entspricht. Hier ist es oft möglich, durch eine Kombination des relationalen Kalküls mit Programmentwicklungstechniken einen effizienten RelView-Prototyp in Gestalt eines relationalen Programms zu gewinnen, also, nach dessen Übertragung, beispielsweise ein effizientes Programm in C oder Modula-2[2].

Nachfolgend demonstrieren wir anhand von einigen Beispielen die Entwicklung von relationalen Programmen aus Spezifikationen; weitere Beispiele findet man etwa in [14, 32, 7, 15, 16, 17]. Um die Vorgehensweise nicht sofort durch einen übermäßigen Gebrauch von technischen relationenalgebraischen Rechnungen zu überdecken, beginnen wir mit zwei einfachen Beispielen, welche nur fundamentale Gesetze des relationalen Kalküls verwenden. Das erste davon greift das Problem aus dem Beispiel 5.3 für eine spezielle Klasse von gerichteten Graphen noch einmal auf.

6.1 Berechnung der Knotenbasis eines kreisfreien Graphen. Wiederum sei ein gerichteter Graph $g = (X, R)$ vorausgesetzt. Wir betrachten den Vektor $\overline{R^\mathsf{T}\,\mathsf{L}} : X \leftrightarrow \mathbf{1}$, der die Menge aller Knoten ohne Vorgänger darstellt. Dieser Vektor ist in allen eine Knotenbasis darstellenden Vektoren $v : X \leftrightarrow \mathbf{1}$ enthalten. Ein relationaler Beweis ist nicht schwierig: Daß von v aus alle Knoten erreichbar sind, heißt relational $(R^\mathsf{T})^*\, v = \mathsf{L}$. Daraus folgt die Inklusion

$$\overline{R^\mathsf{T}\,\mathsf{L}} \subseteq \mathsf{L} = (R^\mathsf{T})^*\, v = (\mathsf{I} \cup (R^\mathsf{T})^+)\, v = v \cup (R^\mathsf{T})^+\, v\,,$$

also $\overline{R^\mathsf{T}\,\mathsf{L}} \cap \overline{(R^\mathsf{T})^+\, v} \subseteq v$. Nun zeigt $(R^\mathsf{T})^+\, v = R^\mathsf{T}\,(R^\mathsf{T})^*\, v = R^\mathsf{T}\,\mathsf{L}$ die gewünschte Inklusion $\overline{R^\mathsf{T}\,\mathsf{L}} \subseteq v$. Wenn also $(R^\mathsf{T})^*\ \overline{R^\mathsf{T}\,\mathsf{L}} = \mathsf{L}$ gilt, so ist die Menge der Knoten ohne Vorgänger die einzige Knotenbasis von g und wird somit durch die RelView-Funktion

```
base1(R) = -(R^*Ln1(R))
```

berechnet. Die Vorbedingung $(R^\mathsf{T})^*\ \overline{R^\mathsf{T}\,\mathsf{L}} = \mathsf{L}$ zur Korrektheit von `base1` besagt, daß von jedem Knoten aus durch endliches „Rückwärtsschreiten“ ein vorgängerloser Knoten erreicht werden kann. Sie trifft damit insbesondere für endliche kreisfreie Graphen zu. □

Hat man ein effizientes Verfahren, um die Knotenbasis eines kreisfreien Graphen zu bestimmen, so ist das Problem im Prinzip auch für alle Graphen effizient gelöst. Wir werden darauf in Abschnitt 7 zurückkommen.

Als zweites Beispiel greifen wir das RelView-Programm `reach` von Abschnitt 3 noch einmal auf, das zu einem gerichteten Graphen $g = (X, R)$ und einem Vektor $s : X \leftrightarrow \mathbf{1}$ den Vektor $(R^\mathsf{T})^*\, s : X \leftrightarrow \mathbf{1}$ der von s aus erreichbaren Knoten berechnet. Im Vergleich zu einer

[2]Es sollte an dieser Stelle aber auch bemerkt werden, daß relationale Problemspezifikationen nicht immer ausführbar sein müssen, etwa wenn sie, wie in [19] anhand von einigen Beispielen gezeigt wird, Quantifizierungen über Relationen, Fixpunktbildungen oder ähnliche Konstrukte enthalten. Hier bietet sich an, solche Spezifikationen als Programm-Nachbedingungen zu verwenden und direkt mit der Entwicklung eines relationalen Programms zu beginnen. Aus diesem ergibt sich dann in unmittelbarer Weise ein RelView-Prototyp.

sich direkt aus dem Erreichbarkeitsvektor ergebenden relationalen Funktion, wird die Berechnungskomplexität von `reach` nicht durch die Kosten zur Berechnung der reflexiv-transitiven Hülle dominiert. Wie man leicht erkennt, ist durch eine geeignete Implementierung von Relationen in einer gängigen imperativen Programmiersprache dadurch sogar eine $O(|X|^2)$-Version des RelView-Prototyps `reach` möglich.

6.2 Berechnung der erreichbaren Knoten. Bei der nachfolgenden Entwicklung des relationalen Programms `reach` kombinieren wir die bekannte Invariantentechnik für imperative Programme (man vergleiche beispielsweise mit [24, 28]) mit dem relationalen Kalkül. Zur Spezifikation des Problems setzen wir zwei Eingabevariablen R und s und eine Resultatvariable u voraus und betrachten die Nachbedingung

$$u = (R^\mathsf{T})^* \, s \,. \tag{28}$$

Eine Vorbedingung ist nicht notwendig, da wir von R und s nichts fordern, außer der richtigen Typisierung und der zur Terminierung notwendigen Endlichkeit der Menge, auf der sie definiert sind.

Wir wollen, beginnend mit $u = s$, iterativ u solange um seine Nachfolger $R^\mathsf{T} \, u$ vergrößern, bis auf diese Weise keine neuen Knoten mehr erreicht werden. Eine Formalisierung dieser algorithmischen Idee verwendet

$$s \subseteq u \subseteq (R^\mathsf{T})^* \, s \;\wedge\; R^\mathsf{T} \, u \subseteq u \tag{29}$$

als Verstärkung der Nachbedingung (28), den ersten Teil von (29) als Schleifeninvariante und den zweiten Teil von (29) als Negation der Schleifenbedingung. Relational ist einfach zu zeigen, daß (28) von (29) impliziert wird, denn aus $R^\mathsf{T} \, u \subseteq u$ folgt $(R^\mathsf{T})^* \, u \subseteq u$ durch Induktion, also $(R^\mathsf{T})^* \, s \subseteq u$ wegen $s \subseteq u$. Wird u mit s initialisiert, so gilt die Schleifeninvariante trivialerweise. Nun erfülle u die Schleifeninvariante $s \subseteq u \subseteq (R^\mathsf{T})^* \, s$. Dann gilt

$$s \subseteq u \cup R^\mathsf{T} \, u \subseteq (R^\mathsf{T})^* \, s \cup R^\mathsf{T} \, (R^\mathsf{T})^* \, s = ((R^\mathsf{T})^* \cup R^\mathsf{T} \, (R^\mathsf{T})^*) \, s = (R^\mathsf{T})^* \, s$$

wegen $R^\mathsf{T} \, (R^\mathsf{T})^* = (R^\mathsf{T})^+ \subseteq (R^\mathsf{T})^*$. Somit erfüllt auch $u \cup R^\mathsf{T} \, u$ die Schleifeninvariante, d.h. die Zuweisung dieses Terms an u erhält als Schleifenrumpf ihre Gültigkeit. Insgesamt haben wir also gezeigt, daß, nach einer Umformung der Schleifenbedingung $R^\mathsf{T} \, u \not\subseteq u$ in $\overline{u} \cap R^\mathsf{T} \, u \neq \mathsf{O}$ und dem Übergang zu RelView, das nachfolgende Programm den Erreichbarkeitsvektor bestimmt:

```
reach1(R,s)
  DECL u
  BEG  u = s;
       WHILE -empty(-u & R^*u) DO
         u = u | R^*u OD
       RETURN u
  END.
```

Von `reach1` kommen wir nun zum Programm `reach` von Abschnitt 3, indem wir eine zweite Variable v zur inkrementellen Berechnung von $\overline{u} \cap R^\mathsf{T} \, u$ einführen. (In der Literatur wird dies auch Fortschreibungstechnik oder formales bzw. endliches Differenzieren genannt; siehe [39, 42].) Im Hinblick auf die Schleifeninvariante geschieht dies dadurch, daß wir (29) zu

$$v = \overline{u} \cap R^\mathsf{T} \, u \;\wedge\; s \subseteq u \subseteq (R^\mathsf{T})^* \, s \;\wedge\; R^\mathsf{T} \, u \subseteq u \tag{30}$$

ergänzen. Die Initialisierung und auch die Fortschreibung von v sind damit offensichtlich und entsprechen genau den beiden Zuweisungen in `reach`. Wegen (30) ist weiterhin $\overline{u} \cap R^\mathsf{T} \, u \neq \mathsf{O}$

äquivalent zu $v \neq \mathsf{O}$, womit wir die Schleifenbedingung von `reach1` zu der von `reach` vereinfachen dürfen. Eine weitere Konsequenz von (30) ist die Gleichheit

$$u \cup R^{\mathsf{T}}\, u = (u \cup \overline{u}\,) \cap (u \cup R^{\mathsf{T}}\, u) = u \cup (\,\overline{u} \cap R^{\mathsf{T}}\, u) = u \cup v\,,$$

wodurch sich schließlich auch noch die Zuweisung an u in der Schleife von `reach1` zu der von `reach` vereinfacht. □

Im folgenden Beispiel 6.3 greifen wir den Prototyp von Beispiel 5.4 zur Bestimmung der starken Zusammenhangskomponenten noch einmal auf. Wir wollen ein effizientes RelView-Programm entwickeln, das zu einer Äquivalenzrelation $R : X \leftrightarrow X$ die Menge $\mathcal{K}$ aller Äquivalenzklassen in Form einer Relation $C : X \leftrightarrow \mathcal{K}$ spaltenweise berechnet. Haben wir dies erreicht, so können wir natürlich auch die starken Zusammenhangskomponenten eines gerichteten Graphen $g = (X, S)$ spaltenweise berechnen, da diese ja gerade die Äquivalenzklassen der Äquivalenzrelation $S^* \cap (S^*)^{\mathsf{T}}$ sind.

Zur Herleitung eines relationalen Programms zur Bestimmung der Äquivalenzklassen reichen die als bekannt angenommenen fundamentalen Gesetze des relationalen Kalküls nicht mehr aus. Wir benötigen zusätzlich noch die sogenannten Schröder-Äquivalenzen

$$Q\,R \subseteq S \iff Q^{\mathsf{T}}\,\overline{S} \subseteq \overline{R} \iff \overline{S}\,R^{\mathsf{T}} \subseteq \overline{Q}\,. \tag{31}$$

Bei einer axiomatischen Behandlung von Relationen werden die Schröder-Äquivalenzen (31) in der Regel als eines der Axiome der Relationenalgebra definiert; man vergleiche etwa mit [41]. Weiterhin werden wir einige Eigenschaften des symmetrischen Quotienten anwenden, die wir nachfolgend zusammengestellt haben:

$$\begin{array}{rcl} p \text{ Punkt},\ p^{\mathsf{T}}\,p = \mathsf{I} & \Longrightarrow & syq(R\,p, R\,p) = \mathsf{I} \\ p \text{ Punkt} & \Longrightarrow & syq(R,S)\,p = syq(R, S\,p) \\ v \text{ Vektor} & \Longrightarrow & syq(R,v) \text{ Vektor} \\ R \text{ Äquivalenzrelation} & \Longrightarrow & syq(S,R)\,R = syq(S,R) \\ v \neq \mathsf{O} \text{ Vektor},\ R\,\mathsf{L} \subseteq \overline{v} & \Longrightarrow & syq(R,v) = \mathsf{O}\,. \end{array} \tag{32}$$

Alle diese Implikationen kann man recht einfach aus bekannten Eigenschaften des symmetrischen Quotienten ableiten, die man etwa in [41] findet. Schließlich benötigen wir noch einige Beziehungen der in (8) definierten Konkatenations-Funktion *conc*. Um die Lesbarkeit zu erleichtern, werden wir im folgenden, der Programmiersprache ML folgend, die Konkatenation mit dem Infix-Symbol @ bezeichnen. Mit dieser Schreibweise gilt dann die Implikation

$$syq(R,R) = \mathsf{I},\ syq(S,S) = \mathsf{I},\ syq(R,S) = \mathsf{O} \implies syq(R \mathbin{@} S, R \mathbin{@} S) = \mathsf{I} \tag{33}$$

und weiterhin haben wir die beiden Gleichungen

$$\begin{array}{rcl} syq(R, S \mathbin{@} T) & = & syq(R,S) \mathbin{@} syq(R,T) \\ (R \mathbin{@} S)\,\mathsf{L} & = & R\,\mathsf{L} \cup S\,\mathsf{L}\,. \end{array} \tag{34}$$

Die Beweise von (33) und (34) sind nicht viel schwieriger als die von (32), erfordern zusätzlich jedoch noch die Axiome der relationenalgebraischen Charakterisierung der Injektionen einer direkten Summe. Es sollte an dieser Stelle nachdrücklich betont werden, daß die Beziehungen (33) und (34) nicht vom Himmel fallen, sondern die relationalen Beschreibungen von recht anschaulichen Eigenschaften von Sequenzen darstellen. Wir wollen dies am Beispiel der Implikation (33) demonstrieren. Aus der Beziehung (15) erkennt man unmittelbar: Die Gleichungen

$syq(R,R) = \mathsf{I}$, $syq(S,S) = \mathsf{I}$ bzw. $syq(R \mathbin{@} S, R \mathbin{@} S) = \mathsf{I}$ beschreiben, daß alle Spalten von R, S bzw. $R \mathbin{@} S$ paarweise verschieden sind, und die Gleichung $syq(R,S) = \mathsf{O}$ drückt aus, daß R und S keine gemeinsame Spalte besitzen. Somit besagt (33) in Worten: Sind alle Spalten von R und S paarweise verschieden und besitzen beide Relationen keine gemeinsame Spalte, so sind auch alle Spalten ihrer Konkatenation paarweise verschieden.

Nach diesen relationenalgebraischen Vorbereitungen kommen wir nun zum schon angekündigten Beispiel.

6.3 Berechnung der Äquivalenzklassen. Es sei $R : X \leftrightarrow X$ eine Äquivalenzrelation auf einer endlichen Menge X, d.h. es gelte

$$\mathsf{I} \subseteq R \;\wedge\; R = R^{\mathsf{T}} \;\wedge\; R\,R \subseteq R\,. \tag{35}$$

Wir wollen mittels der Invariantentechnik ein relationales Programm herleiten, welches die Menge $\mathcal{K}$ der Äquivalenzklassen von R als Sequenz von Vektoren in Form einer Relation $C : X \leftrightarrow \mathcal{K}$ berechnet. Formal besitzt das gesuchte Programm also eine Eingabevariable R und eine Resultatvariable C und die Eigenschaft (35) entspricht genau seiner Vorbedingung.

Wir haben vor der Herleitung zuerst noch die Nachbedingung des gesuchten Programms relationenalgebraisch zu formulieren. Dazu erinnern wir uns an die relationale Spezifikation der starken Zusammenhangskomponenten in Beispiel 5.4. Mit der Potenzmengenrelation $\varepsilon : X \leftrightarrow 2^X$ erhalten wir als Verallgemeinerung davon den Vektor $syq(\varepsilon, R)\,\mathsf{L} : 2^X \leftrightarrow \mathbf{1}$ zur Darstellung der Äquivalenzklassen von R als Elemente der Potenzmenge 2^X. Eine erste aus dieser Darstellung sich ergebende Forderung an das Resultat C ist die Gleichheit $syq(\varepsilon, R)\,\mathsf{L} = syq(\varepsilon, C)\,\mathsf{L}$, denn diese beschreibt, daß die Spalten von C genau die Äquivalenzklassen von R darstellen. Damit haben wir jedoch noch nichts über die Vielfachheit der Spalten von C gesagt. Beabsichtigt ist natürlich, daß alle Spalten von C paarweise verschieden sind, d.h. im Resultat keine Äquivalenzklasse mehrfach auftaucht. Wie wir oben schon gesehen haben, kann man diese Eigenschaft relationenalgebraisch sehr einfach mit Hilfe des symmetrischen Quotienten ausdrücken. Fassen wir die bisherigen Überlegungen zusammen, so ergibt sich

$$syq(\varepsilon, R)\,\mathsf{L} = syq(\varepsilon, C)\,\mathsf{L} \;\wedge\; syq(C,C) = \mathsf{I} \tag{36}$$

als Nachbedingung des gesuchten relationalen Programms.

Der nächste Schritt ist nun die Verstärkung der Nachbedingung (36), um eine geeignete Schleifeninvariante und auch die Abbruchbedingung der Schleife zu bekommen. In dem vorliegenden Fall bietet sich an, eine zusätzliche Variable v des Typs $[X \leftrightarrow \mathbf{1}]$ einzuführen, in der die noch nicht abgearbeiteten Elemente von X festgehalten werden. Dies führt zu

$$syq(\varepsilon, R)\;\overline{v} = syq(\varepsilon, C)\,\mathsf{L} \;\wedge\; syq(C,C) = \mathsf{I} \;\wedge\; R\,v \cap C\,\mathsf{L} = \mathsf{O} \;\wedge\; v = \mathsf{O}\,. \tag{37}$$

Nach der Intention hinter v wird die Negation der letzten Gleichung von (37) zur Schleifenbedingung und der Rest zur Schleifeninvariante. In Worten besagt letztere, daß die Spalten von C genau die Äquivalenzklassen der schon abgearbeiteten Elemente sind (erste Gleichung), alle Spalten von C paarweise verschieden sind (zweite Gleichung) und kein Element einer noch nicht berechneten Äquivalenzklasse in einer der Spalten von C vorkommt (dritte Gleichung). Es ist offensichtlich, daß die Nachbedingung (36) von (37) impliziert wird. Wir haben also noch eine Initialisierung von v und C zu finden, welche die Schleifeninvariante etabliert, sowie einen Schleifenrumpf, der die Gültigkeit der Schleifeninvariante erhält.

Wir beginnen mit der Initialisierung. Hier bietet sich an, mit einer beliebigen Äquivalenzklasse zu starten, d.h. C durch $R\,point(\mathsf{L})$ mit $\mathsf{L} : X \leftrightarrow \mathbf{1}$ zu initialisieren. Eine Konsequenz davon ist, daß v mit $\overline{C}$ zu initialisieren ist. Mit dieser Initialisierung gilt die Schleifeninvariante. Für die

Beweise der entsprechenden drei Gleichungen kürzen wir $point(\mathsf{L})$ zu p ab. Die erste Gleichung zeigt man wie nachfolgend angegeben:

$$syq(\varepsilon, R)\ \overline{\overline{R\,p}} = syq(\varepsilon, R)\,p = syq(\varepsilon, R\,p) = syq(\varepsilon, R\,p)\,\mathsf{L}\,.$$

Dabei werden der Reihe nach die vierte Implikation von (32) in Verbindung mit der Vorbedingung (35), die zweite Implikation von (32) und die dritte Implikation von (32) verwendet. Die zweite Gleichung

$$syq(R\,p, R\,p) = \mathsf{I}$$

folgt unmittelbar aus der ersten Implikation von (32), denn p ist ein Punkt und wegen $p : X \leftrightarrow \mathbf{1}$ ist $p^{\mathsf{T}}\,p = \mathsf{I}$; man vergleiche mit der Bemerkung am Ende von Abschnitt 2. Zum Beweis der dritten Gleichung starten wir mit der offensichtlichen Äquivalenz

$$R\,\overline{R\,p} \cap R\,p\,\mathsf{L} = \mathsf{O} \iff R\,\overline{R\,p} \subseteq \overline{R\,p}$$

und haben dessen rechte Seite zu beweisen. Aus der Symmetrie und der Transitivität von R, also der Vorbedingung (35), erhalten wir $R^{\mathsf{T}}\,R \subseteq R$. Dies impliziert $R^{\mathsf{T}}\,R\,p \subseteq R\,p$. Nun zeigen die Schröder-Äquivalenzen (31) die Behauptung.

Als nächstes entwickeln wir einen Schleifenrumpf. Da das Resultat C die Äquivalenzklassen spaltenweise „aufzählen“ soll und v die Elemente von X beschreibt, deren Klassen noch nicht Spalten von C sind, ist eine sich direkt anbietende Vorgehensweise, mittels der Zuweisung von $C \mathbin{@} (R\,point(v))$ an C dem Resultat eine neue Klasse als Spalte anzufügen und v zu $v \cap \overline{R\,point(v)}$ abzuändern. In RelView sieht das vollständige Programm dann wie folgt aus:

```
classes(R)
  DECL C, v
  BEG  C = R*point(Ln1(R));
       v = -C;
       WHILE -empty(v) DO
         C = conc(C,R*point(v));
         v = v & -(R*point(v)) OD
       RETURN C
  END.
```

Zur Korrektheit von `classes` ist noch zu verifizieren, daß die beiden Zuweisungen des Schleifenrumpfs die Gültigkeit der Schleifeninvariante erhalten. Es gelte also die Schleifeninvariante für C und v. Für die Beweise der ersten drei Gleichungen von (37) für die neuen Werte von C und v sei p eine Abkürzung für $point(v)$. Hier ist der Beweis für die erste Gleichung:

$$\begin{array}{rcll}
syq(\varepsilon, R)\ \overline{v \cap \overline{R\,p}} & = & syq(\varepsilon, R)\ \overline{v} \cup syq(\varepsilon, R)\,R\,p & \\
& = & syq(\varepsilon, R)\ \overline{v} \cup syq(\varepsilon, R)\,p\,\mathsf{L} & (35),\ (32) \text{ vierte Impl.},\ p = p\,\mathsf{L} \\
& = & syq(\varepsilon, R)\ \overline{v} \cup syq(\varepsilon, R\,p)\,\mathsf{L} & (32) \text{ zweite Impl.} \\
& = & syq(\varepsilon, C)\,\mathsf{L} \cup syq(\varepsilon, R\,p)\,\mathsf{L} & \text{Schleifeninvariante} \\
& = & (syq(\varepsilon, C) \mathbin{@} syq(\varepsilon, R\,p))\,\mathsf{L} & (34) \text{ zweite Gleichung} \\
& = & syq(\varepsilon, C \mathbin{@} R\,p)\,\mathsf{L} & (34) \text{ erste Gleichung}.
\end{array}$$

Wir wollen die Implikation (33) zum Beweis der zweiten Gleichung

$$syq(C \mathbin{@} R\,p, C \mathbin{@} R\,p) = \mathsf{I}$$

anwenden, haben also deren drei Prämissen zu zeigen. Die Gleichung $syq(C, C) = \mathsf{I}$ gilt wegen der Schleifeninvariante. Der Beweis von $syq(R\,p, R\,p) = \mathsf{I}$ wird durch die erste Implikation von

(32) erbracht, denn p ist wiederum ein Punkt mit $p^\mathsf{T}\, p = \mathsf{I}$. Die dritte Prämisse folgt schließlich aus $R\, v \cap C\, \mathsf{L} = \mathsf{O}$, denn dieser Teil der Schleifeninvariante und $p \subseteq v$ implizieren $C\, \mathsf{L} \subseteq \overline{R\, v} \subseteq \overline{R\, p}$ und in Verbindung mit der Vektoreigenschaft von $R\, p$ und $\mathsf{O} \neq p \subseteq R\, p$ (hier gehen die Surjektivität von p und die Reflexivität von R ein) zeigt die fünfte Implikation von (32) das Gewünschte. Zum Beweis der dritten Gleichung

$$R\,(v \cap \overline{R\, v}\,) \cap (C \mathbin{@} R\, p)\, \mathsf{L} = \mathsf{O}$$

mit den neuen Werten der Variablen C und v des Programms `classes` starten wir mit der Inklusion

$$\begin{aligned}
R\,(v \cap \overline{R\, p}\,) \cap (C \mathbin{@} R\, p)\, \mathsf{L} &= R\,(v \cap \overline{R\, p}\,) \cap (C\, \mathsf{L} \cup R\, p\, \mathsf{L}) && \text{(34) zweite Gl.} \\
&\subseteq R\, v \cap R\, \overline{R\, p} \cap (C\, \mathsf{L} \cup R\, p) && p = p\, \mathsf{L} \\
&= R\, \overline{R\, p} \cap ((R\, v \cap C\, \mathsf{L}) \cup (R\, v \cap R\, p)) \\
&= R\, \overline{R\, p} \cap R\, p && \text{Invariante, } p \subseteq v\,.
\end{aligned}$$

Es bleibt somit noch $R\, \overline{R\, v} \cap R\, p = \mathsf{O}$ zu zeigen. Doch dies folgt, analog zur Etablierung der Schleifeninvariante, aus der Inklusion $R^\mathsf{T}\, R\, p \subseteq R\, p$ gefolgt von einer Anwendung der Schröder-Äquivalenzen. □

In den letzten beiden Beispielen 6.2 und 6.3 kombinierten wir die Invariantentechnik mit dem relationalen Kalkül zur Gewinnung von relationalen Programmen. Nachfolgend demonstrieren wir nun die Anwendung einer weiteren, aber ganz anders gearteten Programmentwicklungstechnik, nämlich der transformationellen Programmierung. Diese Technik wurde in den 80er Jahren insbesondere im Rahmen des Münchner CIP-Projekts untersucht; man vergleiche beispielsweise mit [3, 4]. Bei den entsprechenden relationenalgebraischen Rechnungen werden wir von der sogenannten Dedekind-Regel

$$Q\, R \cap S \subseteq (Q \cap S\, R^\mathsf{T})\, (R \cap Q^\mathsf{T}\, S) \tag{38}$$

Gebrauch machen. Diese wird manchmal an Stelle der Schröder-Äquivalenzen (31) bei einer axiomatischen Behandlung von Relationen verwendet. Weiterhin werden wir benutzen, daß die reflexiv-transitive Hülle R^* der kleinste Fixpunkt der monotonen Funktion $\tau_R(Q) = \mathsf{I} \cup R\, Q$ ist. Wegen des Tarski'schen Fixpunktsatzes (siehe [45]) folgt daraus unmittelbar, daß $\tau_R(S) \subseteq S$ die Inklusion $R^* \subseteq S$ impliziert.

6.4 Berechnung der reflexiv-transitiven Hülle. Wir wollen mittels transformationeller Programmierung ein relationales Programm entwickeln, das die relationale Funktion

$$rtc : [X \leftrightarrow X] \to [X \leftrightarrow X] \qquad\qquad rtc(R) = R^* \tag{39}$$

für endliche Mengen X implementiert, dabei aber keinen Bezug auf eine eventuell vorhandene Basisoperation zur Berechnung von R^* nimmt.

Beim Ansatz der transformationellen Programmierung ist es üblich, zuerst eine rekursive Version der ursprünglichen Spezifikation zu entwickeln, d.h. diese durch ein funktionales Programm zu implementieren. Der Terminierungsfall

$$rtc(\mathsf{O}) = \mathsf{I} \tag{40}$$

der relationalen Funktion rtc von (39) ergibt sich unmittelbar aus der Definition der reflexiv-transitiven Hülle. Im verbleibenden Fall $R \neq \mathsf{O}$ versuchen wir nun, die Berechnung von R^*

rekursiv auf die Berechnung der reflexiv-transitiven Hülle einer echten Teilrelation von R zurückzuführen. Grundlegend hierzu ist, daß die Implikation

$$S \mathbin{\mathsf{L}} S \subseteq S \implies (Q \cup S)^* = Q^* \cup Q^* \, S \, Q^* \tag{41}$$

für alle Relationen Q und S gilt. Die Inklusion "$\supseteq$" ihrer rechten Seite folgt aus $Q^* \subseteq (Q \cup S)^*$, $S \subseteq (Q \cup S)^*$ und der Transitivität von $(Q \cup S)^*$. Zum Beweis von "$\subseteq$" der rechten Seite von (41) verwenden wir $\mathsf{I} \subseteq Q^* \cup Q^* \, S \, Q^*$ und

$$\begin{array}{lll} (Q \cup S)\,(Q^* \cup Q^* \, S \, Q^*) & = & Q \, Q^* \cup Q \, Q^* \, S \, Q^* \cup S \, Q^* \cup S \, Q^* \, S \, Q^* \\ & \subseteq & Q^* \cup Q^* \, S \, Q^* \cup S \, Q^* \, S \, Q^* \qquad\qquad Q \, Q^* \subseteq Q^*,\ \mathsf{I} \subseteq Q^* \\ & \subseteq & Q^* \cup Q^* \, S \, Q^* \qquad\qquad S \mathbin{\mathsf{L}} S \subseteq S,\ \mathsf{I} \subseteq Q^* \,. \end{array}$$

Somit haben wir $\tau_{Q \cup S}(Q^* \cup Q^* \, S \, Q^*) \subseteq Q^* \cup Q^* \, S \, Q^*$, was in Kombination mit der unmittelbar vor diesem Beispiel gemachten Bemerkung die Behauptung zeigt.

Für $R \neq \mathsf{O}$ erfüllt $atom(R)$ die in (41) an S gestellte Vorbedingung. Aus der Beschreibung der relationalen Funktion $atom$ in Abschnitt 4 folgt nämlich, daß sowohl $atom(R) \mathbin{\mathsf{L}}$ als auch $atom(R)^\mathsf{T} \mathbin{\mathsf{L}}$ Punkte sind. Daraus erhalten wir $atom(R) \mathbin{\mathsf{L}} atom(R)^\mathsf{T} \subseteq \mathsf{I}$ und weiterhin

$$\begin{array}{lll} atom(R) \mathbin{\mathsf{L}} atom(R) & \subseteq & \mathsf{L} \, atom(R) \cap atom(R) \mathbin{\mathsf{L}} \\ & \subseteq & (\mathsf{L} \cap atom(R) \mathbin{\mathsf{L}} atom(R)^\mathsf{T})\,(atom(R) \cap \mathsf{L} \, atom(R) \mathbin{\mathsf{L}}) \qquad \text{Dedekind} \\ & \subseteq & atom(R) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{siehe oben.} \end{array}$$

Zerteilen wir also die nichtleere Relation R durch das Abtrennen eines einzelnen Atoms in $R \cap \overline{atom(R)}$ und $atom(R)$, so zieht dies

$$rtc(R) = rtc(R \cap \overline{atom(R)}\,) \cup rtc(R \cap \overline{atom(R)}\,) \, atom(R) \, rtc(R \cap \overline{atom(R)}\,) \tag{42}$$

nach sich. Nun fassen wir die beiden Gleichungen (40) und (42) mittels einer Fallunterscheidung zusammen und führen noch eine Hilfsfunktion $\varphi(Q, S) = Q \cup Q \, S \, Q$ zur Vermeidung von mehrfachen Aufrufen von rtc ein. Dies bringt

$$rtc(R) = \begin{cases} \varphi(rtc(R \cap \overline{atom(R)}\,), atom(R)) & : \quad R \neq \mathsf{O} \\ \mathsf{I} & : \quad R = \mathsf{O}\,. \end{cases} \tag{43}$$

Die durch (43) gegebene rekursive Beschreibung der originalen Spezifikation (39) stellt einen Algorithmus zur Berechnung der reflexiv-transitiven Hülle dar, der auch unmittelbar in die Sprache von RelView übertragen werden könnte, da diese Rekursion erlaubt. Unser Ziel ist jedoch ein noch effizienteres imperatives RelView-Programm. Dazu wenden wir auf die Rekursion (43) die Transformation der Funktionsumkehrung (siehe [3]) an und zwar in einer speziellen auf Mengen (also auch Relationen) zugeschneiderten Form, wie sie in [8] bewiesen wird. Diese Transformation bettet die nichtrepetitive Funktion rtc in eine repetitive Funktion F ein, welche, ausgehend vom Endwert I, durch die Umkehrfunktion zu $R \mapsto R \cap \overline{atom(R)}$ die „hängende" Operation φ auf Parameterposition berechnet. Wir erhalten

$$rtc(R) = F(R, \mathsf{O}, \mathsf{I}) \tag{44}$$

mit der dreistelligen repetitiv-rekursiven relationalen Funktion F, definiert durch

$$F(R, S, C) = \begin{cases} F(R, S \cup atom(R \cap \overline{S}\,), \varphi(C, atom(R \cap \overline{S}\,))) & : \quad S \neq R \\ C & : \quad S = R\,. \end{cases} \tag{45}$$

Damit ist die intellektuelle Arbeit der Programmentwicklung erledigt. Die durch (45) beschriebene Rekursion führt nämlich in Verbindung mit der Einbettung (44) in schematischer Weise unmittelbar zu einer imperativen Version mit einer Initialisierung und einer anschließenden while-Schleife. Eine entsprechende Transformation findet man wiederum in [3]. Expandiert man in dem aus (45) sich ergebenden imperativen relationalen Programm den Aufruf der relationalen Funktion φ, führt dann eine Variable zur Abspeicherung von $atom(R \cap \overline{S})$ ein, um Mehrfachberechnungen dieses Terms zu vermeiden, und geht anschließend noch zur Syntax von RelView über, so ist das Resultat das nachfolgend angegebene RelView-Programm zur Berechnung der reflexiv-transitiven Hülle:

```
rtc(R)
  DECL S, C, a
  BEG  S = 0(R);
       C = I(R);
       WHILE -eq(S,R) DO
         a = atom(R & -S);
         S = S | a;
         C = C | C*a*C OD
       RETURN C
  END.
```

Die Aufspaltung einer nichtleeren Relation R durch das Abtrennen eines einzelnen Atoms $atom(R)$ ist nicht die einzige Möglichkeit, die Implikation (41) zur Gewinnung einer Rekursion für die ursprüngliche Spezifikation (39) anzuwenden. Wegen der Eigenschaft

$$v\,v^\mathsf{T}\,R\,\mathsf{L}\,v\,v^\mathsf{T}\,R \subseteq v\,\mathsf{L}\,v^\mathsf{T}\,R = v\,v^\mathsf{T}\,R$$

für Vektoren v kann man beispielsweise durch die Auswahl eines im Definitionsbereich $R\,\mathsf{L}$ von R enthaltenen Punkts p auch eine nichtleere Zeile $p\,p^\mathsf{T}\,R$ von R abtrennen[3]. Diese Vorgehensweise führt auf die nachfolgende Variante des bekannten Algorithmus von Warshall (siehe [48]); wir verzichten an dieser Stelle auf eine formale Herleitung, da diese im wesentlichen aus den gleichen Schritten bestehen würde, wie die eben erbrachte:

```
rtc1(R)
  DECL s, C, p
  BEG  s = On1(R);
       C = I(R);
       WHILE -eq(s,dom(R)) DO
         p = point(dom(R) & -s);
         s = s | p;
         C = C | (C*p)*((p^*R)*C) OD
       RETURN C
  END.
```

Die Klammerung in der dritten Zuweisung des Schleifenrumpfs des RelView-Programms `rtc1` ist wesentlich, um bei einer Matriximplementierung von Relationen, wie sie bei der derzeitigen Systemversion verwendet wird, eine quadratische Laufzeit für den Schleifenrumpf zu erzielen. Insgesamt führt dies für eine Relation $R : X \leftrightarrow X$ zu einem Aufwand $O(|X|^3)$ für das Programm

[3] Analog ist es natürlich mittels eines Punkts p aus dem Wertebereich $R^\mathsf{T}\,\mathsf{L}$ von R möglich, eine einzelne Spalte $R\,p\,p^\mathsf{T}$ von der nichtleeren Relation R abzutrennen.

`rtc1`. Im Vergleich dazu hat das obige Programm `rtc` für eine Eingaberelation $R : X \leftrightarrow X$ eine Laufzeit von $O(|R| * |X|^3)$. Eine Laufzeit von $O(|R| * |X|^2)$ kann jedoch erreicht werden, wenn man, unter Verwendung der gewohnten mathematischen Schreibweise, den Term $C\,a\,C$ in der dritten Zuweisung des Schleifenrumpfs durch $(C\,(a\,\mathtt{L}))\,((\mathtt{L}^{\mathsf{T}}\,a)\,C)$ mit $\mathtt{L} : X \leftrightarrow \mathbf{1}$ ersetzt, da, wegen der Klammerung, der letztgenannte Term nur Multiplikationen von Relationen mit Spalten- und Zeilenvektoren beschreibt, was in der Matriximplementierung in quadratischer Zeit möglich ist.

In [16] wird mittels der Invariantentechnik das nachfolgende relationale Programm zur Berechnung der reflexiv-transitiven Hülle hergeleitet; sein Rumpf entspricht dabei genau dem Warshall-Verfahren zur Berechnung der transitiven Hülle:

```
rtc2(R)
  DECL s, C, p
  BEG  C = R;
       s = Ln1(R);
       WHILE -eq(s,O(s)) DO
         p = point(s);
         s = s & -p;
         C = C | (C*p)*(p^*C) OD
       RETURN C | I(R)
  END.
```

Ein Nachteil des obigen Programms `rtc1` im Vergleich zu `rtc2` ist dadurch gegeben, daß bei jedem Schleifendurchlauf zusätzlich ein Zeilenvektor mit einer Relation multipliziert wird. Dieser Nachteil wird jedoch bei Relationen mit vielen leeren Zeilen (die gerichteten Graphen mit vielen Knoten ohne Nachfolger, z.B. gerichteten Bäumen, entsprechen) in der Regel mehr als aufgehoben durch die Tatsache, daß für solche Eingaben das Programm `rtc1` weit weniger Schleifendurchläufe benötigt als `rtc2`. □

Wir haben am Ende von Abschnitt 5 die Bedeutung von RelView bei der Erstellung einer relationalen Spezifikation erwähnt und wollen nun diesen Abschnitt mit einigen Bemerkungen zum Prototyping der entwickelten relationalen Programme mittels des Systems und zur Verwendung von RelView bei relationenalgebraischen Beweisen beschließen.

Obwohl die hergeleiteten relationalen Programme theoretisch per Konstruktion korrekt bezüglich der Spezifikation sind, ist auch hier ein stichprobenhaftes Testen durchaus noch sinnvoll. So liefert es zusätzliche Sicherheit. Selbst bei relationenalgebraischen Rechnungen sind, sofern sie in der üblichen Weise mit Papier und Bleistift durchgeführt werden, in der Praxis trotz der äußerst formalen Vorgehensweise Rechenfehler nie auszuschließen. Deren Auswirkungen werden dann eventuell beim Testen in Form von nicht erwarteten Resultaten entdeckt. Weiterhin werden durch das Prototyping und das Durchspielen von Programmabläufen manchmal Optimierungsmöglichkeiten erkennbar. Dies kann etwa eine effizientere relationale Modellierung der Daten betreffen. Man kann aber auch eine neue Invariante entdecken, die eine Laufzeitoptimierung erlaubt, beispielsweise durch eine geeignete Fortschreibung wie im Fall der Erreichbarkeitsprogramme `reach1` und `reach`. Zwar ist der Gewinn beim Übergang von `reach1` zu `reach` innerhalb von RelView noch nicht sehr bedeutend, nach einer Übertragung der beiden Programme in eine gängige imperative Programmiersprache mit einer geeigneten Implementierung von Relationen ist der durch die Fortschreibung erzielte Effizienzgewinn jedoch beträchtlich. Aufgrund der Flexibilität und der vielfältigen Visualisierungsmöglichkeiten des RelView-Systems ist es im Allgemeinen sehr einfach, Alternativen durchzutesten und auf ihre Konsequenzen hin zu untersuchen.

RelView ist ein System zur Manipulation von Relationen und kein Beweisassistent. Trotzdem hat sich RelView auch bei der Durchführung von relationenalgebraischen Beweisen, wie sie in dem vorliegenden Anwendungsbereich insbesondere bei der Herleitung von relationalen Programmen aus Spezifikationen auftreten, als große Hilfe erwiesen. Das System erlaubt nämlich – man vergleiche mit den Bemerkungen am Ende von Abschnitt 3 – die Gültigkeit von relationenalgebraischen Formeln zu testen. Dabei können die Testdaten entweder interaktiv durch einen Kommandoknopf oder im Rahmen eines RelView-Programms durch Basisfunktionen zufällig mit einer vorgegebenen Wahrscheinlichkeit erzeugt werden. Besonders die zweite Vorgehensweise ermöglicht es, innerhalb kurzer Zeit in einfacher Weise viele Tests durchzuführen. Diese können beispielsweise dazu dienen, Gegenbeispiele für Behauptungen zu suchen, von deren Gültigkeit man (noch) nicht überzeugt ist. Man kann sie auch verwenden, um sich vor dem Beweis einer nicht offensichtlichen aber doch als zutreffend eingeschätzten relationenalgebraischen Aussage von deren Gültigkeit noch mehr zu überzeugen. Beispiele für solche Aussagen sind die Implikation (33) und die Gleichungen (34). Tests zeigen schließlich auch oft erst Eigenschaften auf, die sich für die weitere Programmentwicklung dann als sehr nützlich erweisen.

7 Einige weitere relationale Programme

In den letzten beiden Abschnitten haben wir anhand von zahlreichen Beispielen aufgezeigt, wie man formal von Spezifikationen zu relationalen Programmen kommen kann und welche Vorteile sich bei diesem Prozeß aus der Benutzung von RelView ergeben können. Nachfolgend geben wir nun noch einige weitere RelView-Programme an, verzichten aber auf die formalen Herleitungen. Ziel dieser Beispiele ist es, die großen Möglichkeiten zu demonstrieren, die RelView auch bietet, wenn man es als Implementierungssystem für Algorithmen verwendet, die in der gängigen semiformalen Art angegeben werden. In Verbindung mit den vielseitigen Visualisierungsmöglichkeiten des Systems scheint diese Anwendung insbesondere für Ausbildungszwecke sehr geeignet zu sein.

7.1 Knotenbasen von beliebigen Graphen. In Beispiel 6.1 haben wir gezeigt, daß ein kreisfreier gerichteter Graph genau die Menge aller Knoten ohne Vorgänger als einzige Knotenbasis besitzt. Damit ist das Problem der Bestimmung einer Knotenbasis im Prinzip aber auch für alle Graphen $g = (X, R)$ gelöst. Durch Betrachtung des kreisfreien reduzierten Graphen $g_{red} = (\mathcal{C}, \Phi^\mathsf{T} R \Phi \cap \overline{\mathsf{I}})$ von $g = (X, R)$, wobei $\Phi : X \leftrightarrow \mathcal{C}$ der kanonische Epimorphismus von X in die Menge $\mathcal{C}$ der starken Zusammenhangskomponenten ist, kann man nämlich recht einfach den folgenden Sachverhalt zeigen: Sind C_i, $1 \leq i \leq n$, die initialen starken Zusammenhangskomponenten von g, d.h. diejenigen Mengen aus $\mathcal{C}$, in die keine Pfeile führen, und $x_i \in C_i$ für $1 \leq i \leq n$, so ist $\{x_1, \ldots, x_n\}$ eine Knotenbasis von g.

Um mittels RelView eine Knotenbasis von $g = (X, R)$ in einer Vektordarstellung zu erhalten, bietet sich beispielsweise die folgende Vorgehensweise an: Wir bestimmen zuerst den kanonischen Epimorphismus Φ. Dies ist möglich durch einen Aufruf des relationalen Programms `classes` von Beispiel 6.3 mit dem Argument $R^* \cap (R^*)^\mathsf{T}$, denn die Matrixdarstellung der Relation Φ ist, wie man sich leicht klar macht, genau die spaltenweise Darstellung der Menge der starken Zusammenhangskomponenten. Im zweiten Schritt entfernen wir dann aus Φ alle Spalten, die nicht initiale starke Zusammenhangskomponenten darstellen. Dazu betrachten wir zuerst den Vektor

$$v = \overline{(\Phi^\mathsf{T} R^\mathsf{T} \Phi \cap \overline{\mathsf{I}})\,\mathsf{L}} \tag{46}$$

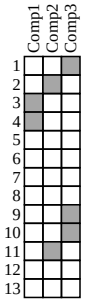
zur Darstellung der vorgängerlosen Knoten des reduzierten Graphen g_{red}. Da g_{red} kreisfrei ist, ist der mittels (46) definierte Vektor $v : \mathcal{C} \leftrightarrow \mathbf{1}$ nicht leer und somit die durch v induzierte injektive

Einbettung $inj(v) : \mathcal{C}_{init} \leftrightarrow \mathcal{C}$ der Menge $\mathcal{C}_{init}$ der initialen starken Zusammenhangskomponenten in die Menge $\mathcal{C}$ aller starken Zusammenhangskomponenten definiert. Die gesuchte spaltenweise Darstellung $C : X \leftrightarrow \mathcal{C}_{init}$ der initialen starken Zusammenhangskomponenten ergibt sich nun in einfacher Weise mittels der Gleichung

$$C = \Phi \, inj(v)^{\mathsf{T}} . \tag{47}$$

Schließlich wählen wir in einem dritten Schritt noch aus jeder Spalte der spaltenweisen Darstellung (47) der initialen starken Zusammenhangskomponenten genau einen Punkt aus und vereinigen alle diese Punkte zu einem eine Knotenbasis von g darstellenden Vektor.

Als ein konkretes Beispiel betrachten wir nun den gerichteten Graphen aus 5.6. Durch Anwendung der beiden Gleichungen (46) und (47) auf seine Relation erhalten wir für die Relation C die nachfolgend angegebene RelView-Darstellung:



Wählen wir in jeder Spalte dieser Booleschen Matrix den obersten Eintrag aus, so führt dies zur Knotenbasis $\{3, 2, 1\}$; insgesamt besitzt der gerichtete Graph 12 verschiedene Knotenbasen.

Ein RelView -Programm zur Realisierung der oben aufgeführten drei Schritte zur Berechnung einer Knotenbasis ist nachfolgend angegeben:

```
base(R)
  DECL Phi, v, C, b, p
  BEG  Phi = classes(rtc(R) & rtc(R)^);
       v = -dom(ip(Phi^ * R^ * Phi)));
       C = Phi * inj(v)^;
       p = init(On1(C^));
       b = point(C * p);
       WHILE -empty(next(p)) DO
         p = next(p);
         b = b | point(C * p) OD
       RETURN b
  END.
```

In diesem Programm ist `ip` die schon in Beispiel 5.5 bei der Spezifikation der Trennknoten verwendete relationale Funktion zur Berechnung des irreflexiven Anteils einer Relation. Die Variable p durchläuft alle Punkte des Typs $[\mathcal{C}_{init} \leftrightarrow \mathbf{1}]$, wobei mittels der Konstruktion $C\,p$ jeweils eine Spalte von C ausgewählt wird. Wegen $p = init$ wird anfangs die erste Spalte von C ausgewählt, dann, da nun $p = next(init)$, die zweite Spalte usw. □

In (2) haben wir eine relationale Funktion *tik* zur Berechnung des einzigen transitiv-irreduziblen Kerns einer kreisfreien Relation angegeben. Diese Funktion war unmittelbar in eine RelView-Funktion `tik` übertragbar. Wir behandeln nun die Bestimmung eines transitiv-irreduziblen Kerns einer beliebigen Relation mit Hilfe des RelView-Systems.

7.2 Transitiv-irreduzible Kerne von beliebigen Graphen. In [37] wird von Noltemeier ein Verfahren zur Berechnung eines transitiv-irreduziblen Kerns eines beliebigen gerichteten Graphen vorgeschlagen, das aus drei Schritten besteht und das wir nachfolgend in der Sprache von RelView beschreiben wollen.

Zuerst behandelt Noltemeier das Problem für den Spezialfall eines stark zusammenhängenden gerichteten Graphen $g = (X, R)$, d.h. für $R^* = \mathsf{L}$. Ist die Knotenmenge einelementig, so stimmen R und der transitiv-irreduzible Kern von g offensichtlich überein. Nun habe der Graph mindestens zwei Knoten. Übertragen in die Sprache der Relationen schlägt Noltemeier dann die folgende Vorgehensweise vor: Es sei $R = r_1 \cup \ldots \cup r_m$ die Vereinigung von paarweise verschiedenen Atomen r_i, $1 \leq i \leq m$ (wobei jedes Atom genau einem Pfeil entspricht). Man startet die Berechnung mit der Relation $R_1 = R \cap \overline{r_1}$ und prüft für alle i, $1 \leq i \leq m$, ob die Gleichung $R_i^* = \mathsf{L}$ gilt. Wenn ja, dann setzt man $R_{i+1} = R_i \cap \overline{r_{i+1}}$. Andernfalls definiert man $R_{i+1} = (R_i \cup r_i) \cap \overline{r_{i+1}}$. Dabei wird für den letzten Schritt $r_{m+1} = \mathsf{O}$ angenommen.

Wenn wir das eben beschriebene Verfahren in die Sprache von RelView übertragen, dann erhalten wir das folgende relationale Programm:

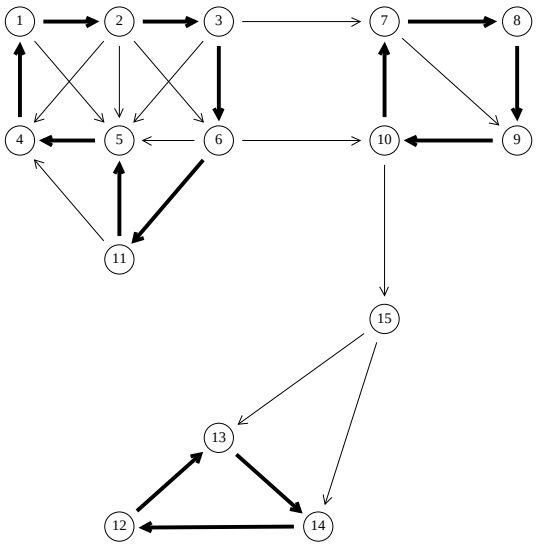
```
tiksc(R)
  DECL S, K, a, b
  BEG  IF eq(I(R),L(R)) THEN K = R
                        ELSE a = atom(R);
                             S = R & -a;
                             K = S;
                             WHILE -empty(S) DO
                               b = atom(S);
                               IF eq(rtc(K),L(K)) THEN K = K & -b
                                                  ELSE K = (K | a) & -b FI
                               S = S & -b;
                               a = b OD;
                             IF -eq(rtc(K),L(K)) THEN K = K | b FI FI
       RETURN K
  END.
```

Nun sei $g = (X, R)$ ein beliebiger gerichteter Graph. Der zweite Schritt von Noltemeiers Verfahren besteht, wiederum ausgedrückt unter Verwendung relationaler Sprechweisen, in der Bestimmung eines transitiv-irreduziblen Kerns für jede starke Zusammenhangskomponente von g als eine Relation des Typs $[X \leftrightarrow X]$ und der Vereinigung aller dieser Relationen. Ein entsprechendes RelView-Programm sieht wie nachfolgend angegeben aus:

```
tiksccs(R)
  DECL Phi, Inj, K, p
  BEG  Phi = classes(rtc(R) & rtc(R)^);
       p = init(On1(Phi^));
       Inj = inj(Phi*p);
       K = Inj^*tiksc(Inj*R*Inj^)*Inj;
       WHILE -empty(next(p)) DO
         p = next(p);
         Inj = inj(Phi*p);
         K = K | Inj^*tiksc(Inj*R*Inj^)*Inj OD
       RETURN K
  END.
```

Dieses relationale Programm berechnet zuerst die spaltenweise Darstellung der starken Zusammenhangskomponenten, d.h. den kanonischen Epimorphismus $\Phi : X \leftrightarrow \mathcal{C}$. Dann werden alle Spalten der Relation Φ durchlaufen, wobei dies, analog zu Beispiel 7.1 der Knotenbasen, durch das Durchlaufen der Punkte p von $[\mathcal{C} \leftrightarrow \mathbf{1}]$ und eine jeweilige Produktbildung $\Phi\, p$ realisiert wird. Für jede Spalte $v = \Phi\, p$, die eine starke Zusammenhangskomponente C darstellt, wird ein transitiv-irreduzibler Kern des durch C induzierten gerichteten Graphen $g_C = (C, \mathit{inj}\,(v)\, R\, \mathit{inj}\,(v)^\mathsf{T})$ als Relation des Typs $[C \leftrightarrow C]$ bestimmt und, durch Multiplikation mit $\mathit{inj}\,(v)^\mathsf{T}$ von links und $\mathit{inj}\,(v)$ von rechts, zu einer Relation des Typs $[X \leftrightarrow X]$ erweitert. In der Matrixinterpretation von Relationen entspricht letzteres einer entsprechenden Erweiterung der Dimension und einer Auffüllung mittels Nullen. Die Vereinigung aller dieser Erweiterungen geschieht ebenfalls im Rahmen der while-Schleife.

Zur Verdeutlichung ist nachfolgend das RelView-Bild eines gerichteten Graphen angegeben, wobei der durch das Programm `tiksccs` berechnete Anteil eines transitiv-irreduziblen Kerns vom System mittels fetter Pfeile markiert wurde:

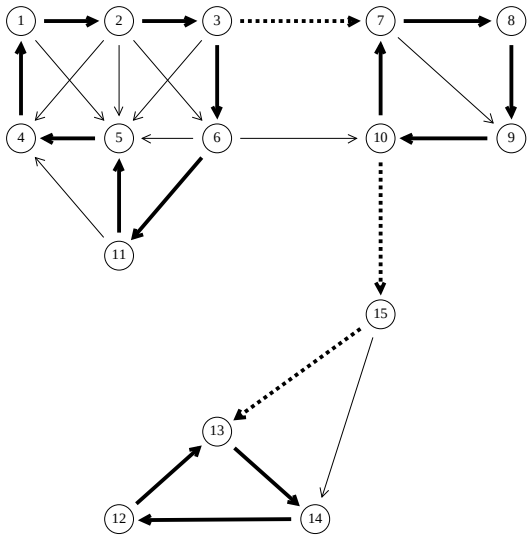


Durch den zweiten Schritt des Verfahrens von [37] wird derjenige Teil eines transitiv-irreduziblen Kerns von $g = (X, R)$ bestimmt, der genau aus den Pfeilen von R besteht, die jeweils innerhalb derselben starken Zusammenhangskomponente von g liegen. Der dritte Schritt beinhaltet schließlich noch die Bestimmung des Rests dieses transitiv-irreduziblen Kerns und die Vereinigung mit dem schon berechneten Teil. Zur Berechnung des Rests betrachtet Noltemeier den kreisfreien reduzierten Graphen $g_{red} = (\mathcal{C}, \Phi^\mathsf{T}\, R\, \Phi \cap \overline{\mathsf{I}})$ von g und bestimmt zuerst dessen eindeutig bestimmten transitiv-irreduziblen Kern. Jeder Pfeil $\langle C_i, C_j \rangle$ dieser Relation repräsentiert die Relation $R_{ij} : X \leftrightarrow X$ derjenigen Pfeile $\langle x, y \rangle$ von R, für die $x \in C_i$ und $y \in C_j$ gelten. Der gesuchte Rest des transitiv-irreduziblen Kerns besteht nun gerade aus je einem Pfeil aus jeder der Relationen R_{ij}.

In dem nachfolgenden Programm `tikrest` ist die Relation des reduzierten Graphen mit *Red* bezeichnet. Ein transitiv-irreduzibler Kern von g_{red}, in `tikrest` mit *Kr* bezeichnet, ergibt sich somit durch den Aufruf der früheren relationalen Funktion `tik` mit dieser Relation als Argument. Durch die while-Schleife von `tikrest` werden schließlich alle Pfeile $a = \langle C_i, C_j \rangle$ von *Kr* durchlaufen und dabei wird jeweils ein Pfeil von R_{ij} zur anfangs leeren Resultatrelation K hinzugefügt. Die Bestimmung von R_{ij} selbst ist sehr einfach möglich mit Hilfe des kanonischen Epimorphismus Φ. Hier ist Noltemeiers Verfahren zur Bestimmung des Rests eines transitiv-irreduziblen Kerns in der RelView-Implementierung:

```
tikrest(R)
  DECL Phi, Red, Kr, K, a
  BEG  Phi = classes(rtc(R) & rtc(R)^);
       Red = ipa(Phi^*R*Phi);
       Kr = tik(Red);
       K = 0(R);
       WHILE -empty(Kr) DO
         a = atom(Kr);
         K = K | atom(Phi*a*Phi^ & R);
         Kr = Kr & -a OD
       RETURN K
  END.
```

Wie schon erwähnt, ergibt sich ein transitiv-irreduzibler Kern eines beliebigen gerichteten Graphen durch die Vereinigung der Resultate der Aufrufe von `tiksccs` und `tikrest`. Das nächste RelView-Bild zeigt noch einmal den obigen gerichteten Graphen, wobei nun aber auch noch der restliche Teil des transitiv-irreduziblen Kerns markiert ist:



Um den von `tiksccs` berechneten Teil von dem von `tikrest` berechneten Teil unterscheiden zu können, haben wir das System angewiesen, die Pfeile des zweiten Teils mittels fetter punktierter Linien hervorzuheben. □

Im nächsten Beispiel behandeln wir ein Problem für ungerichtete Graphen. Gegensätzlich zum früheren Beispiel der Trennknoten müssen wir diesmal aber Bezug nehmen auf einzelne Kanten. Wir fassen einen ungerichteten Graphen als ein Paar $g = (X, R)$ mit einer symmetrischen und irreflexiven Relation $R : X \leftrightarrow X$ auf und eine Kante von g als eine Relation der Form $atom(R) \cup atom(R)^{\mathsf{T}}$[4]. Da Aufrufe der relationalen Funktion *atom* mit gleichen Argumenten gleiche Resultate liefern, besteht eine Kante somit aus zwei entgegengerichteten Pfeilen, was in RelView bildlich durch eine Linie mit je einer Pfeilspitze an einem Ende ausgedrückt wird und durch die Funktion

```
edge(R) = atom(R) | atom(R)^
```

[4]Eine Kante als zweielementige Knotenenge $\{x, y\}$ im üblichen graphentheoretischen Sinn wird also in dieser relationalen Auffassung zur zweielementigen Relation.

die Auswahl einer Kante aus einem nichtleeren ungerichteten Graphen erlaubt. Weiterhin werden wir im nächsten Beispiel den Begriff einer *Brücke* benötigen. Eine Brücke des ungerichteten Graphen $g = (X, R)$ ist eine Kante $e : X \leftrightarrow X$, durch deren Entfernung die Anzahl der Zusammenhangskomponenten echt erhöht wird, was wiederum relationenalgebraisch durch die Bedingung $e \not\subseteq (R \cap \overline{e})^+$ charakterisiert werden kann. Aus dieser Bedingung erhalten wir sofort eine einfache RelView-Funktion zum Testen der Brückeneigenschaft, die wir nachfolgend verwenden.

7.3 Eulersche Kreise. Von Euler stammt eine Charakterisierung der zusammenhängenden ungerichteten Graphen (heutzutage *Eulersch* genannt), die man ohne Absetzen des Bleistifts zeichnen kann, d.h. für die ein einfacher Kreis existiert, der alle Kanten genau einmal durchläuft. Genau diejenigen zusammenhängenden ungerichteten Graphen sind Eulersch, wo jeder Knoten einen geraden Grad besitzt.

Für das Problem, in einem Eulerschen Graphen einen einfachen Kreis zu konstruieren, der alle Kanten genau einmal durchläuft, sind die Algorithmen von Fleury und Hierholzer aus dem 19. Jahrhundert die bekanntesten Lösungsverfahren. In [2] findet man beispielsweise eine Beschreibung des Verfahrens von Fleury zum Eingabegraphen $g = (X, R)$, die wir nachfolgend mit geringfügigen Änderungen wiedergeben: Man startet mit einem beliebigen Knoten x_1. Nun sei der Anfang $c = \langle x_1, \ldots, x_i \rangle$ eines Eulerschen Kreises von g bereits konstruiert und g_i der Restgraph, der aus g durch Entfernen der Kanten $\{x_j, x_{j+1}\}$, $1 \leq j \leq i-1$, von c entsteht. Ist g_i leer, so ist das Verfahren beendet und c das Resultat. Ansonsten wählt man unter den mit x_i inzidierenden Kanten eine, die keine Brücke in g_i ist, solange dies möglich ist, und fügt den anderen Endknoten rechts an c an. Dies wird solange wiederholt, bis der Restgraph leer ist.

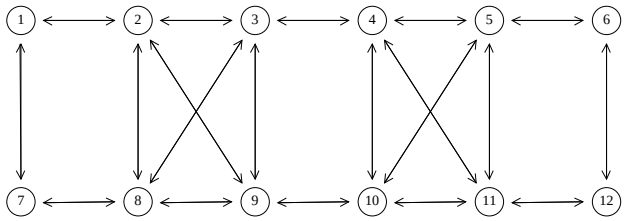
Das nachfolgend angegebene relationale Programm stellt eine RelView-Implementierung der eben beschriebenen Vorgehensweise dar:

```
eulercycle(R)
  DECL isbridge(R,e) = -incl(e,trans(R & -e));
       S, C, E, e, p
  BEG  p = point(Ln1(R));
       S = R;
       E = p;
       WHILE -empty(S) DO
         C = S & (p*L1n(S) | Ln1(S)*p^);
         IF empty(C & -edge(C)) THEN e = C
                                ELSE e = edge(C);
                                     C = C & -e;
                                     WHILE isbridge(S,e) DO
                                       e = edge(C);
                                       C = C & -e OD FI;
         p = e*p;
         S = S & -e;
         E = conc(E,p) OD
       RETURN E
  END.
```

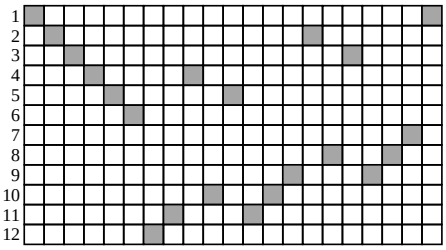
In diesem Programm wird die Variable E zur spaltenweisen Konstruktion des Eulerschen Kreises verwendet und die Variable S beinhaltet die Relation des oben erwähnten Restgraphen. In der ersten Zuweisung der äußeren while-Schleife wird die Menge der Kanten, die mit dem durch den

Punkt p dargestellten Knoten inzidieren, in C abgespeichert. Zu einer Kante e aus C ist dann $e\,p$ derjenige Punkt, der den verbleibenden Knoten von e darstellt.

Als eine konkrete Anwendung dieser RelView-Implementierung des Algorithmus von Fleury betrachten wir nun den in dem folgenden Bild dargestellten Eulerschen Graphen $g = (X, R)$.



Wenden wir das relationale Programm `eulercycle` auf die Relation R an, so erscheint auf dem Schirm des RelView-Systems die folgende 12×21 Boolesche Matrix:



Die Sequenz von 21 Punkten, welche durch diese Matrix spaltenweise modelliert wird, stellt genau den Eulerschen Kreis $\langle 1, 2, 3, 4, 5, 6, 12, 11, 4, 10, 5, 11, 10, 9, 2, 8, 3, 9, 8, 7, 1\rangle$ von g dar. □

Wir kommen nun zur relationalen Formulierung eines der fundamentalsten Prinzipien für Graphenalgorithmen, der Tiefensuche (englisch Depth-first search oder kurz DFS). Im nachfolgenden Beispiel wird diese zuerst in RelView formuliert. Dann werden einige „klassische“ Anwendungen von Tiefensuche angegeben. Dabei behandeln wir sowohl gerichtete als auch ungerichtete Graphen.

7.4 Tiefensuche mit Anwendungen. Bei der Tiefensuche durchläuft man, ausgehend von einem Startknoten, einen vorliegenden Graphen nach der folgenden Strategie: Suche solange in die Tiefe (Pfeil für Pfeil bzw. Kante für Kante), bis ein Knoten ohne unbesuchte Nachfolger bzw. Nachbarn erreicht ist. Gehe dann zurück bis zum ersten Knoten mit einem noch nicht besuchten Nachfolger bzw. Nachbarn und starte bei diesem wieder die Tiefensuche.

In der Literatur, beispielsweise in [43, 1, 26], wird die Methode der Tiefensuche üblicherweise als rekursive Prozedur formuliert. Dabei werden die Strukturinformationen, die man beim Durchlaufen gewinnt, insbesondere der durch das Durchlaufen erhaltene gerichtete Wald (genannt DFS-Wald), die Reihenfolge der Knoten, in der sie beim Durchlaufen besucht werden (genannt DFS-Numerierung), und die Klassifizierung der Pfeile bzw. Kanten, in geeigneten globalen Datenstrukturen (meistens Feldern) abgespeichert. Die Sprache von RelView erlaubt Rekursion. Sie kennt jedoch keine Prozeduren, sondern nur Funktionsprozeduren im Sinne von Pascal oder Modula-2, die Werte berechnen. Dies bedingt, daß nur jeweils eine der Strukturinformationen als entsprechendes Resultat abgeliefert werden kann. Von besonderer Wichtigkeit ist hier der DFS-Wald. Es hat sich nämlich gezeigt, daß bei einem relationalen Ansatz zur Tiefensuche aus diesem alle weiteren gängigen Strukturinformationen sehr einfach mittels relationaler Terme berechnet werden können.

Das nachfolgende rekursive RelView-Programm `dfs` durchläuft einen gerichteten bzw. ungerichteten Graphen $g = (X, R)$ ausgehend von dem durch den Punkt $p : V \leftrightarrow \mathbf{1}$ beschriebenen Knoten, wobei der Vektor $v : X \leftrightarrow \mathbf{1}$ die Menge der schon (durch Tiefensuchen ausgehend von anderen Knoten) besuchten Knoten darstellt. Sein Resultat ist, als ein Teil des späteren Gesamtergebnisses der Tiefensuche, die Relation eines gerichteten Baums mit dem durch p bezeichneten Knoten als Wurzel. Die Formulierung von `dfsvisit` ergibt sich unmittelbar aus der Übertragung der entsprechenden Prozedur von beispielsweise [43] in RelView-Notation:

```
dfsvisit(R,p,v)
  DECL T, u, w, q
  BEG  T = 0(R);
       u = v | p;
       w = R^*p & -u;
       WHILE -empty(w) DO
         q = point(w);
         T = T | p*q^ | dfsvisit(R,q,u);
         u = u | dom(T^);
         w = R^*p & -u OD
       RETURN T
  END.
```

Auch das folgende RelView-Programm `dfs`, das, als Hauptprogramm, die Tiefensuche für den gesamten Graphen durchführt und die Relation F eines DFS-Waldes $f = (V, F)$ abliefert, ergibt sich unmittelbar aus der Übertragung der entsprechenden DFS-Hauptprogramme der Literatur:

```
dfs(R)
  DECL F, v, p
  BEG  F = 0(R);
       v = 0n1(R);
       WHILE -empty(-v) DO
         p = point(-v);
         F = F | dfsvisit(R,p,v);
         v = v | p | dom(F^) OD
       RETURN F
  END.
```

Eine schon erwähnte Strukturinformation, die man bei einer Tiefensuche gewinnt, ist die Klassifizierung der Pfeile bzw. Kanten. Schlingen bereiten hier gewisse Schwierigkeiten, deshalb setzen wir zur Pfeilklassifizierung eines gerichteten Graphen $g = (X, R)$ die Relation R als irreflexiv voraus. Die erste Klasse von Pfeilen sind die des DFS-Waldes $f = (X, F)$. Sie werden auch *Baumpfeile* genannt. Neben den Baumpfeilen gibt es als zweite Klasse die *Vorwärtspfeile*. Diese sind keine Pfeile von F, gehören aber zur transitiven Hülle von F, d.h. überbrücken einen Weg mit Mindestlänge 2 im DFS-Wald. Der unmittelbar aus dieser Beschreibung sich ergebende relationale Term $R \cap \overline{F} \cap F^+$ für die Vorwärtspfeile führt zu

```
farcs(R,F) = R & -F & trans(F)
```

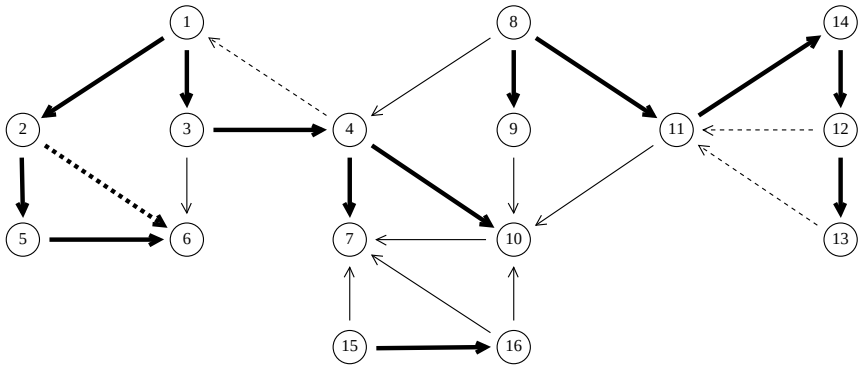
als entsprechende RelView-Funktion. Das Gegenstück zur Klasse der Vorwärtspfeile ist die Klasse der *Rückwärtspfeile*. Diese führen von einem Knoten x zu einem anderen Knoten y, wobei x von y aus im DFS-Wald $f = (X, F)$ erreichbar ist, d.h. F^+_{yx} gilt. Auch diese Definition

führt, über den relationalen Term $R \cap (F^{\mathsf{T}})^+$, unmittelbar zu einer Berechnung der Klasse in RelView mittels

```
barcs(R,F) = R & trans(F^).
```

Die Pfeile der verbleibenden Klasse werden *Seitwärtspfeile* oder auch *Querpfeile* genannt und verbinden Knoten von verschiedenen gerichteten Bäumen des DFS-Waldes. Sie können in RelView leicht mit Hilfe von `dfs`, `farcs` und `barcs` berechnet werden.

Das folgende RelView-Bild zeigt einen gerichteten Graphen, wobei die Pfeilklassifizierung anhand der verschiedenen Markierungsmöglichkeiten des Systems vorgenommen wurde. Alle Baumpfeile sind fett und mit durchgezogenen Linien gezeichnet. Die Vorwärtspfeile sind ebenfalls fett gezeichnet, aber mit gepunkteten Linien versehen. Zum Zeichnen der Rückwärtspfeile und Seitwärtspfeile wurde die normale Strichstärke verwendet; die Unterscheidung geschieht hier ebenfalls durch gepunktete (Rückwärtspfeile) bzw. durchgezogene (Seitwärtspfeile) Linien.



Eine Anwendung von Tiefensuche bietet sich häufig für Zusammenhangsprobleme an. Zwei klassische Beispiele hierzu sind beispielsweise die Bestimmung der Trennknoten bzw. der Brücken eines ungerichteten Graphen $g = (X, R)$, welche wir nun relational behandeln. Die entscheidende Strukturinformation zur Lösung dieser beiden Probleme ist die auf Tarjan [43] zurückgehende *lowpoint*-Funktion. Ihre Definition erfolgt üblicherweise unter Verwendung der DFS-Numerierung als Funktion von der Knotenmenge X in die natürlichen Zahlen; man vergleiche etwa mit der oben angegebenen Literatur. Man kann jedoch, aufbauend auf die Relation F des DFS-Waldes $f = (X, F)$ von g und die Relation $B = R \cap (F^{\mathsf{T}})^+$ der Rückwärtspfeile bezüglich f, diese Funktion mittels

$$lowpoint(x) = min_{F^+}\{y \in X : x = y \vee \exists\, z : F^*_{xz} \wedge B_{zy}\} \tag{48}$$

auch direkt auf den Knoten definieren, wobei der Index bei der Minimumsbildung ausdrückt, daß diese bezüglich der irreflexiven Ordnungsrelation F^+ erfolgt. Die Eindeutigkeit des Minimums in (48) folgt aus der Tatsache, daß die Menge $\{y \in X : F^*_{yx}\}$ der Vorfahren von x im DFS-Wald bezüglich F^+ linear geordnet ist[5].

Unser Ziel ist es, die Funktion *lowpoint* von (48) als eindeutige und totale Relation des Typs $[X \leftrightarrow X]$ durch einen relationalen Term in F und B zu beschreiben. Dazu benötigen wir zuerst die relationale Funktion

$$min : [X \leftrightarrow X] \times [X \leftrightarrow X] \to [X \leftrightarrow X] \qquad min(Q, S) = S \cap (Q \setminus \overline{S}). \tag{49}$$

[5]Umgangssprachlich ist $lowpoint(x)$ unter allen Knoten, zu denen es von x aus einen Weg in dem gerichteten Graphen mit der Relation $F \cup B$ gibt, der aus einer (möglicherweise leeren) Sequenz von Baumpfeilen besteht gefolgt von genau einem Rückwärtspfeil, derjenige Knoten y, der am frühesten beim Durchlaufen besucht wird. Dies gilt jedoch nur, wenn y nicht nach x besucht wird. Ansonsten definiert man $lowpoint(x) = x$.

Diese berechnet zu einer irreflexiven Ordnungsrelation Q und einer Relation S diejenige Relation, deren einzelne Spalten, als Vektoren des Typs $[X \leftrightarrow \mathbf{1}]$ aufgefaßt, genau die minimalen Elemente der entsprechenden Spalten von S darstellen. Die erste Spalte von $min(Q, S)$ stellt also beispielsweise die minimalen Elemente der durch die ersten Spalte von S dargestellten Teilmenge von X bezüglich Q dar. Mit Hilfe der durch (49) definierten relationalen Funktion bekommen wir nun für die gesuchte Relation $lowpoint : X \leftrightarrow X$ die komponentenbehaftete Beschreibung

$$lowpoint_{xy} \iff min(F^+, \mathsf{I} \cup (F^* \, B)^\mathsf{T})_{yx}, \tag{50}$$

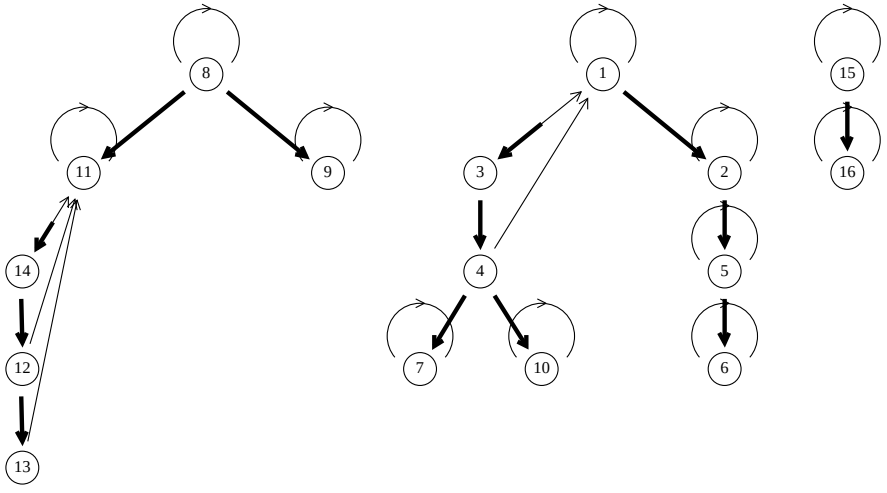
denn die einem Knoten x entsprechende Spalte der Relation $\mathsf{I} \cup (F^* \, B)^\mathsf{T}$ stellt als Vektor, wie man sich sofort klar macht, genau die Menge derjenigen y dar, deren Minimum in (48) zu bestimmen ist. Aus (50) bekommen wir schließlich durch Transposition und Abstraktion nach den beiden Indizes die gewünschte relationenalgebraische Darstellung

$$lowpoint = min(F^+, \mathsf{I} \cup (F^* \, B)^\mathsf{T})^\mathsf{T} \tag{51}$$

von *lowpoint*. Die Gleichung (51) führt aber auch sofort zu einer Realisierung der *lowpoint*-Berechnung in RelView durch das nachfolgende Programm:

```
lowpoint(R,F)
  DECL min(Q,S) = S & (Q \ -S);
       B, Q, S
  BEG  B = barcs(R,F);
       Q = trans(F);
       S = I(R) | (B | Q*B)^
       RETURN min(Q,S)^
  END.
```

Eine Anwendung von `dfs` und `lowpoint` ist in dem nachfolgenden Bild angegeben. Es zeigt den DFS-Wald des obigen gerichteten Graphen und die dadurch induzierte *lowpoint*-Relation, wobei die Pfeile des DFS-Waldes fett und die Pfeile von *lowpoint* in normaler Strichstärke gezeichnet sind.



Dieses Bild demonstriert anhand der zwei Pfeilpaare $\langle 1, 3\rangle$, $\langle 3, 1\rangle$ und $\langle 11, 14\rangle$, $\langle 14, 11\rangle$ auch eine der Möglichkeiten, in RelView entgegengesetzte Pfeile zu markieren.

Wir kommen nun zur ersten der oben erwähnten Anwendungen der Tiefensuche bei Zusammenhangsproblemen, der Berechnung der Trennknoten eines ungerichteten Graphen $g = (X, R)$.

Mit Hilfe der Relation *lowpoint* kann man eine Teilmenge der Pfeile des DFS-Waldes $f = (X, F)$ von g wie folgt auszeichnen: Ein Baumpfeil $\langle x, y\rangle$ heißt ein *Leitpfeil*, falls, unter Verwendung der originalen Funktionsnotation für *lowpoint*, die Beziehung $lowpoint(y) \in \{x, y\}$ gilt. Die Leitpfeile von f sind zur Bestimmung der Trennknoten von g äußerst hilfreich, denn es gilt (siehe beispielsweise [26]), daß eine Quelle des DFS-Waldes genau dann ein Trennknoten ist, falls von ihr mehr als ein Leitpfeil ausgeht, und eine Nichtquelle des DFS-Waldes genau dann ein Trennknoten ist, falls von ihm mindestens ein Leitpfeil ausgeht.

Um aus den bisherigen Überlegungen ein RelView-Programm zur Berechnung der Trennknoten zu erhalten, bieten sich die folgenden Schritte an. Man beschreibt zuerst relationenalgebraisch die Relation $larcs : X \leftrightarrow X$ der Leitpfeile mittels

$$larcs = F \cap (lowpoint^{\mathsf{T}} \cup F\, lowpoint^{\mathsf{T}}). \tag{52}$$

Diese Gleichheit ergibt sich aus der obigen sprachlichen Definition der Leitpfeile in unmittelbarer Weise. Anschließend formuliert man (52) als ein relationales Programm in der Sprache des RelView-Systems, was ebenfalls offensichtlich ist und zu

```
larcs(R,F)
  DECL Lp
  BEG  Lp = lowpoint(R,F)
       RETURN F & (Lp^ | F*Lp^)
  END
```

führt. Im dritten Schritt stellt man die zwei Mengen der Knoten, von denen mehr als ein Leitpfeil ausgeht bzw. mindestens ein Leitpfeil ausgeht, durch zwei Vektoren dar. Der erste Vektor entspricht genau dem Definitionsbereich des sogenannten mehrdeutigen Anteils (man vergleiche bezüglich dieser relationalen Konstruktion mit [41]) von *larcs* und ergibt somit zu $(larcs \cap larcs\ \overline{\mathsf{I}})\,\mathsf{L}$; der zweite Vektor entspricht genau dem Definitionsbereich von *larcs*, ist also $larcs\,\mathsf{L}$. Aus der obigen Charakterisierung ergibt sich der Vektor der Trennpunkte schließlich, indem man den ersten Vektor mit dem Vektor $\overline{F^{\mathsf{T}}\,\mathsf{L}}$ der Quellen von f schneidet, den zweiten Vektor mit dem Vektor $F^{\mathsf{T}}\,\mathsf{L}$ der Nichtqellen von f schneidet und beide Resultate vereinigt. In RelView sieht dies, unter Verwendung der Basisfunktion `dom` zur Berechnung des Definitionsbereichs-Vektors, wie folgt aus:

```
cuts1(R)
  DECL F, La, c
  BEG  F = dfs(R);
       La = larcs(R,F);
       c = (-dom(F^) & dom(La & La*-I(R))) | (dom(F^) & dom(La))
       RETURN c
  END.
```

Wir haben oben als zweite Anwendung von Tiefensuche noch die Berechnung der Brücken angesprochen und kommen nun zu diesem Problem. Die Relation der Brücken eines ungerichteten Graphen $g = (X, R)$ kann man ebenfalls mit Hilfe von *lowpoint* sehr einfach bestimmen. Unter Verwendung der originalen funktionalen Schreibweise gilt nämlich: Eine Kante $e : X \leftrightarrow X$ ist genau dann eine Brücke, wenn sie einen Baumpfeil $\langle x, y\rangle$ mit $lowpoint(y) = y$ enthält. Die Menge der Pfeile, deren Endknoten Fixpunkte der *lowpoint*-Funktion sind, kann man, wenn man *lowpoint* als Relation auffaßt, relationenalgebraisch durch $\mathsf{L}\,(lowpoint \cap \mathsf{I})$ beschreiben. Die Relation der Brücken ist die Symmetrisierung des Durchschnitts der Relation F der Baumpfeile

mit L (*lowpoint* ∩ I) und dies führt sofort zu

```
bridges(R)
  DECL F, Lp, Br
  BEG  F = dfs(R);
       Lp = lowpoint(R,F);
       Br = F & L(R)*(Lp & I(R))
       RETURN Br | Br^
  END
```

als RelView-Programm zur Berechnung der Brücken eines ungerichteten Graphen. □

Nach diesen Beispielen soll zum Abschluß dieses Abschnitts noch kurz ein spezieller Aspekt von RelView im Hinblick auf die Ausbildung in Algorithmik angesprochen werden. Gegenwärtig erfolgt an den Hochschulen die Vermittlung des algorithmischen Grundwissens in den entsprechenden Vorlesungen in der Regel durch die Erarbeitung von Algorithmen in der gewohnten mathematischen Vorgehensweise und deren Formulierung in einer semiformalen Art unter Verwendung von Schreibweisen gängiger Programmiersprachen zu Strukturierungszwecken; man vergleiche etwa mit den Bemerkungen in [31], Abschnitt 2.4. Auf eine „lauffähige" Implementierung, beispielsweise in Programmiersprachen wie C oder Modula-2, wird normalerweise verzichtet. Dies gilt insbesondere für graphentheoretische Algorithmen, wenn vielfältige Datenstrukturen einen großen Implementierungsaufwand erforden würden.

Es gibt zwar in der Forschung Informatiker, die grundsätzlich auf Implementierungen verzichten und eine Algorithmenentwicklung vor allem als intellektuelle Leistung betrachten[6], aber die meisten von uns wollen doch sehen, was die Ausführung eines Algorithmus auf einem Rechner bewirkt. Die Erfahrungen an den Hochschulen zeigen zudem, daß, besonders bei sehr komplizierten Algorithmen, durch das konkrete und interaktive Ausführen und Visualisieren von Zwischen- und/oder Endergebnissen bei den Studierenden in der Regel auch eine Verbesserung des algorithmischen Verständnisses erreicht werden kann. Anhand der Beispiele insbesondere dieses Abschnitts sollten Hinweise dafür gegeben werden, wie man RelView auch als ein Werkzeug in dieser Richtung verwenden kann, sofern es sich um Probleme auf diskreten Strukturen handelt, die relativ einfach durch Relationen modelliert werden können. Gleichzeitig sollte aber an dieser Stelle auch betont werden, daß der Anwendungsbereich des Systems bei der Algorithmenentwicklung eindeutig auf dieses Gebiet beschränkt ist, um so von vorneherein falsche Erwartungen der Benutzer zu verhindern.

8 Implementierung von Relationen mittels OBDDs

In der aktuellen Implementierung von RelView können gewisse Anwendungen nur für relativ kleine Eingaben durchgeführt werden. Der Grund dafür liegt oft in der Größe der benötigten Zwischenergebnisse, deren interne Darstellung als Boolesche Felder sehr viel Speicherplatz in Anspruch nimmt. Die Berechenbarkeit scheitert bei relationalen Spezifikationen zum Beispiel häufig daran, daß die zu einer Menge X zugehörige Potenzmengenrelation $\varepsilon : X \leftrightarrow 2^X$ benutzt wird. Wenn wir davon ausgehen, daß zur Darstellung eines Relationeneintrags 1 Bit im Speicher benötigt wird, nimmt eine solche Potenzmengenrelation als Boolesches Feld für $|X| = 25$ bereits über 100 MByte Speicherplatz in Anspruch. Da der Bildbereich der Potenzmengenrelation exponentiell wächst und bei der Hinzunahme von einem zusätzlichen Element in X mehr

[6] E.W. Dijkstra schreibt beispielsweise in einer Einleitung zu einem seiner Bücher: „None of the programs in this monograph, needless to say, has been tested on a machine".

als doppelt so viel Speicherplatz zur Darstellung benötigt wird, sind ca. 25 Elemente in der Grundmenge X bei einer durchschnittlich ausgestatteten Hardware eine obere Schranke für die Darstellbarkeit von ε. Dies ist für viele Anwendungen nicht befriedigend. Somit stellt sich die Frage nach einer besseren Implementierung von Relationen. In diesem Abschnitt skizzieren wir, wie Relationen und zugehörige Operationen mittels geordneter binärer Entscheidungsdiagramme dargestellt werden können und welche Vorteile dies für praktische Anwendungen bedeutet. Wir stellen zunächst die Datenstruktur und ihre grundlegenden Eigenschaften vor. Anschließend beschreiben wir eine Möglichkeit, Relationen mit ihrer Hilfe darzustellen, und erläutern an Beispielen die Implementierung von relationalen Konstanten und Operationen. Am Ende des Abschnitts demonstrieren wir schließlich am Beispiel der Kernberechnung bei gerichteten Graphen die Vorteile einer solchen Implementierung von Relationen.

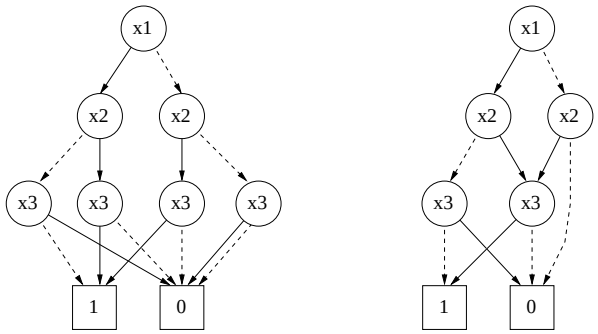
8.1 Geordnete binäre Entscheidungsdiagramme. Zur Darstellung von Booleschen Funktionen entwickelte R. Bryant in den 80er Jahren (siehe [21]) die neue graphenbasierte Datenstruktur der geordneten binären Entscheidungsdiagramme (ordered binary decision diagrams, kurz OBDDs). Ist π eine lineare Ordnung auf den Variablen einer n-stelligen Booleschen Funktion f, dargestellt durch die Sequenz $x_1, \ldots, x_n$ der Variablen, so ist ein OBDD bzgl. π ein gerichteter azyklischer Graph mit genau einer Wurzel und den folgenden Eigenschaften:

- Es gibt genau zwei Knoten ohne ausgehende Pfeile, die mit den beiden Konstanten 0 bzw. 1 markiert sind. Wir bezeichnen diese Knoten als die 0- bzw. 1-Senke.
- Jeder andere Knoten ist mit einer Variablen x_i markiert und hat genau zwei ausgehende Pfeile: einen 1-Pfeil und einen 0-Pfeil.
- Die Reihenfolge der Variablen auf jedem Weg ist mit der Variablenordnung π konsistent, d.h. führt von einem mit x_i markierten Knoten ein Pfeil zu einem mit x_j markierten Knoten, so gilt $x_i <_\pi x_j$.

Ein Berechnungspfad zu einer Eingabe $\langle b_1, \ldots, b_n \rangle \in \mathbb{B}^n$ ist der Weg von der Wurzel zu einer Senke des OBDDs, der in jedem mit x_i markierten Knoten dem b_i-Pfeil folgt. Mit OBDDs können nun Boolesche Funktionen dargestellt werden, indem man die folgende Festlegung trifft: Ein OBDD repräsentiert genau dann eine Boolesche Funktion $f : \mathbb{B}^n \to \mathbb{B}$, wenn der Berechnungspfad jeder Eingabe $\langle b_1, \ldots, b_n \rangle \in \mathbb{B}^n$ zur Senke mit der Markierung $f(b_1, \ldots, b_n)$ führt. Die Boolesche Funktion

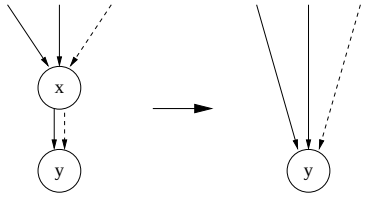
$$g : \mathbb{B}^3 \to \mathbb{B} \qquad g(x_1, x_2, x_3) = x_2 x_3 + x_1 \overline{x_2 x_3},$$

wobei wir im folgenden, wie in der Literatur üblich, Negation durch Überstreichung, Disjunktion durch Addition und Konjunktion durch Multiplikation darstellen, wird zum Beispiel durch die nachstehenden zwei OBDDs repräsentiert, in denen, wiederum wie üblich, die 0-Pfeile mit gestrichelten und die 1-Pfeile mit durchgezogenen Linien dargestellt sind.

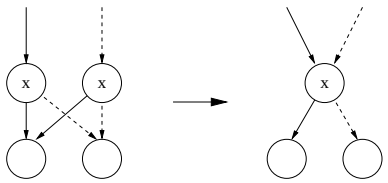


Zu jeder Booleschen Funktion existiert ein OBDD, der sie repräsentiert, da alle möglichen Berechnungspfade einfach einzeln dargestellt werden können. Wie man aber an den obigen Beispielen sieht, müssen Berechnungspfade für verschiedene Eingaben nicht disjunkt sein. Dies ist auch der Schlüssel zur Kompaktheit der Darstellung von Booleschen Funktionen bei der Benutzung von OBDDs. Isomorphe Untergraphen eines OBDDs können "zusammengelegt" werden und Knoten, die mit Variablen beschriftet sind, deren Belegung keinen Einfluß auf den Funktionswert hat, können entfallen. Solche Redundanzen können mit Hilfe der folgenden Reduktionsregeln entfernt werden:

- Eliminationsregel: Wenn 1- und 0-Pfeil eines Knotens v auf den gleichen Knoten u zeigen, dann eliminiere v und lenke alle eingehenden Pfeile auf u um.



- Isomorphieregel: Sind zwei Knoten u und v mit der gleichen Variablen markiert und führen ihre 1-Pfeile bzw. ihre 0-Pfeile jeweils zum gleichen Knoten, dann eliminiere u oder v und lenke alle eingehenden Pfeile auf den anderen Knoten um.



Kann keine der beiden Regeln angewendet werden, so heißt das OBDD reduziert. Auf diesem Wege läßt sich die zunächst fehlende Eindeutigkeit der Darstellung von Booleschen Funktionen durch OBDDs erreichen. Man kann nämlich zeigen, daß reduzierte OBDDs, die bzgl. einer festen Variablenordnung dieselbe Funktion repräsentieren, bis auf Isomorphie gleich sind. Anders verhält es sich, wenn wir verschiedene Variablenordnungen betrachten. In diesem Fall sind die reduzierten OBDDs nicht nur verschieden, sondern können in der Anzahl der Knoten sehr stark variieren. Im Extremfall hängt die Anzahl der OBDD-Knoten exponentiell von der Variablenanzahl ab, was für viele Anwendungen inakzeptabel ist. Leider ist das Finden einer optimalen Variablenordnung für OBDDs ein NP-vollständiges Problem. Verfahren zur heuristischen Konstruktion guter Variablenordnungen stehen daher im Mittelpunkt der Forschungsarbeiten.

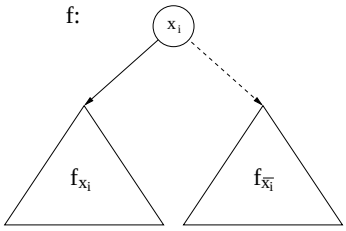
Da die Eindeutigkeit der Darstellung für eine effiziente Implementierung eine entscheidende Rolle spielt, gehen wir im folgenden immer von reduzierten OBDDs aus. Bevor wir auf die algorithmischen Eigenschaften eingehen, sind noch einige weitere Bemerkungen zur Größe von OBDDs angebracht. Die Anzahl der Knoten in einem reduzierten OBDD hängt nicht von der Anzahl der verwendeten Variablen ab, wie das zum Beispiel bei einer Tabellendarstellung von Booleschen Funktionen der Fall ist. Die konstanten Funktionen 0 und 1 werden immer durch OBDDs mit nur den beiden Senken beschrieben, egal welche Stelligkeit sie haben. Bei OBDDs kommt es nur darauf an, wie viele Redundanzen im Sinne der Reduktionsregeln in der darzustellenden Funktion vorhanden sind. Betrachtet man alle Funktionen einer festen Stelligkeit, so steigt zunächst die Größe der zugehörigen reduzierten OBDDs mit der Anzahl der 1-Werte, bis

diese die Hälfte aller möglichen Werte annimmt. Wächst die Anzahl der 1-Werte dann weiter, so sinkt die Größe der OBDDs. Dies ist dadurch zu erklären, daß zunächst viele gleiche Unterfunktionen existieren, die 0 liefern. Dank der Isomorphieregel werden diese durch nur einen Untergraphen dargestellt. Gibt es wiederum mehr 1- als 0-Werte, so hat man die Reduktionen auf der Seite der Unterfunktionen, die konstant 1 ergeben. Die Größenverteilung der OBDDs ist also generell bei Funktionen gleicher Stelligkeit symmetrisch bzgl. der 50%-Marke beim "Füllgrad" mit 1-Werten und nimmt bei dieser Marke ihr Maximum an. Gibt es nur Einsen bzw. Nullen als Werte, so handelt es sich um die konstante 1-Funktion bzw. 0-Funktion, die beide mit OBDDs minimaler Größe (zwei Knoten) darstellbar sind.

Neben der Kompaktheit der Darstellung zählt die gute algorithmische Handhabbarkeit von OBDDs zu den entscheidenden Vorteilen dieser Datenstruktur. Sie ist unter anderem darauf zurückzuführen, daß die sogenannte Shannon-Zerlegung einer Booleschen Funktion in einem zugehörigen OBDD widergespiegelt wird. Betrachten wir eine Variable x_i, so ist die Shannon-Zerlegung von $f : \mathbb{B}^n \to \mathbb{B}$ gegeben durch

$$f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}},$$

wobei $f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$ der positive und $f_{\overline{x_i}} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$ der negative Cofaktor von f bzgl. x_i sind. In einem f repräsentierenden OBDD mit x_i als der Wurzel finden wir die Cofaktoren als Untergraphen wieder, die unter dem mit x_i beschrifteten Knoten liegen:



Die meisten Operationen auf OBDDs können damit durch Rekursion realisiert werden, indem zunächst das Ergebnis für die beiden Cofaktoren berechnet wird und anschließend daraus das gesamte OBDD gebildet wird. Als ein einfaches Beispiel betrachten wir eine binäre Boolesche Operation $*$ (beispielsweise Konjunktion oder Disjunktion von Wahrheitswerten) und zwei Boolesche Funktionen $f, g : \mathbb{B}^n \to \mathbb{B}$. Mit Hilfe der Shannon-Zerlegung läßt sich dann die auf die Ebene der Booleschen Funktionen „geliftete" Operation $f * g$ mittels

$$f * g = x_i(f_{x_i} * g_{x_i}) + \overline{x_i}(f_{\overline{x_i}} * g_{\overline{x_i}})$$

berechnen. Bei einer rekursiven Berechnung der OBDDs F_1 für $f_{x_i} * g_{x_i}$ und F_2 für $f_{\overline{x_i}} * g_{\overline{x_i}}$ erhält man das OBDD für $f * g$, indem ein neuer mit x_i markierter Knoten eingeführt wird, dessen 1-Pfeil auf die Wurzel von F_1 und dessen 0-Pfeil auf die Wurzel von F_2 zeigen.

Setzt man dieses rekursive Verfahren direkt um, so müssen allerdings zuerst $2^n - 2$ Zwischenergebnisse berechnet werden, was keine befriedigenden Laufzeiten liefert. Die Redundanzen innerhalb der OBDDs erlauben jedoch eine effiziente Implementierung: Um $f * g$ zu berechnen, müssen alle Kombinationen $f' * g'$ für jede Unterfunktion f' von f und jede Unterfunktion g' von g vorhanden sein. Da aber jede Unterfunktion mit genau einem OBDD-Knoten korrespondiert, ist die Anzahl der auszuführenden Berechnungen durch das Produkt der Knotenanzahlen der f und g repräsentierenden OBDDs beschränkt. Während der Berechnung müssen dafür nur die bereits vorhandenen Zwischenergebnisse in einer Tabelle vermerkt werden, damit sie bei Bedarf wiederverwendet werden können. □

Eine Reihe weiterer Designüberlegungen erlaubt es, OBDDs effizient zu implementieren (siehe etwa [18]), womit sie neben ihrer Kompaktheit für praktische Anwendungen mit Booleschen Funktionen insgesamt eine sehr vorteilhafte Datenstruktur bieten. Bezüglich einer genaueren Darstellung von OBDDs sei auf die Literatur verwiesen, beispielsweise [22, 36, 25].

8.2 Anwendung auf die Darstellung von Relationen. Jede Relation kann duch binäre Codierung des Urbild- und Bildbereichs auf naheliegende Weise als Boolesche Funktion aufgefaßt werden. Wir wollen das anhand eines Beispiels erläutern: Es seien $X = \{a, b, c, d\}$, $Y = \{r, s\}$ und $R : X \leftrightarrow Y$ mit $R = \{\langle a, r\rangle, \langle c, r\rangle, \langle c, s\rangle\}$. In RelView haben wir also die Darstellung von R wie in dem linken der beiden nachfolgenden Bilder gegeben:

Nun codieren wir die Elemente des Urbild- und Bildbereichs durch binäre Zahlen in der Reihenfolge, wie sie in der Booleschen Matrix auftreten; man vergleiche mit dem rechten obigen Bild. Wir benutzen also die folgenden binären Codierungen $c_1 : X \to \mathbb{B}^2$ und $c_2 : Y \to \mathbb{B}$:

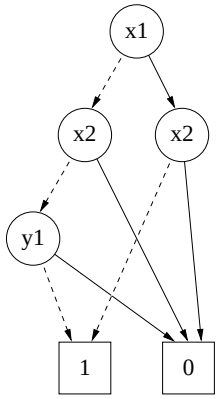
$$\begin{array}{lcllcl} c_1(a) & = & 00 & \qquad c_2(r) & = & 0 \\ c_1(b) & = & 01 & \qquad c_2(s) & = & 1 \\ c_1(c) & = & 10 & & & \\ c_1(d) & = & 11 & & & \end{array}$$

Die charakteristische Funktion $\chi_R : X \times Y \to \mathbb{B}$ von R, definiert durch $\chi_R(x, y) = 1$ falls R_{xy} und $\chi_R(x, y) = 0$ falls $\overline{R}_{xy}$, kann unter Verwendung der Codierungen durch die Festlegung

$$f(x_1, x_2, y_1) = \chi_R(c_1^{-1}(x_1, x_2), c_2^{-1}(y_1)) \tag{53}$$

direkt in eine Boolesche Funktion $f : \mathbb{B}^2 \times \mathbb{B} \to \mathbb{B}$ umgesetzt werden, um dann als ein OBDD bzgl. der Variablenordnung $x_1 < x_2 < y_1$ wie folgt dargestellt zu werden:

x_1	x_2	y_1	$f(x_1, x_2, y_1)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0



Diese direkte Umsetzung einer Relation in ein OBDD ist nur dann möglich, wenn die entstandene Boolesche Funktion total ist, d.h. nur dann, wenn die Größen des Urbild- und Bildbereichs der Relation Zweierpotenzen sind. Trifft diese Einschränkung nicht zu, so müssen die Undefiniertheiten der durch die Codierung entstandenen Funktion berücksichtigt werden.

Mit Hilfe von OBDDs können auch partielle Boolesche Funktionen dargestellt werden, indem man zusätzliche Knoten einführt. Diesen Aufwand kann man aber bei einer Implementierung

von Relationen umgehen. Wie man schon an dem obigen Beispiel sieht, ist der Urbild- und Bildbereich einer Relation aus der OBDD-Darstellung nicht erkenntlich. Es ist daher notwendig, diese Größen neben dem OBDD auch zu speichern. Will man nun eine Relation R darstellen, die durch Codierung analog zu (53) nur eine partielle Funktion f liefert, so kann man zu einer totalen Funktion f_R übergehen, die an den in f undefinierten Stellen den Wert 0 annimmt. Der OBDD für f_R zusammen mit der Dimension der dargestellten Relation liefern dann alle nötigen Informationen, die die ursprüngliche Relation R betreffen. Dieser methodisch etwas unsaubere Ansatz hat hier im Hinblick auf die Größe der OBDDs den Vorteil einer effizienteren Implementierung, an der wir letztendlich in erster Linie interessiert sind. Bei dieser Lösung muß man allerdings immer darauf achten, daß der "irrelevante" Bereich des OBDDs immer den Wert 0 liefert, was eventuell mit zusätzlichem Aufwand verbunden ist. Dies ist aber notwendig, da bei beliebigen Werten in diesem Bereich einige relationale Operationen wie zum Beispiel die Komposition falsche Ergebnisse liefern würden.

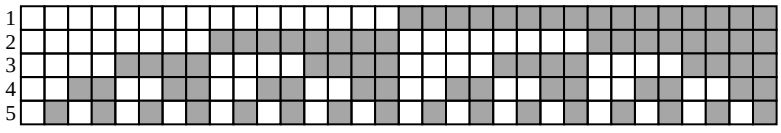
Um im folgenden eine Relation R durch ein OBDD darzustellen, betrachten wir also Codierungen $c_1 : X \to \mathbb{B}^n$ und $c_2 : Y \to \mathbb{B}^m$ laut obiger Beschreibung und die Boolesche Funktion $f_R : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}$ definiert durch

$$f_R(x_1, \ldots, x_n, y_1, \ldots, y_m) = \begin{cases} 1 & : \quad \chi_R(c_1^{-1}(x_1, \ldots, x_n), c_2^{-1}(y_1, \ldots, y_m)) = 1 \\ 0 & : \quad \text{sonst.} \end{cases}$$

Die Relation R wird dann durch ihre Dimension und das zu f_R zugehörige OBDD F_R bzgl. der Variablenordnung $x_1 < x_2 < \ldots < x_n < y_1 < \ldots < y_m$ repräsentiert. □

Bei der oben geschilderten Darstellung von Relationen durch OBDDs haben wir eine Variablenordnung gewählt, die zunächst willkürlich ist. Da die üblichen Algorithmen auf OBDDs gleiche Variablenordnung der Operanden verlangen und eine Änderung der Variablenordnung zur Laufzeit sehr aufwendig wird, ist es sinnvoll, bei der OBDD-Implementierung von Relationen in RelView mit einer festen Variablenordnung zu arbeiten. Aber welche Ordnung soll man wählen? Im Hinblick auf die beliebige Gestalt der Relationen, die in RelView verwendet werden können, bietet keine Variablenordnung besondere Vorteile, da es immer Relationen gibt, die zu großen OBDDs führen. Die Wahl der Variablenordnung kann aber durch die Größe oft verwendeter relationaler Konstanten motiviert werden. Wir orientieren und dabei an der für Spezifikationszwecke sehr oft verwendeten Potenzmengenrelation, die aber bei einer Implementierung durch Boolesche Felder bereits für Mengen ab ca. 25 Elementen nicht mehr dargestellt werden kann.

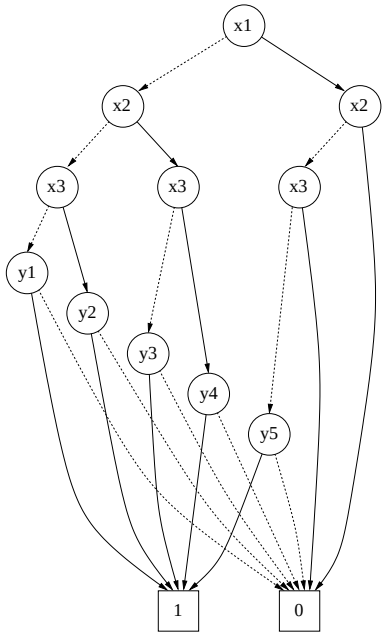
8.3 Potenzmengenrelationen als OBDDs. Zu einer endlichen nichtleeren Menge X ist die Potenzmengenrelation $\varepsilon : X \leftrightarrow X$ definiert durch ε_{xY} genau dann, wenn $x \in Y$. In RelView wird für die Darstellung als Boolesche Matrix eine feste Ordnung auf 2^X gewählt, die zu einer regelmäßigen Struktur führt. Für $X = \{1, 2, 3, 4, 5\}$ erhalten wir zum Beispiel das folgende Bild:



Wie man sich an diesem Beispiel klar machen kann, entsteht die Boolesche Matrix für n Elemente in X bildlich dadurch, daß die Matrix für $n - 1$ Elemente einmal oben durch eine 0-Zeile und einmal durch eine 1-Zeile ergänzt wird und beide nebeneinander gesetzt werden. Soll nun eine solche Relation durch einen OBDD dargestellt werden, indem die in 8.2 beschriebene Codierung durchgeführt wird, so tritt ein Phänomen auf, das sich sehr günstig auf die Größe des OBDDs

auswirkt. Durch die vorgegebene Ordnung auf 2^X entspricht die Codierung jeder Spalte genau dem „Inhalt“ dieser Spalte. Ein Eintrag in der zu einem Element i zugehörigen Zeile hängt also nur davon ab, ob die i-te Variable y_i zur Codierung des Bildbereichs den Wert 0 oder 1 annimmt. Die Belegung aller anderen Variablen hat hier keinen Einfluß mehr.

Im zugehörigen reduzierten OBDD entspricht also einer Zeile für das Element i ein Untergraph, der bis auf die 0- und 1-Senke nur einen mit y_i markierten Knoten enthält. Für die obige Beispielrelation erhält man folgendes OBDD:



Bei n Elementen in X werden weniger als $2 * n$ Knoten im OBDD benötigt, um die Aufschlüsselung nach Zeilen darzustellen (im Beispiel mit x-Variablen markierte Knoten). Insgesamt erhält man also einen OBDD mit weniger als $3 * n$ Knoten, womit nur ein lineares Wachstum bzgl. $|X|$ gegeben ist. Damit ist es möglich, die Potenzmengenrelation für alle in der Praxis relevanten endlichen Mengen darzustellen. □

Auch die weiteren in RelView vorimplementierten relationalen Konstanten O, L und I haben als OBDDs bei der in 8.2 angegebenen Variablenordnung eine befriedigende Größe. Es ist daher vorteilhaft in RelView mit der durch die Potenzmengenrelation motivierten festen Variablenordnung zu arbeiten.

Als nächstes wollen wir nun anhand der Komposition und der Transposition skizzieren, wie relationale Operationen auf OBDDs realisiert werden können. Da wir uns im wesentlichen auf eine OBDD-Implementierung von Relationen und relationalen Konstrukten konzentrieren wollen, gehen wir davon aus, daß eine Implementierung von OBDDs zur Verfügung steht. Es gibt mittlerweile viele sehr effiziente OBDD-Pakete, die auch die Implementierung derjenigen grundlegenden Operationen auf Booleschen Funktionen beinhalten, welche wir im weiteren ohne eine nähere Erläuterung benutzen wollen. Mehr Details findet man zum Beispiel in [36].

8.4 Komposition und Transposition auf OBDDs. OBDD-basierte Algorithmen für relationale Konstrukte können oft durch Zurückführung auf Boolesche Operationen realisiert werden. Ein Beispiel dafür ist die relationale Komposition, die mit Hilfe der Konjunktion und der existentiellen Quantifizierung gebildet werden kann. Um das näher zu erklären, betrachten wir für zwei Relationen $R : X \leftrightarrow Y$ und $S : Y \leftrightarrow Z$ die charakteristischen Funktionen χ_R und χ_S

für R und S, sowie $\chi_{R\,S}$ für die Komposition $R\,S$. Die Booleschen Funktionen f_R und f_S zu R bzw. S seien wie in 8.2 definiert. Mit $m, n, k \in \mathbb{N}$ seien die minimalen Anzahlen von Variablen bezeichnet, die für die Codierung der Mengen X, Y und Z nötig sind. Damit gilt zunächst:

$$\begin{aligned}
\chi_{R\,S}(x,z) = 1 &\Leftrightarrow (R\,S)_{xz} \\
&\Leftrightarrow \exists y : R_{xy} \wedge S_{yz} \\
&\Leftrightarrow \exists y : \chi_R(x,y) = 1 \wedge \chi_S(y,z) = 1 \\
&\Leftrightarrow \exists x_{m+1}, \ldots, x_{m+n} : f_R(x_1, \ldots, x_{m+n}) = 1 \wedge f_S(x_{m+1}, \ldots, x_{m+n+k}) = 1
\end{aligned}$$

Da die Konjunktion nur für Boolesche Funktionen mit gleichem Definitionsbereich definiert ist, gehen wir zu Funktionen $f'_R, f'_S : \mathbb{B}^m \times \mathbb{B}^n \times \mathbb{B}^k \to \mathbb{B}$ über, die f_R und f_S wie folgt erweitern:

$$\begin{aligned}
f'_R(x_1, \ldots, x_{m+n+k}) = 1 &\Leftrightarrow f_R(x_1, \ldots, x_{m+n}) = 1 \\
f'_S(x_1, \ldots, x_{m+n+k}) = 1 &\Leftrightarrow f_S(x_{m+1}, \ldots, x_{m+n+k}) = 1
\end{aligned}$$

Damit ist $\chi_{RS}(x,z) = 1$ äquivalent zu $\exists\, x_{m+1}, \ldots, x_{m+n} : (f'_R \cdot f'_S)(x_1, \ldots, x_{m+n+k}) = 1$. Unter Verwendung einer Operation $\exists v f$, die für eine Boolesche Funktion f durch $\exists v f = f_v + f_{\overline{v}}$ definiert ist, bekommen wir also

$$\chi_{RS}(x,z) = 1 \quad \Leftrightarrow \quad (\exists x_{m+1}, \ldots, x_{m+n}(f'_R \cdot f'_S))(x_1, \ldots, x_m, x_{m+n+1}, \ldots, x_{m+n+k}) = 1. \qquad (54)$$

Diese Äquivalenz wenden wir nun an, um aus den OBDDs F_R und F_S für R und S ein OBDD für die Komposition RS zu berechnen.

Es müssen zuerst die OBDDs für f'_R und f'_S bestimmt werden. Da die neuen Variablen in f'_R und f'_S im Vergleich zu f_R und f_S keinen Einfluß auf die Werte der Funktionen haben, fallen die damit markierten Knoten durch die Anwendung der Eliminationsregel weg. Die OBDDs F'_R und F'_S für f'_R und f'_S haben also genau dieselbe Struktur wie F_R bzw. F_S, bis auf die Markierungen der Knoten. F'_R und F_R sind sogar identisch, da die neuen Variablen „unten“ angefügt werden. Für F'_S muß jedoch jeder in F_S mit x_i markierte Knoten nun mit x_{i+m} markiert sein. Um dies zu realisieren benutzen wir eine Hilfsprozedur `shift`(F, l, r, s), die zu einem OBDD F und natürlichen Zahlen l, r und s ein OBDD F' folgender Form liefert: F' geht aus F dadurch hervor, daß jeder Knoten, der in F mit x_i markiert ist, in F' die Markierung x_{i+r} für $i < l$ bzw. x_{i+s} für $i \geq l$ bekommt. Der folgende Pseudocode in Modula-2-artiger Syntax zeigt, wie `shift` rekursiv realisiert werden kann:

```
shift(F,l,r,s)
  IF F = one OR F = zero
    THEN RETURN F
    ELSE index = Index der Wurzelmarkierung von F;
         IF index < l
           THEN node = neuer Knoten mit der Markierung index+r
           ELSE node = neuer Knoten mit der Markierung index+s
         END;
         T  = positiver Cofaktor von F;
         E  = negativer Cofaktor von F;
         ST = shift(T,l,r,s);
         SE = shift(E,l,r,s);
         F' = ITE(node,SE,ST);
         RETURN F'
  END
```

Hierbei gehen wir davon aus, daß *one* und *zero* die OBDDs für die konstanten Funktionen 1 und 0 sind. Die Funktionsprozedur `ITE`(v, G, H) soll ferner einen OBDD aufbauen, der eine mit v beschriftete Wurzel hat, von der aus der 0-Pfeil zu der Wurzel von G und der 1-Pfeil zu der Wurzel von H führen, und als Ergebnis den zugehörigen reduzierten OBDD liefern. Damit ist das Resultat von `shift` auch stets reduziert.

Unter Verwendung der Hilfsprozedur `shift` kann die Komposition von R und S sehr einfach berechnet werden. Als Eingabeparameter werden neben den OBDDs F_R und F_S für R und S die Anzahl m der Variablen für den Urbild- und n für den Bildbereich von R benötigt. Wir erhalten dann die folgende Funktionsprozedur:

```
comp(FR,FS,m,n)
  FS'  = shift(FS,n,m,m);
  F_RS = And(FR,FS');
  F_RS = Exist(<m+1,...,m+n>,F_RS);
  F_RS = shift(F_RS,m,0,-n);
  RETURN F_RS
```

Nach der Anwendung von `Exist`, welche die Operation $\exists v f$ aus (54) realisiert und, wie auch die Konjunktion `And`, in allen gängigen Implementierungen von OBDDs vorhanden ist, muß das Ergebnis wieder die Variablenreihenfolge erhalten, die für die Relation RS gilt, was auch mit Hilfe von `shift` realisiert werden kann.

Die Hilfsprozedur `shift` wird im Umgang mit Relationen-OBDDs sehr oft benötigt. Ein weiteres Beispiel für ihren Einsatz ist die Transposition von Relationen. Betrachtet man die Booleschen Funktionen f_R für eine Relation R und $f_{R^\intercal}$ für die transponierte Relation R^T, so sind sie gleich, wenn die Reihenfolge der Variablen für $f_{R^\intercal}$ so geändert wird, daß die Variablen für den Urbildbereich mit den Variablen für den Bildbereich vertauscht werden. D.h. es gilt

$$f_{R^\intercal}(x_{m+1}, \ldots, x_{m+n}, x_1, \ldots, x_m) = f_R(x_1, \ldots, x_m, x_{m+1}, \ldots, x_{m+n}).$$

Für die Berechnung des OBDDs für R^T aus dem für die originale Relation R reicht also eine Anwendung der Funktionsprozedur `shift` und wir erhalten die folgende Realisierung:

```
transp(FR,m,n)
  F_RT = shift(F_R,m,n,-m);
  RETURN F_RT
```

Auch wenn die Transposition von Relationen mittels `shift` einfach realisierbar ist, kann die Berechnung des zugehörigen OBDDs viel Aufwand erfordern. Mit der Funktionsprozedur `shift` wird hier nämlich ein in der Regel völlig anders strukturiertes OBDD aufgebaut, das durch die geänderte Variablenordnung wesentlich größer sein kann als das Ausgangs-OBDD (siehe 8.1). Diesen Aufwand kann man aber oft vermeiden, wenn die Anwendung der Transposition in Verbindung mit anderen relationalen Operationen stattfindet. Wir wollen das kurz anhand der Komposition erläutern.

Wir betrachten zwei Relationen $R : X \leftrightarrow Y$ und $S : Y \leftrightarrow Z$. Um $R^\mathsf{T} S$ zu berechnen, muß R^T nicht explizit bestimmt werden, wenn man statt `comp` eine leicht abgeänderte Funktionsprozedur verwendet. Die Transposition bedeutet für OBDDs ja nur eine Änderung der Variablenordnung, und nicht der dargestellten Booleschen Funktion. Diese Änderung kann bei der Komposition sofort berücksichtigt werden. Damit die Konjunktion in der Berechnung der Komposition ausgeführt werden kann, müssen die Variablenreihenfolgen der beiden Argumente gleich sein. In unserem Beispiel hat das erste Argument f_R zunächst die Variablen für X und dann die für

Y. Das zweite Argument f_S hat auch zunächst die Variablen für X und dann die für Z. Wir erreichen also eine gemeinsame Variablenreihenfolge mittels der Festlegungen

$$\begin{aligned} f'_R(x_1, \ldots x_{m+n+k}) = 1 &\quad \Leftrightarrow \quad f_R(x_1, \ldots x_{m+n}) = 1 \\ f'_S(x_1, \ldots x_{m+n+k}) = 1 &\quad \Leftrightarrow \quad f_S(x_1, \ldots, x_m, x_{m+n+1}, \ldots x_{m+n+k}) = 1. \end{aligned}$$

Im Vergleich zur ursprünglichen Funktionsprozedur `comp` müssen also lediglich die Variablenindizes in den Aufrufen von `shift` und `Exist` angepaßt werden. Dies führt zu der folgenden Funktionsprozedur:

```
comp_tl(FR,FS,m,n)
  FS'  = shift(FS,m,0,n);
  F_RS = And(FR,FS');
  F_RS = Exist(<1,...,m>,F_RS);
  F_RS = shift(F_RS,m,0,-n);
  RETURN F_RS
```

Analog kann auch die Komposition realisiert werden, bei der das zweite Argument oder sogar beide Argumente transponiert sind. □

Wir wollen nun am Beispiel der Kernberechnung von gerichteten Graphen demonstrieren, welche Vorteile eine OBDD-Implementierung von Relationen bieten kann. Wir entwickeln zunächst, aufbauend auf [12], eine relationale Spezifikation des Problems, schildern dann die Probleme bei der Ausführung des Prototyps mit der derzeitigen Version von RelView und zeigen schließlich, wie diese Probleme mit Hilfe von OBDDs reduziert werden können, was durch experimentelle Ergebnisse bestätigt wird.

8.5 Berechnung von Kernen. Es sei $g = (X, R)$ ein gerichteter Graph mit der Knotenmenge X und der Pfeilrelation $R : X \leftrightarrow X$. Eine Teilmenge $a \in 2^X$ von Knoten heißt *absorbierend* in g, wenn von jedem Knoten außerhalb von a mindestens ein Pfeil zu einem Knoten in a führt. Diese Eigenschaft wird offenbar durch die folgende logische Formel beschrieben:

$$\forall\, x : x \notin a \rightarrow \exists\, y : y \in a \wedge R_{xy} \tag{55}$$

Unter Benutzung der in (14) aufgeführten Beziehung des Rechtsresiduums und der Definition der Potenzmengenrelation $\varepsilon : X \leftrightarrow 2^X$ kann die Formel (55) ausdrückt werden durch

$$(\mathsf{L} \setminus (\varepsilon \cup R\,\varepsilon))^{\mathsf{T}}_a \tag{56}$$

mit dem Allvektor $\mathsf{L} : X \leftrightarrow \mathbf{1}$. Abstraktion nach dem Index a in (56) führt zu einer komponentenfreien Form und wir erhalten somit den Vektor $absorb(R) : 2^X \leftrightarrow \mathbf{1}$ zur Beschreibung aller in g absorbierenden Teilmengen von X durch einen Aufruf der relationalen Spezifikation

$$absorb : [X \leftrightarrow X] \rightarrow [2^X \leftrightarrow \mathbf{1}] \qquad absorb(R) = (\mathsf{L} \setminus (\varepsilon \cup R\,\varepsilon))^{\mathsf{T}}. \tag{57}$$

Eine Teilmenge $s \in 2^X$ von Knoten heißt *stabil* in g, wenn keine zwei Knoten in s durch einen Pfeil verbunden sind, wenn für s also gilt

$$\forall\, x : x \in s \rightarrow \forall\, y : y \in s \rightarrow \overline{R}_{xy}. \tag{58}$$

Wegen (13) und der Definition der Potenzmengenrelation $\varepsilon : X \leftrightarrow 2^X$ entspricht die Teilformel $\forall\, y : y \in s \rightarrow \overline{R}_{xy}$ von (58) (nach der vorherigen Umformung in eine existentiell-quantifizierte

Form) der Beziehung $(\overline{R\,\varepsilon})_{xs}$ und die gesamte Formel (58) wird somit durch $\forall\, x : (\overline{\varepsilon \cap R\,\varepsilon})_{xs}$, also, nach Entfernung der relationalen Negation, durch

$$\forall\, x : (\varepsilon \cap R\,\varepsilon)_{xs} \to \mathsf{O}_x \tag{59}$$

beschrieben, wobei der Nullvektor den Typ $[X \leftrightarrow \mathbf{1}]$ besitzt. Den Vektor aller in g stabilen Teilmengen von X erhalten wir dann, wegen (14), durch die Anwendung des Rechtsresiduums mit dem Nullvektor $\mathsf{O} : X \leftrightarrow \mathbf{1}$ und einer nachfolgenden Abstraktion nach dem Index s, d.h. durch den Aufruf der relationalen Spezifikation

$$\mathit{stable} : [X \leftrightarrow X] \to [2^X \leftrightarrow \mathbf{1}] \qquad \mathit{stable}(R) = (\varepsilon \cap R\,\varepsilon) \setminus \mathsf{O}. \tag{60}$$

Eine Teilmenge $k \in 2^X$ heißt schließlich *Kern* von g, wenn sie sowohl absorbierend als auch stabil ist. Aufgrund von (57) und (60) wird somit der Vektor $\mathit{kernel}(R) : 2^X \leftrightarrow \mathbf{1}$, der alle Kerne von g repräsentiert, durch den Aufruf von

$$\mathit{kernel} : [X \leftrightarrow X] \to [2^X \leftrightarrow \mathbf{1}] \qquad \mathit{kernel}(R) = \mathit{absorb}(R) \cap \mathit{stable}(R) \tag{61}$$

berechnet.

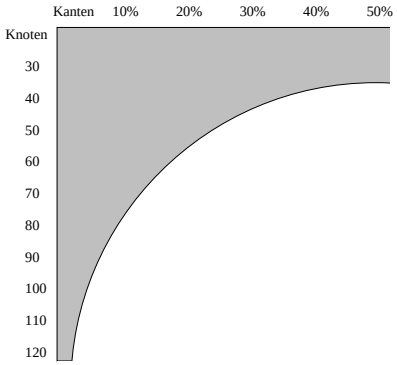
Die drei relationalen Spezifikationen von (57), (60) und (61) kann man direkt in RelView übertragen und damit die Kerne eines gerichteten Graphen mit Hilfe des Systems bestimmen. Bei einer direkten Übertragung ergeben sich jedoch Mehrfachberechnungen der Potenzmengenrelation. Deshalb ist es besser, zu einem relationalen Programm überzugehen, etwa zu

```
kernel(R)
  DECL L, O, epsi, absorb, stable
  BEG  L = Ln1(R);
       O = On1(R);
       epsi = epsi(L);
       absorb = (L \ (epsi | R * epsi))^;
       stable = (epsi & R * epsi) \ O
       RETURN absorb & stable
  END.
```

Angesetzt auf eine Boolesche $n \times n$-Matrix liefert das Programm `kernel` einen Booleschen Vektor der Länge 2^n. Eine spaltenweise Visualisierung der durch diesen Potenzmengenvektor beschriebenen Teilmenge der Potenzmenge der Knoten ist dann analog zu dem in 5.6 beschriebenen Verfahren möglich. □

Die Berechenbarkeit der Vektoren $\mathit{absorb}(R)$ und $\mathit{stable}(R)$ des letzten Beispiels mit Hilfe des RelView-Systems und somit auch der (eventuell auch leeren) Menge der Kerne eines gerichteten Graphen $g = (X, R)$ hängt nun direkt davon ab, ob die zugehörige Potenzmengenrelation $\varepsilon : X \leftrightarrow 2^X$ im System noch dargestellt werden kann. Wie bereits erwähnt, ist das bei der derzeitigen Implementierung von Relationen mit Booleschen Feldern nur für Graphen mit bis zu 25 Knoten möglich. Die OBDD-Implementierung bietet hier hingegen eine Lösung in viel mehr (teils auch in der Praxis relevanten) Fällen, da die Größe des ε-OBDDs nur linear mit der Knotenanzahl wächst. Bei der Berechnung der Kerne wirkt sich erst die Größe von $R\,\varepsilon$ maßgeblich aus. Durch die Komposition der beiden Relationen gehen in der Regel viele Redundanzen des ε-OBDDs verloren und es kann hier ein großes OBDD entstehen. Dies führt dann auch dazu, daß weitere Operationen, die mit diesem OBDD ausgeführt werden müssen, viel Rechenkapazität in Anspruch nehmen können. Wir beenden diesen Abschnitt mit einigen konkreten Ergebnissen.

8.6 Berechnung von Kernen mittels OBDDs. Im Rahmen einer experimentellen OBDD-Implementierung von Relationen haben wir viele gerichtete Graphen mit verschiedenen Knoten- und Pfeilanzahlen hinsichtlich der Kernberechnung untersucht. Dabei stellte sich heraus, daß sich, im Vergleich zur ursprünglichen RelView-Implementierung, die Kerne von teils wesentlich größeren Graphen berechnen lassen. Die folgende Graphik stellt die Berechenbarkeitsuntersuchungen graphisch dar. Ihr schraffierter Bereich gibt an, wann Kerne in Abhängigkeit von der Knotenanzahl und der prozentualen Anzahl aller möglichen Pfeile bestimmt werden konnten.



Generell konnten wir also für alle von uns gewählten Eingabegraphen mit bis zu ca. 40 Knoten alle Kerne berechnen bzw. feststellen, daß es keinen Kern gibt. Dies ist, im Vergleich zur bisherigen Implementierung, immerhin eine Steigerung der Knotenanzahl von etwa 60%.

In der Praxis hat man es aber sehr oft mit Graphen zu tun, die, mit n als Knotenanzahl, nur wenige der insgesamt möglichen $n * n$ Pfeile besitzen. Beispielsweise besitzt ein planarer Graph ohne entgegengesetzte Pfeile und Schlingen mit n Knoten höchstens $3*n-6$ Pfeile, also höchstens 2.9% aller Pfeile für $n = 100$ oder höchstens 1.48% aller Pfeile für $n = 200$. Je weniger Pfeile ein gerichteter Graph $g = (X, R)$ enthält, um so kleiner ist das OBDD, das seine Pfeilrelation R repräsentiert, und damit auch das $R\,\varepsilon$ darstellende OBDD. Wie die Graphik zeigt, ist es uns für solche „dünnen“ Graphen mit bis zu 120 Knoten gelungen, Kerne zu bestimmen. Angesichts der NP-Vollständigkeit des Tests auf Kerne schon für planare Graphen (siehe [27]), der Einfachheit des relationalen Programms `kernel` von Beispiel 8.5 und der Tatsache, daß man hier mit Größen der Ordnung 2^{120} hantiert, ist das ein (unserer Meinung nach) schönes Ergebnis. □

9 Abschließende Bemerkungen

In dem vorliegenden Bericht stellten wir eine auf Relationen basierende Methode zur formalen Erstellung von Prototypen für Programme auf diskreten Strukturen vor und demonstrierten ihre Rechnerunterstützung mittels RelView. Die Attraktivität des Ansatzes beruht insbesondere auf zwei Punkten.

Zuerst ist hier der Stil der Programme und der Beweise zu nennen. Ein wesentliches Merkmal des relationalen Programmierstils ist, daß Programme außerordentlich präzise, klar und kompakt sind. Sie lassen sich als Prototypen verwenden und sind sehr einfach in gängige imperative Programmiersprachen zu überführen. Ist ein relationales Programm nach der vorgestellten Methode hergeleitet, also korrekt bezüglich der ursprünglichen Problemspezifikation, so erübrigt sich nach einer sorgfältigen Übertragung in die endgültige Sprache auch eine weitere aufwendige Verifikation. Die Programmherleitungen selbst sind formal und, insbesondere was den relationenalgebraischen Anteil betrifft, ebenfalls außerordentlich präzise, klar und kompakt. Außerdem verfügt die komponentenfreie, algebraische Art und Weise, in der man im relationalen Kalkül beweist, inhärent über ein sehr hohes Sicherheitspotential. Dies reduziert die Gefahr fehlerhafter

Beweisschritte auf ein Minimum. Darüberhinaus gestattet der relationenalgebraische Beweisstil die Anwendung von Beweisassistenten wie RALF [29] oder RALL [38].

Neben einem mächtigen formalen Rahmen ist auch die Unterstützung der Validierung und der Intuition für eine formale Programmentwicklung wesentlich. Die Vorteile, die sich für die Methode aus der Unterstützung durch das RelView-System während des gesamten Entwicklungsprozeßes ergeben können, wurden schon in den Abschnitten 5 und 6 aufgezeigt; sie müssen deshalb an dieser Stelle nicht noch einmal wiederholt werden.

In den vergangenen Jahren wurde die in diesem Bericht beschriebene Programmentwicklungsmethode nicht nur in der entsprechenden Fachliteratur präsentiert (siehe [10, 12, 13, 14, 32, 15, 7, 16, 17]), sondern an der Universität Kiel auch in der Lehre erprobt. Über die dabei gemachten Erfahrungen soll nun ebenfalls kurz berichtet werden.

Die allgemeine Vorgehensweise war wie folgt: Vorausgesetzt wurden Grundkenntnisse in diskreter Mathematik, Algorithmik, Programmiersprachen und Programmiermethodik, wie sie üblicherweise im Grundstudium erworben werden. Auf diese aufbauend erfolgte dann die Vermittlung der theoretischen Grundlagen des Relationenkalküls (inklusive der dazu notwendigen verbandstheoretischen Begriffsbildungen) in Vorlesungen über relationale Methoden in der Informatik. In diesen Vorlesungen und den sie begleitenden Übungen wurde auch die Methode mittels kleinerer und überschaubarer Beispiele vorgeführt und der praktische Einsatz von RelView durch eine Computer-Projektionsanlage demonstriert. An die Vorlesungen schloß sich dann in der Regel ein Seminar an, in dem größere Fallstudien bearbeitet wurden. Einige Studierende vertieften das Thema schließlich noch einmal jeweils im Rahmen einer Diplomarbeit; siehe [46, 30, 47, 35][7].

Mit den bisherigen Ergebnissen kann man sehr zufrieden sein. Die Studierenden waren kontinuierlich und mit großer Motivation bei der Arbeit. Dies hat sicher auch mit dem in der Einleitung erwähnten Experimentieren beim Prototyping zu tun, welches durch die besondere Kürze und Ausdruckskraft der relationalen Funktionen und Programme und die Flexibilität und Visualisierungsmöglichkeiten von RelView sehr unterstützt wird. Als zwei schöne Resultate seien die Diplomarbeiten [46, 47] erwähnt. Im Laufe der Bearbeitung des sehr allgemein gehaltenen Themas von [46] stellte sich heraus, daß Petri-Netze ein neues und scheinbar auch besonders interessantes Gebiet für den Einsatz relationaler Methoden und des RelView-Systems darstellen. Dies führte zur relationalen Entwicklung von Algorithmen für einige fundamentale Probleme auf Bedingungs/Ereignis-Netzen, wie Erreichbarkeit, Lebendigkeit usw. Eine Überarbeitung des entsprechenden Teils von [46] wurde später beim Workshop „Tools and Applications for the Construction and Analysis of Systems“ akzeptiert; siehe [13]. Auch die Untersuchungen von [7] bezüglich einer relationenalgebraischen Behandlung von Petri-Netzen wurden durch [46] motiviert. In der Diplomarbeit [47] wurde anhand von Minimalgerüst-Algorithmen unter anderem erstmals der Versuch unternommen, Probleme auf bewerteten Graphen relationenalgebraisch zu lösen. Auch dies war erfolgreich, was man beispielsweise daran erkennt, daß überarbeitete Teile von [47] später als [15] im Tagungsband der Konferenz „Mathematics of Program Construction“ publiziert wurden. Neben solchen theoretischen Ergebnissen kamen von den Studierenden aber auch viele praktische Hinweise, die nicht nur halfen, noch vorhandene Fehler in RelView zu beheben, sondern auch zu einer beträchtlichen Verbesserung der Benutzerfreundlichkeit des Systems führten.

Zum Schluß soll noch kurz auf die derzeitigen Aktivitäten und mögliche zukünftige Arbeiten auf dem in diesem Bericht behandelten Forschungsgebiet eingegangen werden. Diese kann man grob in zwei Klassen einteilen.

[7] Weitere in den letzten Jahren angefertigte Diplomarbeiten befaßten sich auch mit Implementierungsarbeiten am RelView-System. So wurden insbesondere alle derzeit vorhandenen Algorithmen zum schönen Zeichnen von Graphen im Rahmen von drei Diplomarbeiten realisiert.

Die erste Klasse betrifft die Erweiterungen der Methode. Motiviert durch die Resultate von [47, 15] ist hier beispielsweise die Behandlung von bewerteten Graphen mittels Relationen derzeit ein intensives Forschungsgebiet. Darüberhinaus planen wir noch eine Vielzahl von Beispielen aus anderen Problembereichen relationenalgebraisch zu untersuchen, um das Repertoire an relationalen Gesetzen, Programmentwicklungstechniken, Transformationsregeln, generischen Programmen usw. ständig zu erweitern. Dem liegt die Idee eines „Werkzeugkastens“ zur relationalen Programmentwicklung zugrunde, wie sie in [7] beschrieben wird. Weitere zukünftige Arbeiten in dieser Richtung betreffen aber auch Verallgemeinerungen von Relationenalgebra. Hier sind wir insbesondere an der sogenannten sequentiellen Algebra interessiert, welche von Hoare und von Karger [33, 34] zur algebraischen Modellierung von reaktiven Systemen eingeführt wurde. Ihr Einsatz bietet sich vor allem bei Problemstellungen an, wo Sequenzen eine entscheidende Rolle spielen. Obwohl man, wie im Laufe dieses Berichts an verschiedenen Stellen gezeigt wurde, diese Datenstruktur unter bestimmten Umständen auch relational zufriedenstellend modellieren kann, erfordert eine relationale Behandlung von Sequenzen jedoch oft auch einen sehr großen technischen Aufwand und wird dann ziemlich unhandlich. Erste erfolgreiche Kombinationen von Relationenalgebra und sequentieller Algebra bei der Programmentwicklung findet man in [7] (Datenfluß-Analyse) und [35] (endliche Automaten).

Neben der Erweiterung der Methode arbeiten wir derzeitig auch an der Weiterentwicklung von RelView. Dies betrifft kleinere Verbesserungen und Erweiterungen, wie etwa die Fehlermeldungen des Parsers, die Benutzeroberfläche und eine ASCII-Eingabe und -ausgabe von Relationen und Graphen, hauptsächlich aber, den Ausführungen von Abschnitt 8 folgend, die OBDD-Version des Systems. Für die Zukunft planen wir, motiviert durch die bisherigen Erfahrungen, die Ergänzung von RelView um eine Reihe von sogenannten Satellitensystemen. Solch ein System soll sich an einem speziellen Anwendungsbereich orientieren und die dort übliche Darstellung der behandelten Objekte für die Eingabe in RelView aufbereiten bzw. die RelView-Ausgabe entsprechend zurücktransformieren. Beispielsweise wird ein endlicher Automat $\mathcal{A} = (Q, A, \Delta, s, F)$ relational als ein Tupel $\mathcal{R} = (\Delta_{a_1}, \ldots, \Delta_{a_n}, v_s, v_F)$ modelliert, wobei die Relationen $\Delta_{a_i} : Q \leftrightarrow Q$ die durch die Zeichen $a_i \in A$ möglichen Zustandsübergänge darstellen $(1 \leq i \leq n)$ und die Vektoren $v_s : Q \leftrightarrow \mathbf{1}$ und $v_F : Q \leftrightarrow \mathbf{1}$ den Anfangszustand $s \in Q$ bzw. die Menge der Endzustände $F \subseteq Q$ beschreiben (siehe [35]). Einem RelView-Satellitensystem für Automaten fällt somit als Hauptaufgabe zu, die übliche Darstellung von $\mathcal{A}$ als gerichteten knoten- und pfeilmarkierten Graphen in das Relationentupel $\mathcal{R}$ überzuführen und umgekehrt aus $\mathcal{R}$ eine „schöne“ graphische Darstellung von $\mathcal{A}$ zu zeichnen.

Literatur

[1] Aho A.V., Hopcroft J.E., Ullman J.D.: The design and analysis of computer algorithms. Addison Wesley (1974)

[2] Aigner M.: Diskrete Mathematik. Vieweg (1993)

[3] Bauer F.L., Wössner H.: Algorithmische Sprache und Programmentwicklung. 2. Auflage, Springer (1998)

[4] Bauer F.L. et al.: The Munich Project CIP, Volume I: The wide spectrum language CIP-L. LNCS 183, Springer (1985)

[5] Behnke R., Berghammer R., Schneider P.: Machine support of relational computations. The Kiel RelView system. Bericht 9711, Institut für Informatik und Praktische Mathematik, Universität Kiel (1997)

[6] Behnke R., Berghammer R., Meyer E., Schneider P.: RelView - A system for calculating with relations and relational programming. In: Astesiano E. (ed.): Proc. Conference on Fundamental Approaches to Software Engineering (FASE '98), LNCS 1382, Springer, 318-321 (1998)

[7] Behnke R.: Transformationelle Programmentwicklung im Rahmen relationaler und sequentieller Algebren. Dissertation, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel (1998)

[8] Berghammer R.: Zur formalen Entwicklung von graphentheoretischen Algorithmen durch Transformation. Dissertation, Institut für Informatik, Technische Universität München (1984)

[9] Berghammer R., Zierer H.: Relational algebraic semantics of deterministic and nondeterministic programs. Theoretical Computer Science 43, 123-147 (1986)

[10] Berghammer R.: Computing the cut completion of a partially ordered set – An example for the use of the RelView -system. Report Nr. 9205, Fakultät für Informatik, Universität der Bundeswehr München (1992)

[11] Berghammer R., Schmidt G.: RelView – A computer system for the manipulation of relations. In: Nivat M., Rattray C., Rus T., Scollo G. (eds.): Proc. 3rd Conference on Algebraic Methodology and Software Technology (AMAST '93), Workshops in Computing, Springer, 405-406 (1993)

[12] Berghammer R., Gritzner T., Schmidt G.: Prototyping relational specifications using higher-order objects. In: Heering, J., Meinke, K., Möller, B., Nipkow, T. (eds.): Proc. Int. Workshop on Higher Order Algebra, Logic and Term Rewriting (HOA '93), LNCS 816, Springer, 56-75 (1994)

[13] Berghammer R., Karger B. von, Ulke C.: Relation-algebraic analysis of Petri nets with RelView. In: Margaria T., Steffen B. (eds.): Proc. 2nd Workshop on Tools and Applications for the Construction and Analysis of Systems (TACAS '96), LNCS 1055, Springer, 49-69 (1996)

[14] Berghammer R., Karger B. von: Algorithms from relational specifications. In: Brink C., Kahl W., Schmidt G. (eds.): Relational methods in Computer Science. Advances in Computing Science, Springer, 131-149 (1997)

[15] Berghammer R., Karger B. von, Wolf A.: Relation-algebraic derivation of spanning tree algorithms. In: Jeuring J. (ed.): Proc. 4th Conference on Mathematics of Program Construction (MPC '98), LNCS 1422, Springer, 23-43 (1998)

[16] Berghammer R.: Combining relational calculus and the Dijkstra-Gries method for deriving relational programs. Information Sciences: An International Journal (in Druck)

[17] Berghammer R., Hoffmann T.: Deriving relational programs for computing kernels by reconstructing a proof of Richardson's theorem. Science of Computer Programming (in Druck)

[18] Brace K. S., Rudell R. L., Bryant R. E.: Efficient implementation of a BDD package. In: Proc. 27th ACM/IEEE Design Automation Conference, 40-45 (1990)

[19] Brink C., Kahl W., Schmidt G. (eds.): Relational methods in Computer Science. Advances in Computing Science, Springer (1997)

[20] Budde R., Kuhlenkamp K., Matthiassen H., Züllinghoven H. (eds.): Approaches to prototyping. Springer (1984)

[21] Bryant R. E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, C-35:677-691 (1986)

[22] Bryant R. E.: Symbolic Boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys 24, 293-318 (1992)

[23] Chin L.H., Tarski A.: Distributive and modular laws in the arithmetic of relation algebras. University of California Publications in Mathematics Univ. of California Publ. in Mathematics (new series) 1, 341-384 (1951).

[24] Dijkstra E.W.: A discipline of programming. Prentice-Hall (1976)

[25] Drechsler R., Becker B.: Graphenbasierte Funktionsdarstellung: Boolesche und Pseudo-Boolesche Funktionen. Teubner (1998)

[26] Emden-Weinert T., Hougardy S., Kreuter B., Prömel H.J., Steger A.: Einführung in Graphen und Algorithmen. Vorlesungs-Manuskript (vorläufige Fassung vom 13. September 1996), Institut für Informatik, Humbold-Universität Berlin, erhältlich über WWW an der URL `www.informatik.hu-berlin.de/Institut/struktur/algorithmen/ga`

[27] Fraenkel A.S.: Planar kernel and Grundy with $d \leq 3, d_{out} \leq 2, d_{in} \leq 2$ are NP-complete. Disc. Appl. Math. 3, 257–262 (1981)

[28] Gries D.: The science of computer programming. Springer (1981)

[29] Hattensperger C.: Rechnergestütztes Beweisen in heterogenen Relationenalgebren. Dissertation, Fakultät für Informatik, Universität der Bundeswehr München (1997)

[30] Hoffmann T.: Formale Verifikation relationaler Programme mittels Relationenalgebra und wp-Kalkül. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel (1997)

[31] Jungnickel D.: Graphen, Netzwerke und Algorithmen. BI Wissenschaftsverlag, 3. Auflage (1994)

[32] Karger B. von, Berghammer R.: Computing kernels in directed bichromatic graphs. Information Processing Letters 62, 5-11 (1997)

[33] Karger B. von, Hoare C.A.R.: Sequential calculus. Information Processing Letters 53, 123-130 (1995)

[34] Karger B. von: Temporal algebra. Habilitationsschrift, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel (1998)

[35] Lindecke F.: Relationenalgebraische Untersuchung von automatentheoretischen Problemen. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel (1999)

[36] Meinel Ch., Theobald T.: Algorithmen und Datenstrukturen im VLSI-Design. Springer (1989)

[37] Noltemeier H.: Reduktion von Präzedenzstrukturen. Zeitschrift für Operations Research 20, 151-159 (1976)

[38] Oheimb D. von, Gritzner T.: RALL: Machine-supported proofs for relation algebra. In: McCune W. (ed.): Proc. 14th Conference on Automated Deduction (CADE 14), LNAI 1249, Springer, 380-394 (1997)

[39] Paige R., Koenig S.: Finite differencing of computable expressions. ACM TOPLAS 4, 402-454 (1982)

[40] Promberger G, Blaschek G: Software Engineering: Prototypen und objektorientierte Software-Entwicklung. Hanser (1993)

[41] Schmidt G., Ströhlein T.: Relations and graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoretical Computer Science, Springer (1993)

[42] Sharir M.: Some observations concerning formal differentiation of set-theoretic expressions. ACM TOPLAS 4, 196-226 (1982)

[43] Tarjan R.: Depth-first-search and linear graph algorithms. SIAM Journal Comp. 4, 146-160 (1972)

[44] Tarski A.: On the calculus of relations. Journal of Symbolic Logic 6, 73-89 (1941).

[45] Tarski A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. 5, 285-309 (1955)

[46] Ulke C.: Rechnergestützte Spezifikation und Entwicklung relationaler Algorithmen. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel (1995)

[47] Wolf A.: Relationale Untersuchung von Inzidenzgraphen. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel (1997)

[48] Warshall S.: A theorem on Boolean matrices. Journal of the ACM 9, 11-12 (1962)