Tagungsband des GI-Workshops

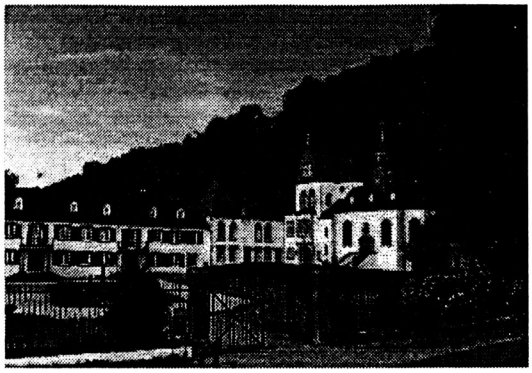
Softwarevisualisierung 2000

11.-12. Mai 2000, Schloß Dagstuhl

Hrsg.: Stephan Diehl, Andreas Kerren

Technischer Bericht A/01/2000

FR 6.2 – Informatik
Universität des Saarlandes, Postfach 15 11 50
66041 Saarbrücken , GERMANY
Phone: ++49-681-3023915
diehl@cs.uni-sb.de



Softwarevisualisierung

SV 2000

Internationales Begegnungs- und
Forschungszentrum für Informatik, Schloß Dagstuhl

11-12 Mai 2000

Workshop in Zusammenarbeit mit den Fachgruppen 2.1.1 "Softwaretechnik" und 4.1.2 "Imaging und Visualisierungstechniken" der Gesellschaft für Informatik

Ausgehend von der Erkenntnis, daß der Mensch bildliche Eindrücke besonders effizient verarbeitet und konkrete Objekte besser erfassen kann als abstrakte Beschreibungen, suchte man in den letzten Jahren nach Möglichkeiten, die unsichtbare Maschine Software sichtbar zu machen.

Dabei kann eine Visualisierung auf unterschiedliche Art erfolgen. Die Konstruktion von Programmen sollte z.B. im gegenständlichen Sinn möglich sein, d.h. vereinfacht ausgedrückt, Programme sollten durch Zusammenstecken und Ausprobieren entstehen. Bei dieser Art der Programmierung kommen vorwiegend grafische Notationen und interaktiv manipulierbare Softwarekomponenten zum Einsatz. Als Bezeichnung dafür hat sich der Begriff visuelle Programmierung etabliert.

Eine andere Art ist etwa die Verwendung von Algorithmenanimationen, in denen dem Beobachter der Ablauf eines Algorithmus visuell dargestellt wird. Man beobachtet den Algorithmus sozusagen bei der Arbeit. Eine solche Animation kann unterschiedlichen Zwecken dienen. Sie kann stark abstrahieren und versuchen, die grundlegende Idee des Algorithmus zu veranschaulichen, sie kann aber auch sehr nah an der Implementierungssprache bleiben, um etwa als graphischer Debugger eingesetzt zu werden.

Softwarevisualisierung wird in vielen Bereichen der Informatik betrieben und die Papiere häufig auf den Workshops und Konferenzen dieser Bereiche veröffentlicht. Wer im deutschsprachigen Raum auf diesem Gebiet aktiv ist, ist daher schwer zu überschauen. Der Workshop soll Softwarevisualisierern ein Forum bieten, sich kennenzulernen und gemeinsame Projekte zu planen.

ORGANISATION

Stephan Diehl (Universität des Saarlandes)

PROGRAMMKOMITEE

Stephan Diehl (Universität des Saarlandes)
Gitta Domik (Universität Paderborn)
Daniel A. Keim (Universität Halle)
Andreas Kerren (Universität des Saarlandes)
Heinrich Müller (Universität Dortmund)
Petra Mutzel (Technischen Universität Wien)
Reinhard Wilhelm (Universität des Saarlandes)

Inhaltsverzeichnis

- **Automatisches Zeichnen von UML-Klassendiagrammen durch den Sugiyama-Algorithmus**
  Holger Eichelberger, Universität Würzburg
  Seite: 1

- **Ablaufvisualisierung für nebenläufige Java-Programme mit UML Interaktionsdiagrammen**
  Katharina Mehner, Universität Paderborn
  Annika Wagner, Universität Paderborn
  Seite: 15

- **Datenstrukturen visualisieren und animieren mit DDD**
  Andreas Zeller, Universität Passau
  Seite: 29

- **Über generisches visuelles Programmieren**
  Berthold Hoffmann, Universität Bremen
  Mark Minas, Universität Erlangen-Nürnberg
  Seite: 41

- **KIEL - Ein Computersystem zur Visualisierung der Auswertung von funktionalen Programmen**
  Rudolf Berghammer, Universität Kiel
  Seite: 53

- **Generierung interaktiver Animationen von Berechnungsmodellen**
  Stephan Diehl, Universität des Saarlandes
  Andreas Kerren, Universität des Saarlandes
  Seite: 65

- **Visualisierung von ausgewählten Lehrinhalten der Informatik - Beispielanimationen aus dem Projekt VIVALDI**
  Rainer Oechsle, Fachhochschule Trier
  Dieter Jahn, Fachhochschule Trier
  Seite: 79

- **Ein System zur Visualisierung von Algorithmen und Datenstrukturen mit LEDA**
  Stefan Näher, Universität Trier
  Oliver Zlotowski, Universität Halle-Wittenberg
  Seite: 91

- **Algorithmus-Animation in Lehr- und Lernsystemen der Computergraphik**
  Wolfram Luther, Universität Duisburg
  Seite: 103

- **Algorithmen lernen mit interaktiven Visualisierungen durch Betrachten, Nachgehen und Experimentieren**
  Nils Faltin, Universität Oldenburg
  Seite: 115

Automatisches Zeichnen von UML-Klassendiagrammen durch den Sugiyama-Algorithmus

Holger Eichelberger

Lehrstuhl für Informatik II, Am Hubland, 97074 Würzburg
eichelberger@informatik.uni-wuerzburg.de

Zusammenfassung SugiBib ist ein Framework, das eine allgemeine Implementierung des Sugiyama-Algorithmus bereitstellt. Es wurde ausgeprägt, um das Layout von UML-Klassendiagrammen zu berechnen. In diesem Beitrag wird das Framework und der praktische Einsatz im Software-Entwurf und in der Lehre beschrieben.

Schlüsselwörter: Sugiyama, Klassendiagramme, UML, st-Nummerierung, Manhattan-Layout, gridline-Layout, Framework, spring embedder, graph drawing, ELP, graph folding

1 Einleitung

Im Rahmen des Software-Engineering ergibt sich im praktischen Einsatz häufig die Notwendigkeit, das Layout von Software-Entwurfsdiagrammen anhand einer textuellen Beschreibung automatisch zu berechnen. Einsatz für diese Technik findet sich u.a. im inkrementellen Round-Trip-Engineering, wie auch im Re-Engineering bestehender Software.
Grundziel der Entwicklung des in JAVA geschriebenen Frameworks SugiBib war eine möglichst allgemeine Implementierung des Sugiyama-Algorithmus [10]. Besondere Aufmerksamkeit sollte auf allgemeine Entwurfsziele, wie Erweiterbarkeit, Wartbarkeit und Wiederverwendbarkeit gerichtet werden. Das Framework wurde zur Berechnung des Layouts von UML-Klassendiagrammen ausgeprägt und in Form einer einfachen Applikation, die zunächst nur Eingaben in Form von UMLscript [7] bearbeitet, verfügbar gemacht. Mit der Veröffentlichung der UML Version 1.3 im Jahr 1999 wurden die Fähigkeiten der UML-Ausprägung von SugiBib modifiziert und erweitert. Seit Januar 2000 steht eine Server-Variante auf dem WWW-Server unseres Lehrstuhls für Interessierte zur Verfügung. Zudem sind diverse Erweiterungen, die einen größeren Anwendungsbereich erschließen, geplant.
Im folgenden Abschnitt werden zunächst kurz die verwendeten Algorithmen beschrieben. Danach gibt der Abschnitt „Das Framework SugiBib" einen Einblick in die Struktur unseres Frameworks. Abschließend folgt eine Übersicht über Anwendungen und zukünftige Entwicklungen.

2 Sugiyama- und Seemann-Algorithmus

In der Grundform besteht der Vorschlag von K. Sugiyama zur Visualisierung hierarchischer Datenstrukturen [10] aus einem mehrstufigen Algorithmus:

1. Die Eingabe, ein gerichteter azyklischer Graph, wird in der ersten Stufe auf einzelne Ränge verteilt. Lösungsansätze findet man in den Methoden der linearen Programmierung oder dem Netzwerk-Simplex-Algorithmus.
2. Die dabei entstandene Hierarchie wird durch eine zweite Stufe, die die Kantenkreuzungen zwischen zwei benachbarten Rängen minimiert, nachbearbeitet. „Heuristics are appropriate since minimizing edge crossings in layouts of ranked graphs is NP-complete, even for only two ranks"[3].
3. Die dritte Stufe berechnet die Koordinaten der Knoten und Kanten. Die allgemeine Lösung betrachtet ein lineares Programm, dessen Komplexität mit der Größe der zu betrachtenden Matrix ($VE+E^2$ Einträge mit V Knoten und E Kanten) wächst.

Grundlegend für eine Implementierung ist die Arbeit von Gansner et. al. [3], in der diverse Ansätze für Approximationen und Kompromisse in Form von Heuristiken vorgestellt werden. Wir verwenden den Netwerk-Simplex-Algorithmus zur Rangberechnung eines azyklischen Graphen, Median- oder Barycenter-Gewichte in einer Sortierheuristik zur Kantenkreuzungsminimierung und mehrere, teilweise interferierende Einzelheuristiken für die Koordinatenberechnung.
Ein UML-Klassendiagramm besteht aus einer Menge rechteckiger Knoten, die i.d.R. unterschiedliche Größe besitzen, Vererbungs- und Assoziationskanten (siehe Abbildung 1). Kanten und Knoten besitzen jeweils eine vertikale und eine horizontale Ausdehnung. Knoten werden in zudem in einen Außenbereich und einen Innenbereich unterteilt (siehe unten). Eine Anwendung des oben kurz skizzierten Algorithmus auf UML-Klassendiagramme ist ohne weitere Anpassungen nicht möglich, da ein Klassendiagramm i.allg. einen zyklischen Graphen darstellt.

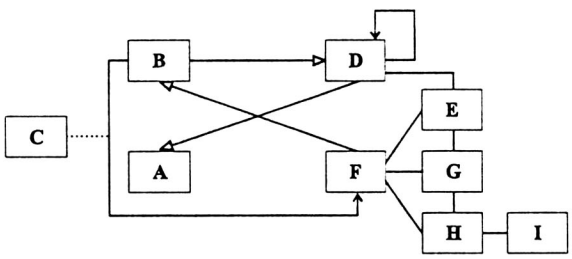


Abbildung 1. Ein beliebiges Klassendiagramm vor Anwendung des Seemann-Algorithmus. Kanten mit Pfeilen wie zwischen den Knoten D und A sind Vererbungskanten, die anderen Kanten stellen Assoziationen dar. Knoten C ist eine Assoziationsklasse dar, Knoten D besitzt eine reflexive Assoziation.

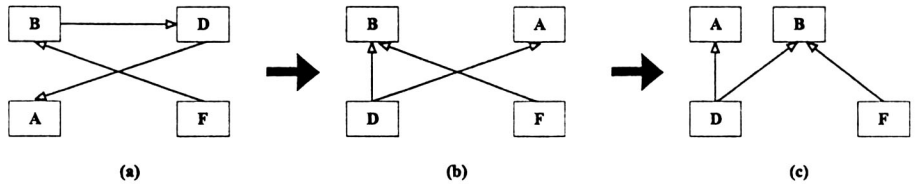


Abbildung 2. Entfernung der Assoziationskanten und zugehörigen Knoten (a), Rangzuweisung (b) und Minimierung der Kantenkreuzungen (c) am Beispielgraphen.

J. Seemann beschreibt in [9] ein geeignetes Konzept, im Folgenden kurz Seemann-Algorithmus genannt, das die Anwendung des Sugiyama-Algorithmus auf Klassendiagramme ermöglicht. Grundlegend für den Seemann-Algorithmus ist die Beobachtung, dass ein Klassendiagramm einen azyklischen Graph in Form des Vererbungsgraphen enthält. Entfernt man also temporär die Assoziationskanten aus dem Eingabegraphen, kann man die ersten zwei Schritte des Sugiyama-Algorithmus ohne Modifikationen anwenden.
Problematisch ist die Existenz reflexiver Assoziationen, also Assoziationskanten, die gleichen Start- und Endknoten besitzen. Diese stellen Zykel im Graph dar und erschweren die Beschreibung der Algorithmen. Seemann folgend, werden diese Kanten im Vorbereitungsschritt entfernt und als Information den betreffenden Knoten hinzugefügt. Die Kanteninformation kann dann z.B. in Attribute umgewandelt werden. Sollte der Graph einen Wald darstellen und keine eindeutige Wurzel besitzen, wird ein virtueller Knoten als Wurzel in den Graph eingefügt. Virtuelle Knoten sind in der Regel nicht sichtbar und dienen z.B. zur Formatierung langer Kantenketten, die nicht durch eine einzige Linie dargestellt werden können.

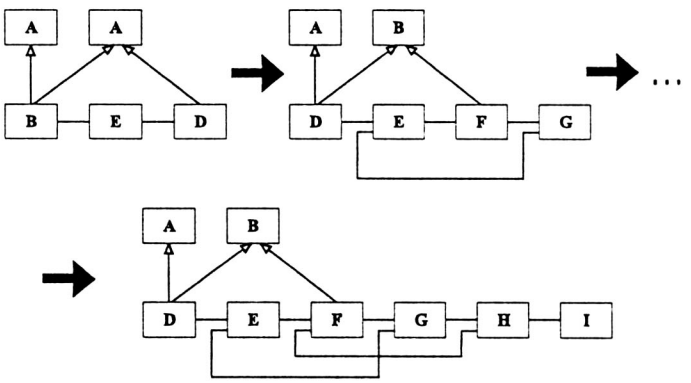


Abbildung 3. Inkrementelle Layout-Erweiterung (incremental extension) nach Seemann.

In einem zusätzlichen Schritt, der inkrementellen Erweiterung, wird das Grundlayout des Vererbungsgraphen durch die entfernten Assoziationkanten angereichert und eine Approximation des Eingabegraphen aufgebaut.
In Abhängigkeit von der Anzahl der Assoziationskanten werden ggf. Ränge in weitere Unterränge aufgeteilt und zwischen den existierenden Rängen eingefügt (siehe Abbildung 4). Eine Optimierung eines Unterrangs bezüglich der Kantenkreuzungen kann z.B. durch einen matrixbasierten Algorithmus, der durch st-Nummerierung und Strafgewichte gesteuert wird, erreicht werden.

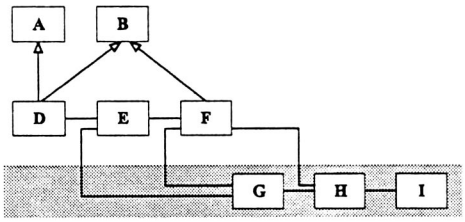


Abbildung 4. Aufspalten eines Rangs in einen Unterrang (grau hinterlegt) und Optimierung der Anzahl der Kantenkreuzungen.

Für die Koordinatenberechnung schlägt Seemann ein geteiltes Layout vor: Die Assoziationskanten werden orthogonal dargestellt, die Vererbungskanten direkt gezeichnet. Assoziationskanten verlassen Knoten nur an den vertikalen Knotenseiten, Vererbungskanten nur an den horizontalen Seiten. Das Lesen der Diagramme wird durch die genannten Darstellungstechniken für Kanten erheblich vereinfacht. Die Abbildungen 2 mit 5 entsprechen diesem Vorschlag.
Die Koordinatenberechnung garantiert zunächst, dass Kanten, die über mehrere Ränge geführt werden müssen, in Kantenketten aufgebrochen werden. Anschließend werden die Platzbedürfnisse der Knoten und Kanten berechnet. Unter Verwendung der Knotensortierung, die sich aus der Rangzuweisung und der Kantenkreuzungsminimierung ergibt, lassen sich initiale Koordinaten für Knoten und Kanten festlegen. Durch iterative Anwendung mehrerer Einzelheuristiken (siehe auch [3]) ergibt sich eine geeignete Koordinatenzuweisung: Ausrichten der Knoten unter Verwendung einer prioritätsgesteuerten Median-Heuristik, alternativ durch einen angepassten spring-embedder Algorithmus, Positionieren der virtuellen Knoten (in Bezug auf die benachbarten Knoten und Kantenketten), Platzieren von Knoten unter Berücksichtigung von Sohn- und Elternknoten, usw. Diese Heuristiken werden angewendet bis das Qualitätsmerkmal, eine gewichtete Summe der Kantenlängen, nicht mehr verbessert werden kann. Dabei werden Verschlechterungen in gewissen Grenzen toleriert. Abschließend werden besondere Korrekturen am Resultat vorgenommen, z.B. Abstände benachbarter Knoten bzw. Ränge minimiert und besondere Kantenkonstellationen bereinigt.
Durch die Verschiebung mehrerer Knoten in Form von Knotenbereichen unter Beachtung des Layouts der Vererbungskanten, die i.d.R. mit höherer Priorität

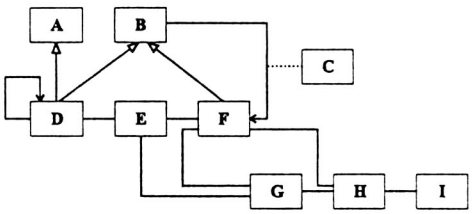


Abbildung 5. Endergebnis für das Beispieldiagramm.

bearbeitet werden, lässt sich für die meisten Assoziationskanten ein orthogonales Layout erreichen. Innerhalb eines Rangs wird ein modifiziertes Manhattan-Layout (einfacher gridline-Algorithmus) eingesetzt. Kanten(ketten), die zu diesem Zeitpunkt nicht orthogonal dargestellt werden können, werden erneut aufgebrochen, und zwischen den Rängen geführt. Die Kreuzungsminimierung dieser Kanten wird bisher heuristisch durch Sortieralgorithmen angenähert, ist aber noch nicht optimal.

3 Das Framework SugiBib

SugiBib wurde in reinem JAVA (JDK 1.2.2) implementiert und garantiert somit Plattformunabhängigkeit. Um eine möglichst frei einsetzbare Grundlage für weitere, zukünftige Anwendungen legen zu können, orientiert sich der Entwurf am Baukastenprinzip. Eine Stufe sei begrifflich zunächst an eine Stufe des Sugiyama-Algorithus angelehnt, kann aber prinzipiell die Implementierung eines Teils eines beliebigen anderen Graph-Algorithmus sein. Daraus ergeben sich fünf Grundprinzipien, die die Struktur des Frameworks im Wesentlichen festlegen:

1. Das Framework implementiert die einzelnen Stufen des Sugiyama-Algorithmus in getrennten Paketen. Zusätzliche Stufen, die sich durch eine Modifikation des Sugiyama-Algorithmus oder Erweiterung durch andere Algorithmen ergeben, werden in geeigneten Paketen realisiert.
2. Das Paket `sugi.admin` ist Grundlage aller Implementierungen einer Stufe. Es enthält z.B. die Basisimplementierung eines Graphen, eines Knotens und einer Kante, die jeweils auf geeigneten Interfaces beruhen. Weitere Klassen, wie Matrizen und Sortieralgorithmen erleichtern und optimieren die Implementierung. Die Graph-Klasse(n) einer Stufe besitzt eine implizite Transformationssemantik: Durch Aufruf des Konstruktors einer Graph-Klasse mit dem Eingabegraphen der jeweiligen Stufe, wird implizit das Resultat der Stufe berechnet.
3. Die Stufen sollen prinzipiell beliebig kombinierbar sein, wobei sich eine sinnvolle Reihenfolge hier durch den Seemann-Algorithmus ergibt. Neue Stufen sollen nahtlos zwischen bereits bestehenden Stufen eingefügt werden können. Die temporären Informationen, die zur Berechnung des Ergebnisses einer

Stufe notwendig sind, sollen in die folgende Stufe nur in dem Maße übernommen werden, in dem die Informationen tatsächlich benötigt werden. Die letzte Stufe, die das Resultat beschreibt, blendet sämtliche, für die Darstellung des Resultats unnötigen Informationen aus. Zudem kann jede Stufe ihre eigenen Graph-, Knoten- und Kantenklassen, ausgehend von den Basisklassen in `sugi.admin`, spezialisieren. Um die Informationsweitergabe und die Erzeugung der korrekten Instanzen in den unterschiedlichen Stufen zu ermöglichen, wird ein erweiterbarer Konversionsalgorithmus bereits im Basispaket `sugi.admin` implementiert. Der Effizienzverlust, der sich durch die Konversion eines Eingabegraphen in einer Stufe ergibt, wird durch die einfache Erweiterbarkeit, die konzeptionelle Trennung der einzelnen Stufen und durch die, in allen Stufen geltende Trennung der Eingabe von der Ausgabe aufgehoben. Die Anwendung der einzelnen Stufen erfolgt durch sukzessive Instanziierung der Graph-Klassen der geeigneten Stufen in der gewünschten Reihenfolge. Diese Reihenfolge wird in einer Algorithmus-Klasse festgelegt und bleibt dem Endanwender i.d.R. verborgen.

4. Da Graphen, Knoten und Kanten unabhängig vom Layout-Prozess, in Abhängigkeit der jeweiligen Anwendung des Frameworks, eigene Informationen tragen können (im Fall von UML-Klassendiagrammen würde diese zusätzliche Information z.B. in einer Knotenklasse die graphische Darstellung einer UML-Klasse beschreiben) und Eingabeobjekte in einen konkreten Algorithmus von den Ausgabeobjekten unabhängig sind, ist es notwendig, diese Information in Form eigener Klassen, den Informationsklassen, auszulagern. Graph-, Knoten- und Kantenklassen tragen prinzipiell eine eigene Informationsklasse, die das entsprechende Interface aus `sugi.admin` implementiert. Eine Informationsklasse besitzt z.B. die Fähigkeiten, ihre graphischen Ausmaße zu berechnen, sich graphisch darzustellen oder sich in einen Datenstrom zu speichern. Zudem können applikationsabhängige Daten in Informationsobjekten abgelegt werden, die während des Layout-Prozesses invariant bleiben.
5. Die Erweiterung des Frameworks erfolgt neben dem Baukastenprinzip durch hot spots wie z.B. Fabrikmethoden, Schablonenmethoden, geeignete Datenklassen und Strategieklassen (siehe auch [2, 4]).

SugiBib besteht aus einer Menge von Kernpaketen, die die allgemeine Implementierung des Sugiama-Algorithmus enthalten. Andere Layout-Algorithmen können dem Framework durch zusätzliche Pakete problemlos hinzugefügt werden. Zudem sind weitere Vorverarbeitungsschritte, die spezielle Sichten eines Eingabegraphen ausfiltern oder Faltungen berechnen, denkbar. Dazu stellt das Framework einen allgemeinen Anfragemechanismus an Knoten und Kanten zu Verfügung, um beliebige Selektionen einfach berechnen zu können.
Die Darstellung von UML-Klassendiagrammen stellt eine Ausprägung des Basis-Frameworks dar, und wurde im Paket `sugi.uml.classDiagrams` implementiert. Der Seemann-Algorithmus ergibt sich durch das Zusammenspiel der folgenden Pakete des Frameworks (siehe auch Abbildung 6):

1. `sugi.preparation` enthält diverse Vorbereitungsschritte, wie z.B. Entfernung von Zyklen, Falten spezifizierter Kanten oder Knoten (graph folding).
2. `sugi.ranking` beinhaltet die Rangzuweisung durch den Netzwerk-Simplex-Algorithmus.
3. `sugi.ordering` approximiert die Minimierung der Kantenkreuzung zwischen benachbarten Rängen nach der Median- bzw. der Barycenter-Methode.
4. In `sugi.uml.classDiagrams.incrementalExtension` wurde die zusätzliche Stufe, die die inkrementelle Layout-Erweiterung nach Seemann durchführt, implementiert.
5. `sugi.coordinates` ermöglicht die Berechnung der Koordinaten von Knoten und Kanten. Initiale Koordinatenzuweisung ist durch die Strategien „Gansner" (prioritätsgesteuerte Medianpositionierung) oder „spring embedder" möglich.
6. `sugi.result` beschreibt das Resultat des Layout-Prozesses und blendet Layout-Informationen, die für das Zeichnen des Resultats unwichtig sind, aus.

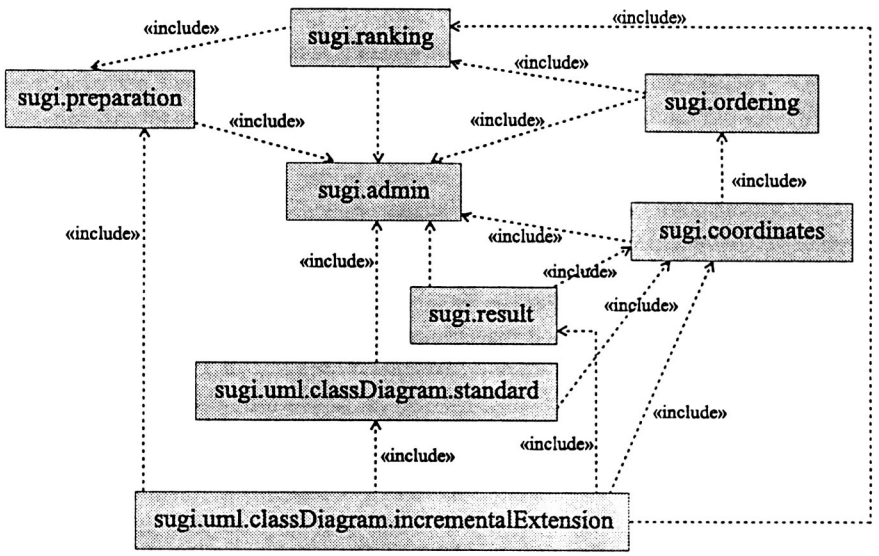


Abbildung 6. Abhängigkeiten der wichtigsten Pakete.

Implementiert wird der Seemann-Algorithmus in der Klasse `SeemannAlgorithmus`, die die Reihenfolge der auszuführenden Schritte festlegt. Konkrete Layout-Algorithmen können, insbesondere zur Laufzeit, konfiguriert oder rekursiv auf sich selbst angewendet werden. Das folgende Fragment zeigt einen Ausschnitt aus der Implementierung der Klasse `SeemannAlgorithmus`; es wird das Layoutergebnis des Eingabegraphen `input` berechnet:

```
CheckVirtualRootGraph vrgraph = new CheckVirtualRootGraph(input);
RankingGraph rgraph = new RankingGraph(vrgraph);
OrderingGraph ograph = new BarycentricOrderingGraph(rgraph,24,10,10);
IncrementalExtensionGraph igraph = new IncrementalExtensionGraph(ograph);
ICoordinatesGraph cgraph = new CoordinatesGraph(igraph);
return new ResultGraph(cgraph);
```

Die unten angegebenen Pakete wurden zur Unterstützung und Gliederung, insbesondere aber zur Ausprägung von SugiBib zum Layout von UML-Klassendiagrammen implementiert:

- `sugi.admin` ist das Paket mit den grundlegenden Klassen.
- `sugi.uml.classDiagrams.standard` stellt die Ausprägung des Frameworks für UML-Klassendiagramme dar. Die bisher einzige Eingabesprache für Klassendiagramme wurde in Form einer Erweiterung von UMLscript [7] durch einen generierten Parser bereitgestellt. Um die Informationen aus dem Parser in das Objektmodell von SugiBib zu übertragen, wurde eine Meta-Klassenhierarchie entworfen und nach Vorbild des Bridge-Musters an die Klassen im Paket `standard` angebunden.

Durch Variation der Darstellungsweise ergibt sich ein Layout, das eine entscheidende Verbesserung des Höhen-Seiten-Verhältnisses (aspect ratio) ermöglicht. Vielfach entstehen, z.B. durch die Beschreibung spezialisierter Klassen in Klassendiagrammen, breite Unterbäume, die durch keine oder nur wenige Assoziationskanten mit den restlichen Klassen verbunden sind. Diese Klassen kann man mit horizontalem oder vertikalem Versatz darstellen. Eine vertikale Ausrichtung verbessert das Höhen-Seiten-Verhältnis erheblich (siehe Abbildung 7). Um dies zu ermöglichen, werden innerhalb der inkrementellen Erweiterung Kandidaten ausgewählt, aus dem Graphen entfernt und in den Eltern-Knoten eingefügt. Knoten werden in SugiBib durch eine Aufteilung in Innenbereich (eigentlicher, sichtbarer Knoten) und Außenbereich (zusätzlich benötigter Platz) realisiert. Da die grafischen und textuellen Kantenattribute, dem Außenbereich zugeordnet werden, lassen sich die Kantenattribute überschneidungsfrei darstellen. Im Fall der Verbesserung des Höhen-Seiten-Verhältnisses liegt es nahe, den Außenbereich in geeigneter Weise zu vergrößern, um das gewünschte Unterlayout darzustellen. Die Technik des Außenbereichs kann insbesondere im Fall reflexiver Assoziationen eingesetzt werden. Die zyklischen Assoziationen, die, wie oben erwähnt, im Vorbereitungsschritt entfernt und in den zugeordneten Knoten transferiert werden, können als Kanten in der Knoteninformationsklasse gespeichert werden und, innerhalb des Außenbereiches eines Knotens, durch einen geeigneten Algorithmus fest einem Unterlayout zugeordnet werden. Analog kann man bei der Darstellung von Schablonenvereinbarungen (Templates) vorgehen.
Im Rahmen der Anpassungen und Erweiterungen der Urversion zeigte sich auch, wie nützlich die Eigenschaft, unregelmässige Knotenformen darzustellen, ist. Der Außen- und Innenbereich eines Knotens ist grundsätzlich rechteckig; die Darstellung innerhalb des Innenbereichs erfolgt durch entsprechende Methoden des Informationsobjekts des Knotens.

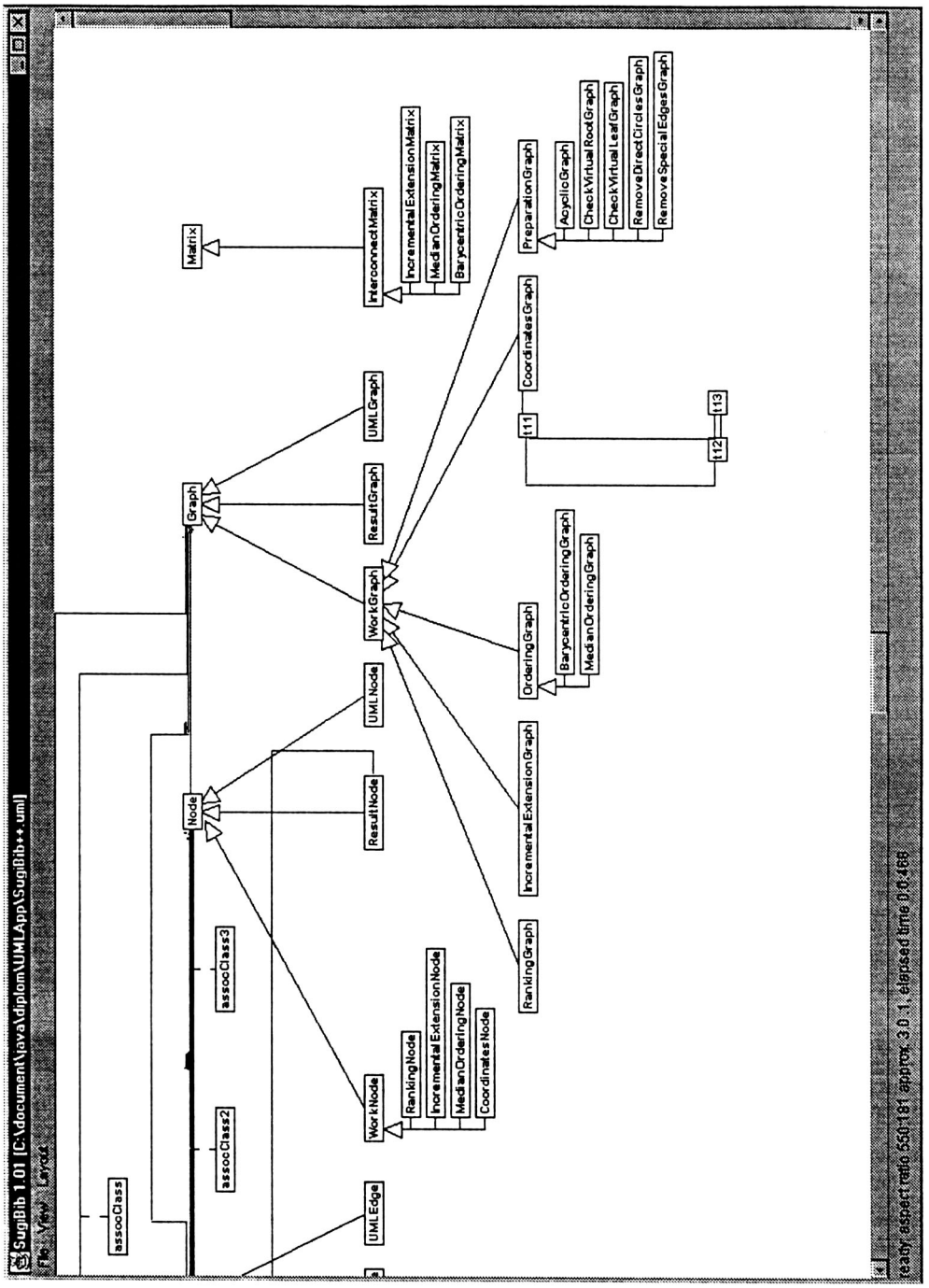


Abbildung 7. Screenshot der Beispielapplikation zur UML-Ausprägung von SugiBib. Dargestellt ist der Ausschnitt eines Teils der Klassenhierarchie von SugiBib und einiger zusätzlicher Diagrammelemente, die zu Testzwecken eingefügt wurden. Am Knoten `WorkNode` sieht man das Knoten-Unterlayout zur Verbesserung des Höhen-Seiten-Verhältnisses.

Dadurch können problemlos z.B. die Zentralknoten n-ärer Assoziationen (Raute) und Entwurfsmuster (unterbrochen gezeichnetes Oval) dargestellt werden. Durch die Kommunikation von Informationsobjekt und den Kanten, die dem Knoten zugeordnet sind, ist eine exakte Positionsbestimmung der Kantenkoordinaten in Abhängigkeit der Knotenform gewährleistet.
Weitere Probleme entstehen durch die Möglichkeit, Assoziationsklassen und Notizen in UML-Klassendiagrammen darzustellen. Assoziationsklassen werden in SugiBib in virtuellen Knoten als darstellbare Information gespeichert. Die Layoutalgorithmen, die in SugiBib verwendet werden, erkennen virtuelle Knoten, die einen Außenbereich besitzen, und erlauben damit die Darstellung von Assoziationsklassen. Die relative Positionierung an einer Kante muss aber unter Verwendung unvollständiger Information innerhalb des Layoutprozesses prädiziert werden. Dadurch entsteht eine Unsicherheit in der Qualität des Ergebnisses, die bisher zu keinerlei praktischen Problemen führte. Die allgemeine Verwendung von Notizen an beliebigen Modellelementen stellt aber weiterhin eine noch zu lösende Herausforderung dar.
Es wurde auch das Layout von Diskriminatoren und ähnlichen textuellen Attributen an Vererbungskanten untersucht. Es stellt sich hierbei ein sehr eingeschränktes ELP (edge label placement) Problem, das durch eine geeignete Sortierung und eine angemessene Vergrößerung des Zwischenraums benachbarter Ränge zufriedenstellend gelöst werden konnte.
Zur Zeit besteht SugiBib mit seiner UML-Ausprägung aus 129 Klassen in 13 Paketen und ca. 66.000 Zeilen Quellcode bei einem Kommentaranteil von 55%.

4 Anwendungen

Die Demonstrations-Applikation besitzt die Fähigkeit, UMLscript-Dateien zu lesen, das Resultat in verschiedenen Vergrößerungsstufen darzustellen und auszudrucken. Ein interaktiver Zoom-Mechanismus, der eine gewählte Klasse in ursprünglicher Größe, benachbarte Klassen etwas verkleinert und die restlichen Klassen eines Diagramms nur durch ihre Kurzform (keine Attribute, keine Methoden) darstellt, und die Kanten zwischen der ausgewählten und den direkt benachbarten Klassen farblich hervorhebt, untermauert die nachträgliche Erweiterbarkeit der Framework-Ausprägung. Eine Option, die Vererbungskanten, die in einem spitzen Winkel gegenüber der Horizontalen dargestellt werden müssten, orthogonal entsprechend der UML-Spezifikation darstellt, rundet die Applikation ab.
Der Sprachumfang von UMLscript, der in der Implementierung von SugiBib zum Einsatz kommt, stellt eine Erweiterung von [7] dar. Die Anwendung von Eigenschaftslisten (tag-values) wurde an den meisten Sprachkonstrukten ermöglicht, die Darstellung von Entwurfsmustern (*design patterns*) und einige Details, wie z.B. Klassen-Compartments zur Unterstützung der UML Version 1.3, hinzugefügt.
Das Framework wurde bisher zur Darstellung von Klassenstrukturen im Rahmen der Übungen zur Vorlesung „Objektorientierte Programmierung“ im Win-

tersemester 1999/2000 eingesetzt. Hierbei zeigte sich eine Besonderheit von UMLscript: Bei der Darstellung der Zusammenhänge typparametrisierter Klassen in C++ (z.B. zwischen einer typparametrisierten Listenklasse und ihren Iteratoren oder bei der Visualisierung der Grundstruktur der *Standard Template Library* STL) ergeben sich Einschränkungen in der Verwendung der Modellelemente: UML verbietet die Vererbung parametrisierter Klassen unter Verweis auf die Verwendung undefinierter Elemente in der wohldefinierten Semantik von UML. UMLscript hingegen erlaubt die freie Vererbung parametrisierter Klassen. Es wurde beschlossen, diese Eigenschaft von UMLscript unter dem Aspekt der anwendungsrelevanten Darstellung von Software-Entwurfsdiagrammen beizubehalten. Hierbei zeigt sich deutlich, dass SugiBib selbst keinerlei Prüfung auf semantische Korrektheit durchführt. Die UML-Ausprägung von SugiBib verlässt sich auf die Eingabesprache und den Autor der Eingabedatei, der die semantische Korrektheit selbst, oder durch ein anderes Tool, z.B. ein CASE-Tool, garantieren muss.
Bisher existierte die UML-Ausprägung nur innerhalb der Grenzen unseres Lehrstuhls als Tool, das ständig weiterentwickelt wird. Um möglichst vielen Interessenten Zugang zur jeweils aktuellen Implementierung zu ermöglichen, gibt es seit Januar 2000 eine WWW-Schnittstelle zur UML-Ausprägung, die als heterogene Lösung unter Verwendung von CGI und Perl realisiert wurde. Ein Einsatz als Servlet oder Java Server Page (JSP) ist ebenfalls denkbar. Unter `http://www2.informatik.uni-wuerzburg.de/SugiBib` finden Interessierte eine kurze Beschreibung des Projekts, die aktuelle Dokumentation des Frameworks, die Beschreibung von UMLscript und den Verweis auf das CGI-Skript. Der Benutzer kann im Skript eine UMLscript-Datei angeben; das Ergebnis wird in Form einer EPS-Datei an die angegebene email-Adresse gesendet. Unter Verwendung eines abgeleiteten, angepassten Graphik-Kontexts, der die Zeichenanweisungen von SugiBib nicht auf dem Bildschirm darstellt, sondern in eine Datei schreibt, und eines virtuellen X-Servers zur Erzeugung des Graphikkontexts, wurde die Server-Version *UMLNet* auf einem Linux-Server in Betrieb genommen.

5 Zukünftige Entwicklungen

SugiBib und die Ausprägung zur Darstellung von UML-Klassendiagrammen stellt kein abgeschlossenes Projekt dar. Das Layout von Klassendiagrammen beinhaltet unvorhersehbare Komplikationen durch interferierende, heuristikbasierte Algorithmen, die nur durch viele Tests erkannt und beseitigt werden können. Daher ist es wünschenswert, dass neben der Anpassung an die jeweils gültige UML-Version, möglichst viele Praxistests durchgeführt werden, um ein stabiles Tool bereitstellen zu können. Erweiterte Funktionalität kann durch eine graphische Benutzeroberfläche, die das Modifizieren von Klassen- und Kanteneigenschaften interaktiv ermöglicht und durch die Integration standardisierter Datenaustauschformate wie z.B. XML bzw. UML XMI DTD oder Tuple-Attribute-Language [8] erreicht werden. Sobald eine Implementierung vorliegt, ist auch an den Einsatz der UML-Ausprägung von SugiBib als Browser-Komponente

zu denken. Große Hoffnung legen wir auf die offenen Schnittstellen der aktuellen CASE-Tools, wie MID Innovator oder Rational Rose. Eine Integration von SugiBib in ein CASE-Tool ist damit möglich.
Die Layoutfähigkeiten könnten in zukünftigen Versionen in Hinblick auf die Höhen-Seiten-Verhältnisse bestehender Druckmöglichkeiten (z.B. DIN A4) optimiert werden. Die Darstellung von Kommentaren an beliebigen Modellelementen stellt weiterhin ein Entwicklungsziel dar. Zudem wäre es außerordentlich wünschenswert, Algorithmen und Verfahren zu entwerfen, die die Unabhängigkeit des Resultats von der Reihenfolge der Klassen des Eingabediagramms garantieren. Bisher fundieren die meisten Initialisierungsroutinen auf der gegebenen Reihenfolge der Elemente des Eingabegraphen. Selbst kleine Veränderungen in der Eingabe können das Ergebnis wesentlich beeinträchtigen.

Literatur

1. H. Eichelberger: Entwicklung eines Frameworks zum automatischen Zeichnen von Software-Entwurfsdiagrammen, Diplomarbeit Universität Würzburg, 1999
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*, Addison Wesley, 1996
3. E.R. Gansner, E. Koutsofious, S.C. North, K.-P. Vo: A Technique for Drawing Directed Graphs, *IEEE Transactions on Software Engineering*, SE-19(3):214-230, March 1993
4. R. Pree: *Komponentenbasierte Softwareentwicklung mit Frameworks*, dpunkt-Verlag, 1997
5. K.G. Kakoulis, I.G. Tollis: On the Edge Label Placement Problem, *Lecture Notes in Computer Science*, LNCS 1190 S. North (Editor), 241-256, 1997
6. *OMG Unified Modeling Language Specification* Version 1.3, June 1999 via `http://www.rational.com`
7. J. Seemann, J. Wolff von Gudenberg: UMLscript: A Programming Language for Object-Oriented Design, *Ein Buch über Unified Modeling Language*, M. Schader, A. Korthaus (Hrsg.), 160-170, 1998
8. J. Seemann, J. Wolff von Gudenberg: Attributierte Graphen zur Metamodellierung mit UML, Vortrag zum 7-ten GROOM-Workshop „UML-Erweiterungen (Profile) und Metamodellierung", 5.4.2000, Koblenz
9. J. Seemann: Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams, *Lecture Notes in Computer Science*, LNCS 1353 G. DiBattista (Editor), 414-423, 1998
10. K. Sugiyama, S. Tagawa, M. Toda: Methods for Visual Understanding of Hierarchical System Structures, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109-125, February 1981

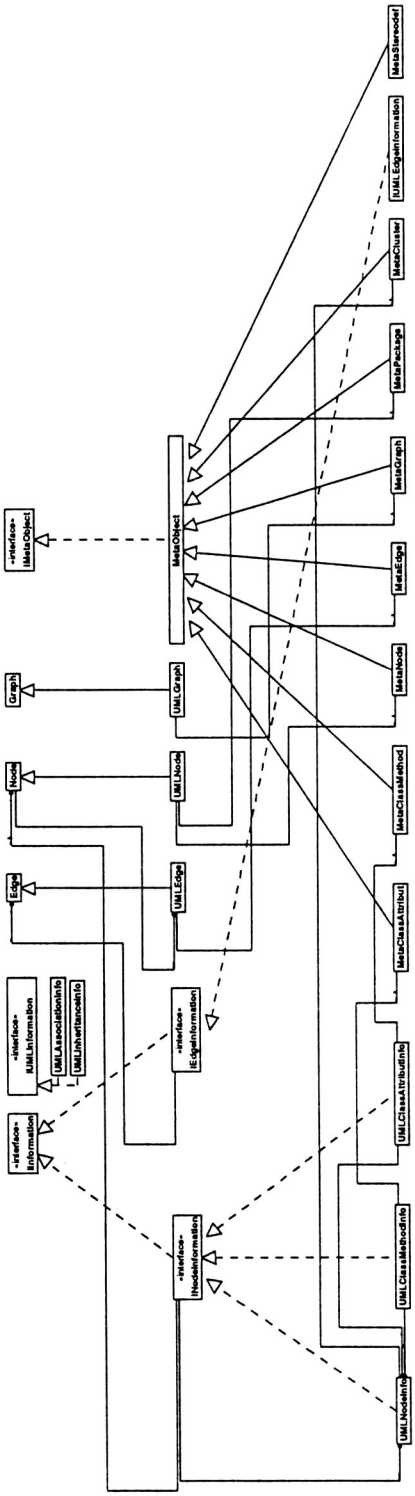


Abbildung 8. Ältere Darstellung der UML-Ausprägung von SugiBib, die orthogonales und direktes Layout der Kanten demonstriert.

Ablaufvisualisierung für nebenläufige Java-Programme mit UML Interaktionsdiagrammen

Katharina Mehner, Annika Wagner

AG Datenbank- und Informationssysteme
Fachbereich Informatik
Universität Paderborn
D-33095 Paderborn

{mehner|awa}@upb.de

Nebenläufige Programmierung ist auch mit modernen Sprachen wie Java, die eigene Sprachkonstrukte für die Synchronisation bieten, komplex. Neben den aus der sequentiellen Programmierung bekannten Fehlern kommt es hier zu Fehlern bei der Sicherheit und der Lebendigkeit der ausgeführten Berechnungen. Diese Fehler sind besonders schwer zu finden, da nebenläufige Programme nichtdeterministisch sind. Während bisherige Testwerkzeuge und formale Analysen mit textuellen Repräsentationen von Programmabläufen arbeiten, denken wir, daß eine Visualisierung das Verständnis und das Auffinden von Fehlern erleichtern kann. Dazu verwenden wir UML, eine weitverbreitete visuelle objektorientierte Modellierungssprache. Wir zeigen wie man Programmabläufe mit UML Sequenz- und Kollaborationsdiagrammen visualisieren kann. Dabei konzentrieren wir uns auf die Darstellung der Synchronisation und erweitern UML, um damit die Laufzeitmechanismen von Java zur Synchronisation darzustellen. Abschließend stellen wir ein Konzept für ein Werkzeug vor.

1 Einleitung

Nebenläufige Programmierung gewinnt zunehmend an Bedeutung, nicht nur in verteilten Anwendungen, sondern auch bei Anwendungen für einzelne Rechner. Dem wird in modernen Programmiersprachen Rechnung getragen, indem sie die Nebenläufigkeit in der Sprache selber unterstützen, statt wie bisher nur durch entsprechende Bibliotheken. Ein wichtiges Beispiel dafür ist die Sprache Java [4], in die abstrakte Konstrukte für die Synchronisation eingegangen sind. Nebenläufige Programmierung bleibt trotzdem schwierig, weil zusätzlich zu den aus der sequentiellen Programmierung bekannten Fehlern, Fehler durch die Nebenläufigkeit hinzukommen. Im wesentlichen unterscheidet man *Sicherheitsprobleme* und *Lebendigkeitsprobleme*, wie zum Beispiel Deadlocks (Verklemmungen) oder Dormancy (d.h. schlafende oder wartende Threads) [8]. Diese Fehler sind besonders schwer zu finden, da nebenläufige Programme nichtdeterministisch sind.

Es existieren eine Reihe von Werkzeugen zum *Testen* [2] und zur *formalen Analyse* [16] nebenläufiger Programme. Hierbei spielen aus der realen Ausführung eines Programmes gewonnene aber auch fiktive Ausschnitte von Programmabläufen (auch

Programm-*Traces* genannt) eine zentrale Rolle. Sie werden oft in textueller Form repräsentiert. Dies ist aber nicht adäquat für komplexe Fehlersituation, wie man sie in nebenläufiger Programmierung findet [10]. Daher betrachten wir die Visualisierung von Programmabläufen. Um Fehler wie Deadlocks oder Dormancy zu verstehen, reicht es aber nicht, Programmabläufe auf Ebene der Sprachkonstrukte von Java zu visualisieren, da die Konstrukte zur Synchronisation sehr abstrakt sind. Um Fehler zu finden, muß man die dahinter verborgenen Laufzeitmechanismen verstehen. Daher ist es unser Ziel, zusammen mit den Programmabläufen die *Laufzeitmechanismen* von Java zur Synchronisation zu *visualisieren.*

Zur Visualisierung verwenden wir die objektorientierte Modellierungssprache *UML (Unified Modeling Language)* [3,14]. Der Vorteil von UML liegt neben der weiten Verbreitung darin, daß man eine Sprache wählt, die bereits im Softwareentwicklungsprozeß eingesetzt wird. Dies erlaubt es, die bei der Visualisierung entstandenen Diagramme mit denen aus der Modellierung zu vergleichen. Darüber hinaus vermeidet man, daß eine weitere neue Sprache erlernt werden muß.

In Abschnitt 2 gehen wir auf Programm-Traces ein. In Abschnitt 3 stellen wir kurz die Synchronisationsmechanismen in Java vor. In Abschnitt 4 stellen wir die Visualisierung mit UML für den wechselseitigen Ausschluß vor. In Abschnitt 5 skizzieren wir eine Werkzeugunterstützung. In Abschnitt 6 betrachten wir verwandte Arbeiten und in Abschnitt 7 geben wir eine Zusammenfassung.

2 Fehlersuche mit Programm-Traces

Man kann auf zwei verschiedene Weisen das Verhalten nebenläufiger Programme untersuchen. *Statische* Analyseverfahren untersuchen den Quelltext, während *dynamische* Analyseverfahren zusätzlich Information aus der Ausführung von Programmen gewinnen. Dazu werden bei der Programmausführung ausgeführte Anweisungen, Methodenaufrufe sowie Änderungen an Variablen und Objekten mitprotokolliert. Ein solches Protokoll nennt man auch einen *Programm-Trace.*

Die Fehlersuche in nebenläufigen Programmen unterscheidet sich wesentlich von sequentiellen Programmen. Zur Fehlersuche in nebenläufigen Programmen muß man nicht nur die Korrektheit von Berechnungsvorschriften prüfen, sondern auch noch verschiedene Abläufe, da Threads (nebenläufige Kontrollflüsse) nichtdeterministisch abgearbeitet werden, wovon aber nicht das Resultat abhängig sein sollte. Zunächst muß man für die Fehlersuche solche verschiedenen Abläufe aus den Programmen gewinnen. Während man bei *nichtdeterministischen* Testen durch mehrfaches Ausführen versucht, voneinander unterschiedliche Programm-Traces zu gewinnen, erzwingt man beim *deterministischen* Testen eine gewünschte Ablaufreihenfolge von Threads [1]. Die so gewonnenen Traces müssen dann auf Fehler untersucht werden. Dazu gibt es verschiedene Hilfmittel wie z.B. *Zeitstempel*[1], um den kausalen Zusammenhang zwischen verschiedenen Threads zu erfassen oder *Race Analysis*, um herauszufinden, wo es zwischen Threads möglicherweise *Race Conditions* gibt [13].

Interessanterweise spielen Programm-Traces nicht nur in der dynamischen Analyse eine Rolle, sondern auch in der statischen. Zur formalen Analyse von Quellcode werden unter anderem *Modelchecker* eingesetzt [16]. Für nebenläufige Programme kann

ein solcher Modelchecker z.B. Programm-Traces berechnen, die zu einem Deadlock führen.

Die Darstellung zeigt, daß es ausgereifte Techniken gibt, um Programm-Traces zu erzeugen. Aufgrund der Komplexität der aufzufindenden Fehler stellt sich die Frage nach geeigneten Visualisierungtechniken. Für die Fehlersuche werden unter Umständen auch verschiedene Sichtweisen auf die erzeugten Programm-Traces gebraucht. In [2] wird z.B. mit Programm-Traces gearbeitet, welche die für die Synchronisation relevanten Ereignisse herausfiltern, allerdings auch nur in textueller Form. Wir wollen für Programm-Traces insbesondere die Laufzeitmechanismen für Sprachen mit abstrakten Synchronisationmechansimen wie Java visualisieren. Bevor wir darauf näher eingehen, geben wir zunächst einen Überblick über das Modell der Nebenläufigkeit in Java.

3 Nebenläufige Programmierung in Java

Die Sprache Java unterstützt nebenläufige Programmierung mit *Threads* in der Sprache selber [4,8]. Ein Thread ist ein sequentieller Kontrollfluß innerhalb eines Programmes. In Java sind Threads zugleich Objekte und die Klassenbibliothek bietet entsprechende Methoden darauf an.

Verschiedene Threads innerhalb eines Programmes können die gleichen Objekte verwenden, um ihre Funktionalität zu implementieren. Aber auch die Kommunikation zwischen Threads geschieht über gemeinsame Objekte (sogenannte *shared objects*). Um Inkonsistenzen gemeinsam genutzter Objekte zu vermeiden, muß der Zugriff verschiedener Threads synchronisiert werden. Darüberhinaus müssen Threads koordiniert werden, um Informationen austauschen zu können, d.h. sie müssen manchmal aufeinander warten.

Der Schutz von Daten vor simultanen Zugriffen und die Koordination von Zugriffen geschieht klassischer Weise mit einem sogenannten *Monitor*. Ein Monitor verwaltet zum einen den Zugriff auf zu schützende Daten, indem er nur maximal einen Thread auf diesen Daten arbeiten läßt, auch bekannt als Konzept des *Wechselseitigen Ausschlusses*. Der Monitor unterstützt zum anderen die Koordination zwischen Threads, indem er Mechanismen anbietet, mit denen Threads auf das Eintreffen bestimmter Bedingungen warten können, bzw. Mechanismen, mit denen Threads signalisieren können, daß sich Bedingungen verändert haben. Wartende Threads werden in einer Warteschlange verwalten.

Monitore sind als Sprachkonzept in Java eingegangen. Dabei ist jedem Objekt ein Monitor zugeordnet. Mit dem Schlüsselwort `synchronized` kann ein Anweisungsblock oder eine gesamte Methode gekennzeichnet werden, die nie von mehr als einem Thread gleichzeitig ausgeführt werden kann. Mit den Methoden `wait`, `notify` und `notifyAll` kann die Warteschlange des Monitors manipuliert werden.

Mit `synchronized` wird immer ein gesamtes Objekt für alle anderen Methoden oder Blöcke, die ebenfalls als `synchronized` deklariert sind, gesperrt. Bei einer als `synchronized` deklarierten Methode ist dies implizit das Objekt auf dem die Methode aufgerufen wird. Im Falle eines Anweisungsblockes muß das Objekt bei der Deklaration von `synchronized` angeben werden.

Nur ein Aufruf einer dieser Methoden oder Anweisungblöcke erhält den *Lock* (dt. Sperre) für das Objekt, um damit arbeiten zu können. Aufrufe andere Threads werden solange blockiert. Ist der erste Aufruf abgearbeitet, erhält einer der blockierten Aufrufe in nichtdeterministischer Auswahl als nächster den Lock und kann abgearbeitet werden.

Durch die Blockierung beim Warten auf einen Lock ergibt sich die Gefahr für einen *Deadlock*. Im einfachsten Fall kann es zu einem Deadlock kommen, wenn zwei Threads je ein Objekt gelockt haben und auf die Freigabe des jeweils anderen Locks warten. Die notwendige Voraussetzung für einen Deadlock ist somit, daß es einen Zyklus in der Beziehung zwischen Objekten gibt, die von verschiedenen Threads genutzt werden. Deadlocks sind schwer zu finden, da sie von einer konkreten Aufrufreihenfolge in verschiedenen Threads abhängen, die an den Stellen, an denen die Threads dann nicht mehr weiterarbeiten können, nicht nachvollziehbar ist. Daher ist der Kontext und die Aufrufgeschichte nötig, um einen Deadlock überhaupt beschreiben zu können. Deadlocks treten aufgrund des Nichtdeterminismus bei der Abarbeitung von Threads nicht zwingend auf, selbst wenn das Programm die Möglichkeit eines Deadlocks birgt.

Die Methoden `wait` und `notify(All)` können nur innerhalb eines `synchronized`-Bereiches verwendet werden. Bei Aufruf von `wait` wird der Thread, der das Objekt gerade benutzt in die Warteschlange eingereiht, dabei geht er in den Zustand wartend über. Aus dieser Warteschlange können Threads nur entfernt werden, wenn andere Threads auf demselben Objekt `notify(All)` aufrufen. Mit `notify` wird nur ein Thread aus der Warteschlange entfernt, während mit `notifyAll` alle Threads entfernt werden. Entfernte Threads werden wieder aktiviert. `notify` kann nur von aktiven Threads aufgerufen werden. Gibt es in einem Programm nur noch wartende Threads und keine aktiven mehr, können die wartenden nie wieder aktiviert werden. Dieses Problem ist auch bekannt als *Dormancy*, d.h. für immer schlafende Threads. Ein lebendiges Programm setzt also eine Aufeinanderabstimmung von `wait`- und `notify`-Methoden voraus.

4 Visualisierung von wechselseitigem Ausschluß

In diesem Abschnitt betrachten wir die Visualisierung des wechselseitigen Ausschlusses, der in Java mittels `synchronized` programmiert wird. Wir verwenden UML-Interaktionsdiagramme (Sequenz- und Kollaborationsdiagramme), mit denen Abläufe auf Instanzebene modelliert werden können [3,14]. In diesem Kontext verwenden wir keine Klassendiagramme, da wir annehmen, daß diese bereits aus der Modellierungsphase vorhanden sind. Wir werden sehen, daß beim Verständnis von Synchronisation verschiedene Aspekte eine Rolle spielen, die sich mit verschiedenen UML-Interaktionsdiagrammen darstellen lassen. Wir setzen voraus, daß dem Leser die wesentlichen Bestandteile von Sequenz- und Kollaborationsdiagrammen bekannt sind.

Wir verwenden den von UML vorgesehenen *Erweiterungmechanismus* des *Stereotyps*, um die UML-Diagramme an Java anzupassen. Mittels Stereotypen lassen sich neue Sprachelemente von gegebenen UML-Modellierungselementen ableiten,

um UML an einen spezifischen Anwendungsbereich anzupassen. Der Stereotyp wird dabei entweder in doppelten spitzen Klammern <<...>> in Verbindung mit dem ursprünglichen Sprachelement angegeben oder durch ein neues graphisches Element dargestellt. So kann z.B. das Modellierungselement Klasse zu einer Javaklasse spezialisiert werden durch Kennzeichnung mit dem Stereotyp <<Java>>.

Sequenzdiagramme

Um die Synchronisation von Threads zu verstehen, spielt die genaue zeitliche Reihenfolge von Aufrufen eine besondere Rolle, weil davon abhängt, zu welchem Ergebnis der Ablauf eines nebenläufigen Programmes führt. In UML gibt es nur eine Diagrammart, mit der man Abläufe in ihrer *zeitlichen* Reihenfolge visualisieren kann: Sequenzdiagramme.

Als Beispiel betrachten wir eine einfache Klasse `cell` (vgl. Abbildung 1). Die Klasse `cell` hat einen Inhalt `value` und besitzt eine Methode `swap()`, um den Inhalt mit einer anderen Zelle zu tauschen. Der Tauschvorgang muß natürlich atomar sein, d.h. es dürfen keine anderen Objekte in dieser Zeit die Zelleninhalte lesen oder ändern. Daher werden wie in einem 2-Phasen-Protokoll zuerst via `synchronized` die benötigten Locks angefordert, nämlich auf beide beteiligten Zellobjekte. Danach werden die Inhalte getauscht.

```
class cell {
      private int value;
      public void swap(cell other){
           synchronized(this){
             synchronized(other){
                int newValue = other.value;
                other.value = value;
                value = newValue;}  }
      }
...}
```

Abbildung 1 Java-Klasse `cell`

Die einfachste denkbare nebenläufige Situation mit Objekten dieser Klasse ist, daß zwei Zellen unabhängig von einander von zwei verschiedenen Threads aufgefordert werden, ihre Inhalte zu tauschen. Zwei verschiedene Abläufe, die sich aus dieser Situation ergeben können, sind in Abbildung 2 und 3 als Sequenzdiagramme dargestellt. Beide Abläufe beruhen aber auf dem gleichen Javaprogramm, daß wir hier aus Platzgründen nicht zeigen. In den Sequenzdiagrammen sind die beiden Threads als aktive Objekte `client1` und `client2` dargestellt.

Problematisch bei dieser Visualisierung ist die Repräsentation von `synchronized`. Hierbei handelt es sich um ein Sprachkonstrukt von Java, was im eigentlichen Sinne keinem Methodenaufruf entspricht. Ohne Repräsentation von `synchronized` verliert das Sequenzdiagramm jedoch jegliche Aussagekraft. Die Frage, wie ein `synchronized`–Block am besten zu repräsentieren ist, läßt sich beantworten,

indem man sich mit seiner Semantik befaßt: zu Beginn des Blocks soll ein Lock auf das Parameterobjekt angefordert werden. Wir repräsentieren das Schlüsselwort `synchronized` daher als expliziten synchronen Methodenaufruf auf dem Parameterobjekt. Der Aufruf muß synchron erfolgen, weil das aufrufende Objekt auf den Erhalt des Locks warten muß. Am Ende des `synchronized`-Block wird der Lock durch einen synchronen Aufruf wieder zurückgegeben.

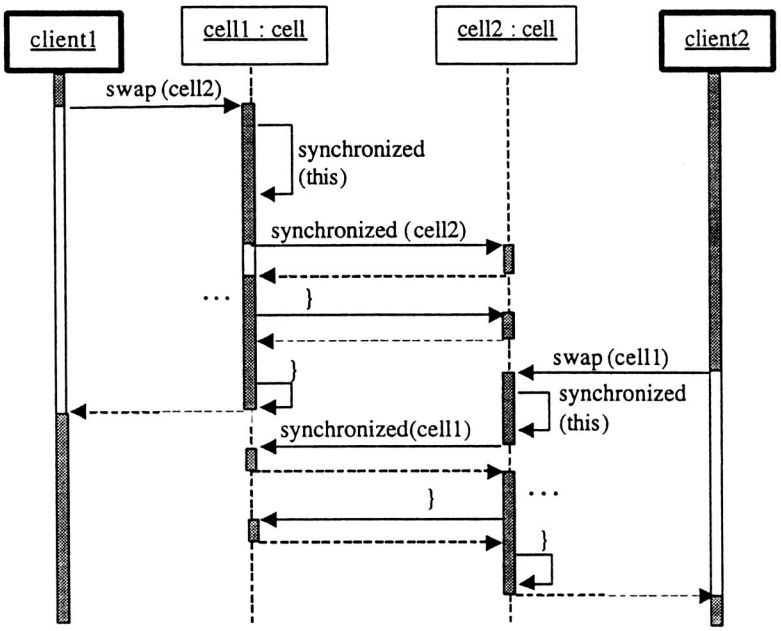


Abbildung 2 Sequenzdiagramm für `swap`

Während das erste Sequenzdiagramm einen Ablauf zeigt, bei dem beide Zellen ihre Inhalte erfolgreich austauschen, kommt es im zweiten Sequenzdiagramm zu einem Deadlock. Betrachten wir zunächst das Diagramm aus Abbildung 2. Hier tauscht `cell1` zuerst ihren Inhalt mit `cell2`. Dazu fordert sie die nötigen Locks an, die sie auch erhält. Der eigentliche Austausch der Inhalte ist an der durch `...` markierten Stelle ausgelassen. Anschließend gibt `cell1` ihre Locks wieder frei. Dann wird `cell2` zum Austausch der Inhalte aufgefordert. Dieser Austausch verläuft genauso problemlos. Man beachte, daß die Schraffur einer Aktivierung angibt, an welcher Stelle ein aktiviertes Objekt tatsächlich etwas berechnet. Eine nicht schraffierte Aktivierung gibt an, daß die Aktivierung warten muß. (Diese Notation wurde in UML 1.3 neu eingeführt.)

Hierin liegt auch schon der Unterschied zum zweiten Sequenzdiagramm in Abbildung 3. An der Tatsache, daß irgendwann alle Aktivierungen nicht mehr schraffiert sind, erkennt man, daß ein Deadlock eingetreten ist. Weiterhin zeigt das Diagramm auch noch die Aufrufreihenfolge, die zu diesem Deadlock geführt hat.

Während `cell1` versucht, ihren Inhalt mit `cell2` zu tauschen, versucht `cell2` den Tausch bereits in umgekehrter Richtung. Was jedoch nach wie vor nicht ersicht-

lich ist, ist der Grund, *warum* diese Aufrufreihenfolge zu einem Deadlock führt. Dieser Grund liegt in der Semantik von `synchronized`, das einen Lock auf das Empfängerobjekt anfordert. Hat bereits ein anderes Objekt diesen Lock, muß der Sender von `synchronized` warten. Die Tatsache, daß ein anderes Objekt diesen Lock besitzt, ist eine besondere Art von Beziehung zwischen dem Objekt, welches den Lock hält, und dem gelockten Objekt.

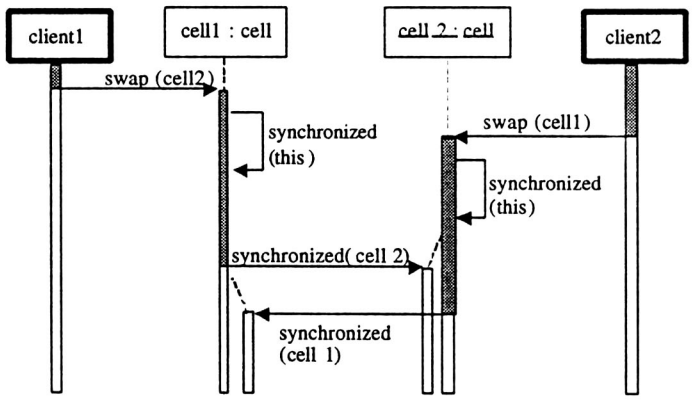


Abbildung 3 Sequenzdiagramm mit Deadlock

Beziehungen jeglicher Art sind im Sequenzdiagramm jedoch nicht sichtbar. Da Beziehungen jedoch in Kollaborationsdiagrammen dargestellt werden können, liegt es nahe, diese zu verwenden, um in Ergänzung zum Sequenzdiagramm auch noch den Grund des Deadlocks analysieren zu können.

Kollaborationsdiagramme

In diesem Abschnitt zeigen wir, wie man mit Hilfe geeigneter Erweiterungen von Kollaborationsdiagrammen die Gründe näher analysieren kann, die zu einer Deadlock-Situation geführt haben.

Dazu betrachten wir zunächst wieder den Ablauf, wie er im Sequenzdiagramm in Abbildung 3 dargestellt ist, und übersetzen ihn in ein Kollaborationsdiagramm (vgl. Abbildung 4). Wie schon im Sequenzdiagramm werden die beiden Threads als aktive Objekte dargestellt. Wir zeigen die gleichen Methodenaufrufe und `synchronized`-Anweisungen wie zuvor. Die zeitliche Anordnung des Sequenzdiagramms wird in Sequenznummern übersetzt. Nebenläufige Threads werden durch unterschiedliche Buchstaben gekennzeichnet, hier mit `A` und `B`. Unteraufrufe werden durch geschachtelte Numerierungen gekennzeichnet. Mit den Sequenznummern kann man jedoch nur die zeitliche Anordnung innerhalb eines einzigen Threads zeigen, nicht jedoch die globale zeitliche Anordnung wie vorher. Diese Information geht bei der Übersetzung verloren, weil Kollaborationsdiagramme nicht zeitliche Bezüge zwischen verschiedenen Threads darstellen können.

Durch die Parameterübergabe entstehen zwei neue Links vom Stereotyp `<<parameter>>`. Im Aufruf `swap(cell2)` von `client1` auf `cell1` werden die Aufru-

fe `synchronized(this)` und `synchronized(cell2)` abgearbeitet (analog für `client2` und `cell2`). Bei der Repräsentation von `synchronized` verfahren wir genauso wie im Sequenzdiagramm. Man beachte, daß somit dem Kollaborationsdiagramm aus Abbildung 4 beide Sequenzdiagramme aus Abbildung 2 und 3 zugrunde liegen könnten. Daher kann es auch nicht den Deadlock zeigen, was eigentlich unsere Absicht war.

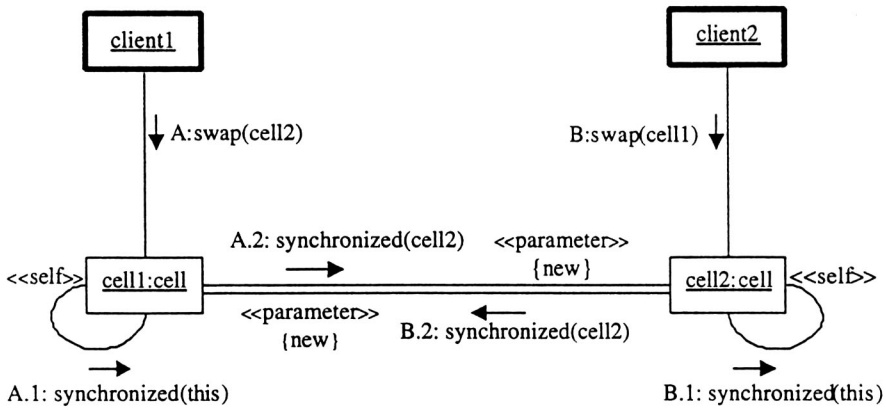


Abbildung 4 Kollaborationsdiagramm

Stereotypisierte Beziehungen im Kollaborationsdiagramm

Der eigentliche Zweck unseres Überganges vom Sequenzdiagramm zum Kollaborationsdiagramm war die Repräsentation der internen Vorgänge bei einer `synchronized`-Anweisung. Der Besitz eines Locks stellt eine gerichtete Beziehung zwischen dem Objekt, das den Lock hält; und dem gelockten Objekt dar. Dabei handelt es sich jedoch um keinen normalen Link. Unserer Beziehung liegt keine Assoziation im Klassendiagramm zugrunde und sie ist auch nicht navigierbar. Daher führen wir einen *Stereotyp* `<<locks>>` ein. Analog verhält es sich mit der Situation, in der ein Objekt einen Lock anfordert, ihn jedoch nicht erhält und deshalb warten muß. Hierfür sehen wir den *Stereotyp* `<<acquires>>` vor.

Repräsentieren wir nun die Situationen aus den Sequenzdiagrammen in Abbildung 2 und 3 als Kollaborationsdiagramm mit diesen Erweiterungen, so unterscheiden sich die entstehenden Kollaborationsdiagramme. Im Falle des Deadlocks sieht das Kollaborationsdiagramm wie in Abbildung 5 dargestellt aus. Sowohl die beiden Instanzen des Stereotyps `<<locks>>` als auch die des Stereotyps `<<acquires>>` sind neu zugefügt und mit dem Schlüsselwort `{new}` markiert. Im Falle eines Kollaborationsdiagrammes für das linke Sequenzdiagramm aus Abbildung 2 wären nur die beiden Locks vorhanden und diese wären `{transient}`, weil sie auch wieder freigegeben werden.

Mit der Einführung der Stereotypen `<<locks>>` und `<<acquires>>` sind wir unserem Ziel, die Gründe eines Deadlocks zu visualisieren, näher gekommen. Erkennbar wird, daß alle Threads blockiert sind. Beide Threads haben als letzte Anwei-

sung mittels `synchronized` einen Lock angefordert, auf den sie warten müssen. Verfolgt man die Links der Stereotypen `<<locks>>` und `<<acquires>>`, so kann man sehen, daß die Threads sich sogar gegenseitig blockieren, da sie jeweils einen Lock anfordern, welchen der andere hält. Genau das macht den Deadlock aus.

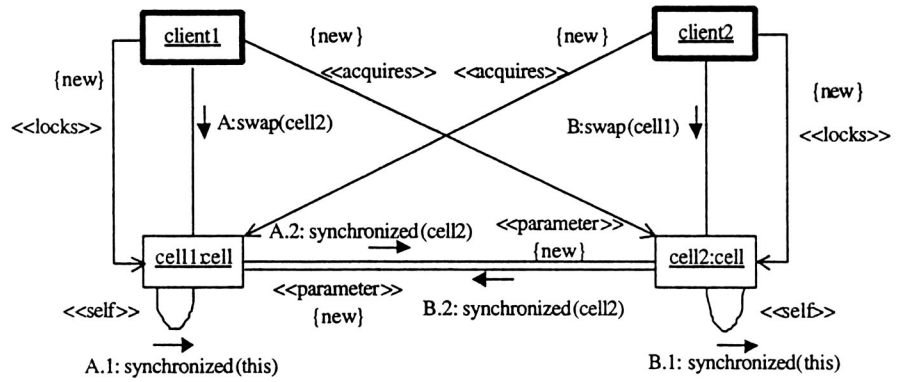


Abbildung 5 Kollaborationsdiagramm mit stereotypisierten Links

Unsere bisherigen Erweiterungen haben also dazu beigetragen, daß man die für den Deadlock verantwortlichen Locks bestimmen kann. Um den Deadlock beheben zu können, muß man jedoch die angeforderten Locks und das Warten auf Locks auch den entsprechenden Anweisungen zuordnen können. Bei diesem Beispiel ist das sehr einfach, weil sich die `synchronized` Anweisungen direkt eine Ebene unter dem `swap()` Aufruf befinden. Somit ist die Zuordnung von den `<<locks>>` und `<<acquires>>` Links problemlos durchzuführen. Wenn sich im Rumpf einer Methode zwei synchronized-Blöcke hintereinander befinden, ist die Zuordnung nicht mehr möglich.

Die Situation wird noch wesentlich komplizierter, wenn eine solche `synchronized`-Anweisung sich irgendwo in einer Folge von geschachtelten Methodenaufrufen (alle auf dem gleichen Objekt) befindet. Dann wird man zur Beschreibung von Synchronisation nicht unbedingt alle diese Methodenaufrufe ins Kollaborationsdiagramm aufnehmen, sondern vielleicht nur die `synchronized`-Anweisung oder die umgebenden Methode darstellen. D.h. man will nicht nur wie im Beispiel die Semantik einer einzelnen `synchronized`-Anweisung visualisieren, sondern auch die Möglichkeit nutzen, von einem ganz Block von Anweisungen und ineinander geschachtelten Methodenaufrufen nur die Semantik hinsichtlich der Sychronisation zu abstrahieren. Wie man auch dann noch die Entstehung des Locks der `synchronized`-Anweisung bzw. der umgebenden Methode zuordnen kann, erläutern wir im nächsten Abschnitt.

Numerierung neu erzeugter stereotypisierter Links

Im vorherigen Abschnitt haben wir motiviert, daß die Zuordnung der `<<locks>>`- und `<<acquires>>`-Links zu den sie erzeugenden Anweisungen bzw. Methoden wichtig ist, um die Stellen im Code ausfindig zu machen, die ursäch-

lich für den Deadlock sind. Diese Zuordnung können wir vornehmen, indem wir entstehende Links analog zu Unteraufrufen mit in die Numerierung einbeziehen. In Abbildung 6 haben wir die Links der Stereotypen `<<locks>>` und `<<acquires>>` numeriert.

Normalerweise wird in Kollaborationsdiagrammen bewußt auf die Numerierung dieser Links verzichtet. Der Grund dafür ist, daß von der konkreten Reihenfolge ihrer Erzeugung abstrahiert werden soll, was beim Einsatz von Kollaborationsdiagrammen in der Modellierungsphase Sinn macht. Diese Abstraktion ist aber in unserem Zusammenhang unerwünscht, da wir konkrete Programmabläufe visualisieren wollen.

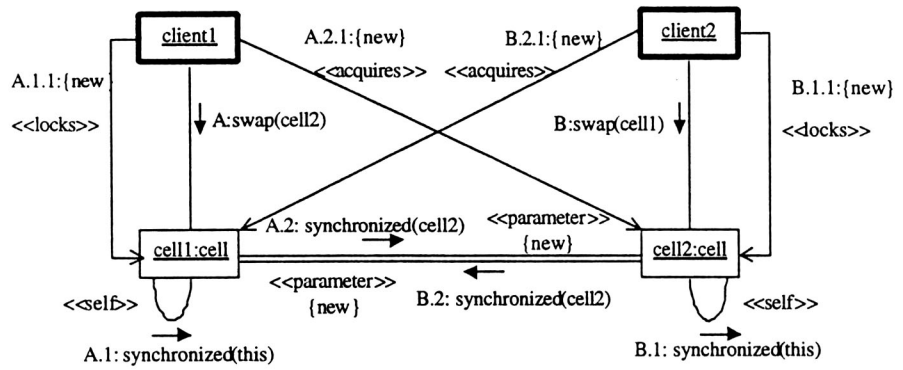


Abbildung 6 Kollaborationsdiagramm mit numerierten Links

Zeitweilig aktive und zeitweilig mehrfach aktive Objekte

Bisher haben bei der Visualisierung der Synchronisation die aktiven Objekte eine herausragende Stellung eingenommen. Ihnen wurden die Locks zugeordnet und sie sind es, die auf Locks warten. Dies resultiert aus der Abbildung der entsprechenden Laufzeitmechanismen von Java. In einer konkreten Deadlocksituation müssen die aktiven Objekte aber nicht immer von Interesse sein. Im Gegenteil, ihre Berücksichtigung kann die zu visualisierende Situation unnötig komplex machen. In diesem Abschnitt zeigen wir, wie man diese Komplexität reduzieren kann, indem man das Kollaborationsdiagramm auf den eigentlich relevanten Ausschnitt beschränkt. Dazu ersetzen wir das Vorkommen aktiver Objekte durch den neu zu definierenden *Stereotyp* der *zeitweilig (mehrfach) aktiven Objekte*.

Was aktive von passiven Objekten unterscheidet, ist der eigene *Kontrollfokus*, über den ein aktives Objekt verfügt. Im Laufe der Ausführung eines Programmes bewegt sich dieser Kontrollfokus bei jedem Methodenaufruf vom aufrufenden Objekt zum aufgerufenen. Das aufgerufene Objekt wird also zu einem *zeitweilig aktiven Objekt*, da es ja zu diesem Zeitpunkt über einen Kontrollfokus verfügt. Der kleinste relevante Ausschnitt eines Kollaborations-diagrammes zur Visualisierung synchronisierter Abläufe wird stets das aufgerufene Objekt enthalten. Zur Repräsentation des Threads genügt es daher, dieses aufgerufene Objekt mit Hilfe eines *Stereotyps* zu einem zeitweilig aktiven Objekt zu machen. Als graphische Darstellung dieses Stereotyps wäh-

len wir die dicke Umrandung des aktiven Objektes, jedoch in gestrichelter Form als Zeichen der Zeitweiligkeit.

Damit man sieht, welche Objekte von dem Thread gelockt sind, biegen wir die `<<locks>>` Links so um, daß sie jetzt von dem zeitweilig aktiven Objekt auf die gelockten Objekte zeigen. Ebenso hängen wir jetzt die `<<acquires>>` Links an die zeitweilig aktiven Objekte (siehe Abbildung 7).

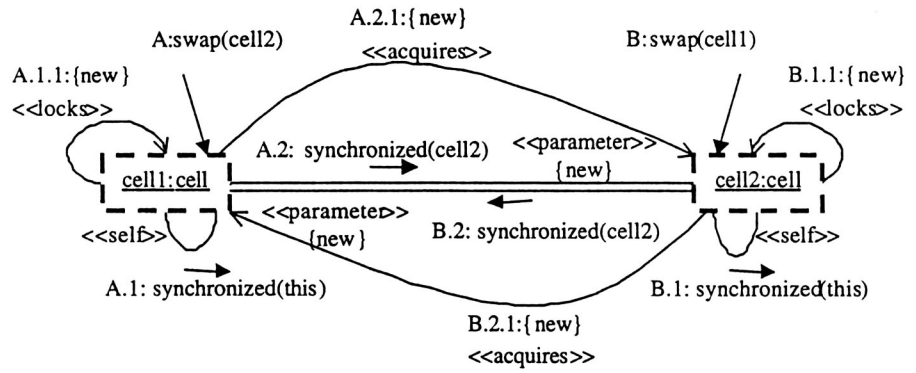


Abbildung 7 Kollaborationsdiagramme mit zeitweilig aktiven Objekten

Da alle passiven Objekte in einem Kollaborationsdiagramm zeitweilig aktiv sind, markieren wir nur Objekt als zeitweilig aktiv, in dem sich der Kontrollfokus zuletzt befindet. An dieses Objekt werden dann auch die `<<locks>>` Links gehängt.

In einem nebenläufigen Kontext können sich die Kontrollfoki mehrerer Threads in einem Objekt befinden, sofern nicht alle Methoden als `sychronized` deklariert sind. Diese können dann gleichzeitig verschiedene oder sogar dieselbe Methode ausführen. Ein Objekt, das über mehrere Kontrollfoki verfügt, nennen wir *zeitweilig mehrfach aktives Objekt.* Dafür führen wir einen Stereotyp mit einer graphischen Darstellung ein. Ein solches Objekt erhält dieselbe gestrichelte Umrandung wie das zeitweilig aktive Objekt, mit dem Unterschied, daß für jeden weiteren Kontrollfokus, über den das Objekt verfügt, ein weiterer gestrichelter Rahmen versetzt hinter dem Objekt gezeichnet wird. Bei mehrfach aktiven Objekten hängen wir die während der Synchronisation entstehenden Links wie `<<locks>>` und `<<acquires>>` nicht alle an die Umrandung des Objektes, sondern nur die für den ersten Kontrollfokus. Für alle weiteren Kontrollfoki werden die jeweils unter einem Fokus bei der Synchronisation entstandenen Links an den jeweiligen versetzten Rahmen gehängt. Aus Platzgründen verzichten wir auf ein Beispiel und verweisen den Leser auf [9].

Zur Demonstration unserer Erweiterungen haben wir hier ein sehr einfaches Beispiel verwendet. Die Gefahr für einen Deadlock bei der Klasse `cell` läßt sich für einen erfahrenen Programmierer leicht erkennen. Der Deadlock kann vermieden werden, indem man eine Reihenfolge für angeforderte Ressourcen festlegt [8]. In realistischeren Beispielen sind Deadlocks jedoch nicht auf eine einzige Klasse oder sogar eine einzige Methode beschränkt, sondern entstehen im Zusammenspiel verschiedener Methoden und Klassen. Gerade dann ist es wichtig, eine visuelle Repäsentation der beteiligten Objekte und Aufrufe zu haben.

5 Visualisierung von Thread-Koordination

Zur Visualisierung von Programmabläufen, in denen `wait-` und `notify-`Methoden aufgerufen werden, können ebenfalls Sequenzdiagramme eingesetzt werden. Fehler wie schlafende Threads erkennt man ähnlich wie bei Deadlocksituationen daran, daß irgendwann kein Thread mehr im Sequenzdiagramm aktiv ist. Aus Platzgründen können wir hier keine Beispiele angeben.

Allerdings unterscheidet sich die Analyse von Dormancy-Probleme von Deadlocks. In Deadlocksituationen benötigen wir exakte Informationen über die Beziehungen, da zyklische Beziehungen der Grund für einen Deadlock sein können. Im Falle fehlender `notify-` Aufrufe läßt aber nicht eindeutig sagen, warum diese Aufrufe fehlen oder wo sie hingehören. Diese Probleme erfordern eine umfassendere Analyse und können auch nicht basierend auf einem einzeln Programm-Trace erfolgen. Die Analyse muß sich damit befassen, welche Bedingungen zum Warten von Threads führen, wann sich Bedingungen ändern, und wer über die Änderungen von Bedingungen benachrichtigen kann und wer benachrichtigt werden muß. Dazu benötigt man die vorher erwähnte *statische Analyse*, wie z.B. das Programm Slicing (siehe Abschnitt 7). Zur Visualisierung reichen dann ebenfalls nicht instanzbezogene UML-Diagramme aus wie die Interaktionsdiagramme. Daher beabsichtigen wir, zur Beschreibung des Verhaltens auf einer höheren Abstraktionsebene UML-Statecharts einzusetzen.

6 Werkzeugunterstützung

Während für einfache Beispiele, wie hier vorgestellt, die UML-Diagramme von Hand erstellt werden können, ist dies für komplexere Beispiel nicht möglich, sondern muß automatisiert werden. Im Zentrum der von uns angedachten Werkzeugunterstützung soll jedoch nicht die Erzeugung von Programm-Traces liegen, sondern eine *Visualisierungskomponente*, deren Hauptfunktionalität darin bestehen soll, aus Programm-Traces UML-Diagramme zu erzeugen und diese um die Darstellung der Synchronisation anzureichern.

Diese Komponente sollte daher eine offene Schnittstelle für die Weiterverarbeitung von Traces aus existierenden Tools haben. Denkbar wäre, hier XML als Datenaustauschformat zugrunde zu legen. Existierende Tools erzeugen bisher aber noch keine XML-Formate, so daß zusätzlich Transformatoren erstellt werden müssen, um proprietäre Formate nach XML zu übersetzen.

Die Darstellung der entstehenden UML-Diagramme sollte mit einem existierenden UML-Casetool erzeugt werden wie z.B. Rational Rose [11]. Die darin enthaltenen Editoren unterstützen auch selbstdefinierte Stereotypen. UML-Diagramme können in Rational Rose über eine XML-Schnittstelle eingelesen werden. Demnach sollen die von der Visualisierungkomponente erzeugten UML-Diagramme auf dieser Schnittstellendefinition basieren.

7 Verwandte Arbeiten

Unser Ziel, Programmabläufe zu visualisieren, um Fehler finden zu können, steht in Bezug zum sogenannten *Debugging*. Während gewöhnliche Debugger dazu dienen, den Zustand eines Programms anzuzeigen oder fortgeschrittenere Werkzeuge auch Programm-Traces in textueller Form erzeugen können, liegt der Vorteil von unserem Ansatz darin, daß wir eine graphische Sprache zur Visualisierung von Traces verwenden. Darüber hinaus setzen wir für die verschiedenen Aspekte, unter denen man Traces bei der Fehlersuche betrachten will, verschiedene Diagramme ein. Während Sequenzdiagramme besser geeignet sind zeitliche Zusammenhänge darzustellen, können Kollaborationsdiagramme besser strukturelle Beziehungen darstellen. Außerdem gehen wir über gewöhnliche Ansätze des Debuggings hinaus, da wir die Laufzeitmechanismen darstellen. [2] beschäftigt sich speziell mit der Darstellung der Synchronisation in Programm-Traces, allerdings auch nur in textueller Form. Hier werden die für die Synchronisations relevanten Ereignisse aus Programm-Traces herausgefiltert. In [7] wird das Debugging paralleler (und damit nebenläufiger) Programme durch Eventgraphen unterstützt. Dieser Ansatz stellt gegenüber unserem schon einen weiteren Abstraktionsschritt dar, da er die beteiligten Objekte nicht mehr visualisiert.

Unser Ansatz ist auch verwandt mit dem Gebiet des *Programmverstehens. Statisches* Programmverstehen arbeitet auf dem Quelltext. Eine Technik dafür ist beispielweise das *Program Slicing* [5]. Die Grundidee besteht darin, in einem Programm alle Anweisungen zu bestimmen, die den Wert einer zuvor bestimmten Variablen ändern. In [15] wird eine Erweiterung bestehender Slicing-Techniken unter besonderer Berücksichtigung der Nebenläufigkeit in Java-Programmen vorgestellt. *Dynamisches* Programmverstehen verwendet zusätzlich Informationen aus Programm-Traces. Darunter gibt es bereits Ansätze, die graphische Sprachen verwenden, wie z.B. Message Sequence Charts in dem Werkzeug Jinsight [5]. Hier werden jedoch keine Laufzeitmechanismen visualisiert.

Beim dynamischen *Reverse Engineering* verwendet man die Information aus verschiedenen Programm-Traces dazu, um generelle Verhaltensweisen des Programmes zu inferieren, z. B. in Form von State Charts [12]. Die bisherigen Ansätze beschäftigen sich jedoch noch nicht explizit mit Synchronisation. Außerdem liegt der Fokus darauf, lauffähige Programme zu verstehen, nicht jedoch darauf, sich mit Fehlersituationen zu beschäftigen.

8 Zusammenfassung und Ausblick

Wir haben einen Ansatz vorgestellt, der Entwickler von nebenläufigen Programmen bei der Fehlersuche durch visuelle Repräsentation von Programmabläufen unterstützen soll. Dazu verwenden wir UML, eine weitverbreitete Sprache zur objektorientierten Modellierung. Zusätzlich zur normalen Verwendung von Programm-Traces visualisieren wir die Laufzeitmechanismen von Java bei der Synchronisation. Wir setzen dafür verschiedene Diagramme ein, um verschiedene Aspekte zu unterstützen. Während Sequenzdiagramme zur Visualisierung des zeitlichen Ablaufes geeignet sind, ergänzen Kollaborationsdiagramme die Dimension struktureller Beziehungen.

Zur Abbildung der Laufzeitbeziehungen wurden neue Stereotypen eingeführt. Während man mit diesen zwei Diagrammarten Probleme wie Deadlocks gut erkennen kann, muß noch eine geeignete Unterstützung für die Darstellung von Dormancy - Problemen gefunden werden.

Wir haben skizziert, wie wir uns eine Werkzeugunterstützung vorstellen. Dabei ist es wichtig, die Visualisierungkomponente sowohl von einem konkreten UML-Tool unabhängig zu machen, wie auch von Tools zur Generierung der Programm-Traces. Hierzu existieren bereits ausgereifte Werkzeuge, die auf das Testen oder das Modelchecking von nebenläufigen Programmen spezialisiert sind. Unser Schwerpunkt soll auch in Zukunft darauf liegen, zu untersuchen, wie diese Werkzeuge an UML anzubinden sind und inwieweit die Visualisierung mit UML die Analyse von Programm-Traces verbessern kann.

Literatur

[1] A. Bechini, K. Tai, Timestamps for Programs Using Messages and Shared Variables, *Proc. of the 18th International Conference on Distributed Computing Systems*, 1998.

[2] A. Bechini, K.Tai, Design of a Toolset for Dynamic Analysis of Concurrent Java Programs, *Proc. of the 6th International Workshop on Program Comprehension IWPC'98*. 1998.

[3] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Reading, MA. Addison-Wesley, 1998.

[4] J. Gosling, B. Joy, G. Steel, *The Java Language Specification*, Addison-Wesley, 1996.

[5] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12, 1 (January 1990), 26-60.

[6] Jinsight (http://www.research.ibm.com/jinsight), IBM Research 1998.

[7] D. Kranzlmüller, S. Grabner, J. Volkert, Debugging with the MAD Environment, in J. Dongarra, B. Tourancheau (Eds.), *Environments and Tools for Parallel Scientific Computing*, Vol23, Nos. 1-2, April 1997.

[8] D. Lea, *Concurrent Programming in Java*, Addison-Wesley, 1997.

[9] K. Mehner, A. Wagner, Visualisierung der Synchronisation von Java-Threads mit UML, *Workshop Modellierung*, 2000.

[10] M. Pancake, Visualization Techniques for Parallel Debugging and Performance-Tuning Tools, in A.Y. Zomaya, *Parallel Computing: Paradigms and Applications*, Intl. Thomson Computer Press, 1996.

[11] Rational Rose 98, (http://www.rational.com), Rational Inc. 1998.

[12] T. Systa, On the Relationship between Static and Dynamic Models in Reverse Engineering Java Software, *Proc. of the 6th Working Conference on Reverse Engineering WCRE'99*, Atlanta, 1999.

[13] K. Tai, Race Analysis of Traces of Asynchronous Message-Passing Programs, *Proc. of the 17th International Conference on Distributed Computing Systems*, 1997.

[14] UML Specification, Version 1.3. The Object Management Group, Doc. Ad/99-06-08. June 1999.

[15] J. Zhao, Slicing Concurrent Java Programs, *Proc. of the 7th International Workshop on Program Comprehension IWPC'99*, 1999.

[16] J. Zhao, K. Tai, Deadlock Analysis of Synchronous Message-Passing Programs, *International Symposium on Software Engineering for Parallel and Distributed Systems PDSE'99*, Los Angeles 1999.

Datenstrukturen visualisieren und animieren mit DDD

Andreas Zeller

Universität Passau
Lehrstuhl für Software-Systeme
Innstraße 33, 94032 Passau
zeller@acm.org

Zusammenfassung Der graphische Debugger *DDD* ist mit mehr als 250.000 Anwendern eins der verbreitetsten Werkzeuge zur Softwarevisualisierung. Neben den üblichen Debugger-Funktionalitäten ermöglicht DDD die *Visualisierung von Datenstrukturen* im laufenden Programm. DDD kann verzeigerte Strukturen (wie Listen oder Bäume) als Graphen darstellen, aber auch numerische Felder in zwei- oder dreidimensionalen Plots darstellen und im Programmlauf animieren. Dieser Beitrag beschreibt die technischen Grundlagen der Visualisierung, die verwendeten Layoutverfahren, sowie die Animation von Algorithmen.
Schlüsselwörter: Visuelles Debuggen, Visualisierung der Daten und Abläufe in Anwendungen, Visualisierung von Datenstrukturen, Algorithmenanimation.

1 Einführung: Visuelles Debuggen

Trotz aller Fortschritte in den frühen Phasen der Softwareentwicklung gehört die *Fehlersuche* weiterhin zum täglichen Brot der Programmierer. *Interaktive Debugger* haben sich als Universal-Werkzeuge zur Fehlersuche etabliert. Mit einem Debugger kann man

- das Programm in einer definierten Umgebung *ausführen*,
- das Programm unter bestimmten Bedingungen *anhalten lassen*,
- den Zustand des angehaltenen Programms *untersuchen* und
- den Zustand des angehaltenen Programms *verändern*.

Wenn es um die Untersuchung des Programmzustands geht, ist der zentrale Aspekt die *Datenausgabe*. Traditionelle Debugger geben ihre Daten als *Text* aus – so etwa der GNU-Debugger GDB:

```
(gdb) print *tree
*tree = {value = 7, _name = 0x8049e88 "Ada", _left = 0x804d7d8,
  _right = 0x0, left_thread = false, right_thread = false,
  date = {day_of_week = Thu, day = 1, month = 1, year = 1970,
   _vptr. = 0x8049f78 (Date virtual table)}, static shared = 4711}
(gdb) _
```

Abb. 1. Textuelle Datenausgabe mit GDB

Auch wenn moderne Debugger die Gesamtausgabe nicht mehr als Fließtext gestalten, werden doch die Daten nach wie vor als *Text* ausgegeben. Für einzelne Daten ist diese textuelle Darstellung gewiß angemessen – wie anders wollte man eine Zeichenkette

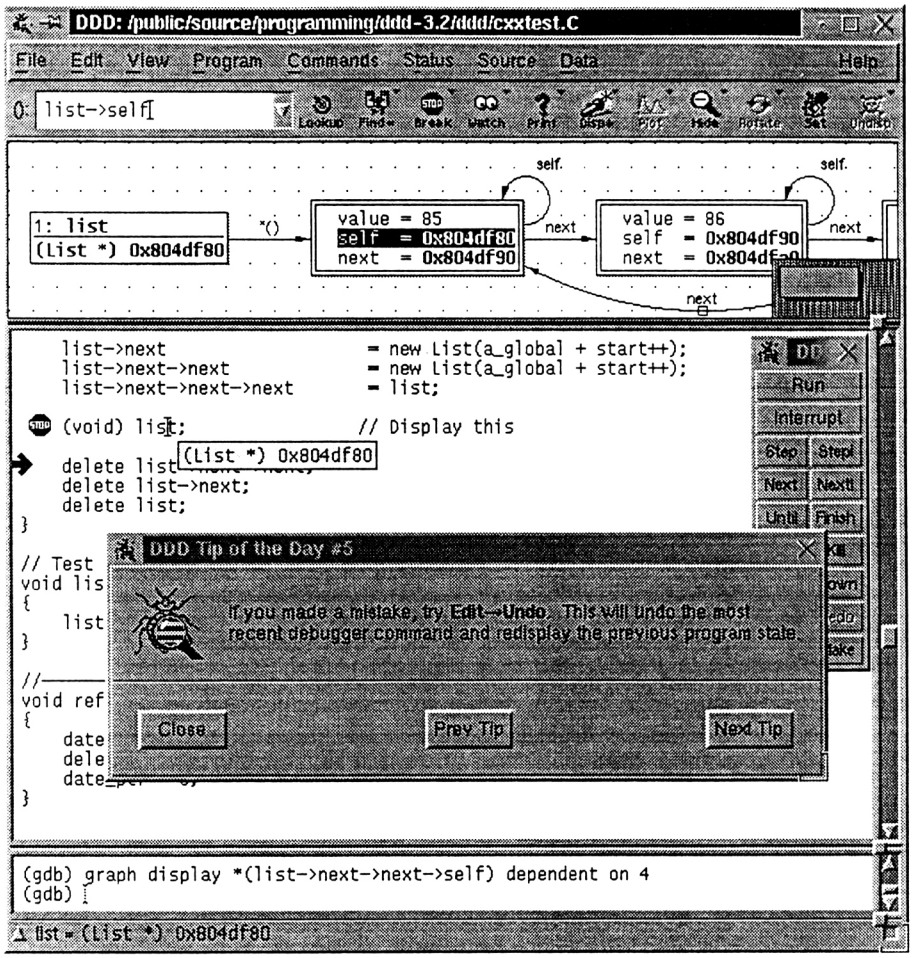


Abb. 2. Darstellung von Datenstrukturen in DDD

schon betrachten? Schwierig wird es aber, wenn es um *Beziehungen* zwischen Daten geht – Referenzen, Zeiger, Indizes und dergleichen. Worauf etwa zeigt `tree->_left`? Natürlich kann der Anwender dies individuell abfragen. Wenn aber zwei Zeiger denselben Wert haben, kann der Anwender dies nur durch mühsames Vergleichen der Adressen bestimmen.

Getreu dem Motto, daß „der Mensch bildliche Eindrücke besonders effizient verarbeiten kann und konkrete Objekte besser erfassen kann als abstrakte Beschreibungen“ [2], haben wir einen Debugger konstruiert, der über *eingebaute Datenvisualisierung* verfügt – den *Data Display Debugger* oder kurz DDD.

DDD ist technisch realisiert als *Front-End* zu herkömmlichen Kommandozeilen-Debuggern. DDD unterstützt derzeit GDB, DBX, LADEBUG, XDB, JDB, den Python-Debugger und den Perl-Debugger; die Oberfläche (Abbildung 2) bietet allen gewohn-

ten Komfort einschließlich einer detaillierten Zustands- und Kontext-sensitiven Hilfe. DDD ist als offizielle GNU-Software im Quelltext frei erhältlich und wird heute von über 250.000 Anwendern zur Programmentwicklung eingesetzt; die Liste der Anwender reicht von Adobe und Boeing bis zu Texas Instruments und Xerox.

Im folgenden zeigen wir, wie sich DDD als Universalwerkzeug zur dynamischen Softwarevisualisierung einsetzen läßt. Abschnitt 2 beschreibt zunächst die Visualisierung von Datenstrukturen aus dem laufenden Programm; die Layoutregeln werden in Abschnitt 3 gesondert erläutert. Abschnitt 4 diskutiert die alternative Darstellung numerischer Werte als Plots. Als Debugger kann DDD dargestellte Daten während des Programmlaufs auf den neuesten Stand bringen; Abschnitt 5 zeigt, wie sich dies zur Algorithmenanimation einsetzen läßt. Verwandte Arbeiten sind in Abschnitt 6 behandelt; Abschnitt 7 schließt mit Zusammenfassung und Ausblick.

2 Visualisierung von Datenstrukturen

Grundidee der Visualisierung in DDD ist, jedem individuellen Datum einen eigenen Bereich, eine sogenannte *Box*, zuzuweisen. Ineinander geschachtelte Strukturen werden durch ineinander geschachtelte Boxen dargestellt, die zum Ausblenden durch Platzhalter ersetzt werden können. Ein solches komplettes Einzel-Datum wird *Display* genannt.

Die Stärke von DDD liegt im Visualisieren der *Beziehungen* zwischen Displays. Hierfür werden die einzelnen Displays durch *gerichtete Kanten* zu einem Graph zusammengefaßt. Wird ein neues Datum d' aus einem bestehenden Datum d angezeigt – etwa durch *Dereferenzieren* eines Zeigers in d –, so stellt DDD die Beziehung durch eine Kante von d nach d' dar. Auf diese Weise lassen sich komplexe Datenstrukturen darstellen (Abbildung 3).

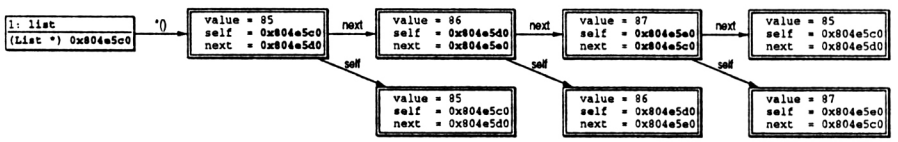


Abb. 3. Darstellung verketteter Listen ohne Aliaserkennung

Normalerweise zeigt auf jedes neu erschaffene Datum maximal eine Kante – die Darstellung ist also auf Baumstrukturen beschränkt. Durch Aktivieren einer besonderen *Aliaserkennung* kann DDD jedoch auch komplexere Datenstrukturen darstellen. Grundidee ist, die Displays, die an derselben Adresse im Speicher stehen (*Aliase*), zu *verschmelzen*, um so mehrere Verweise auf ein Datum zu ermöglichen.

Als Beispiel dienen erneut die Daten von Abbildung 3. Das Vergleichen der Zeiger-Werte macht deutlich, daß die `self`-Zeiger jeweils auf die Displays selbst zeigen; außerdem verweist der letzte `next`-Zeiger wieder auf das erste Element der Liste. Nach Einschalten der Aliaserkennung erfragt DDD für alle angezeigten Daten deren Speicheradressen und verschmilzt die Boxen mit gleicher Adresse. Das Ergebnis ist in Abbildung 4 auf der nächsten Seite zu sehen – das Bild zeigt deutlich, welche Zeiger auf

welche Daten verweisen, ohne daß der Anwender erst deren Adressen textuell vergleichen müßte.

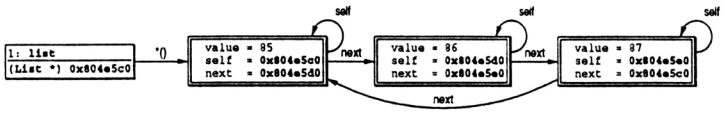


Abb. 4. Darstellung verketteter Listen mit Aliaserkennung

Die Aliaserkennung erlaubt das Darstellen auch komplexerer Datenstrukturen wie im Lehrbuch. Abbildung 5 zeigt, wie sich ein gefädelter Baum in DDD präsentiert.

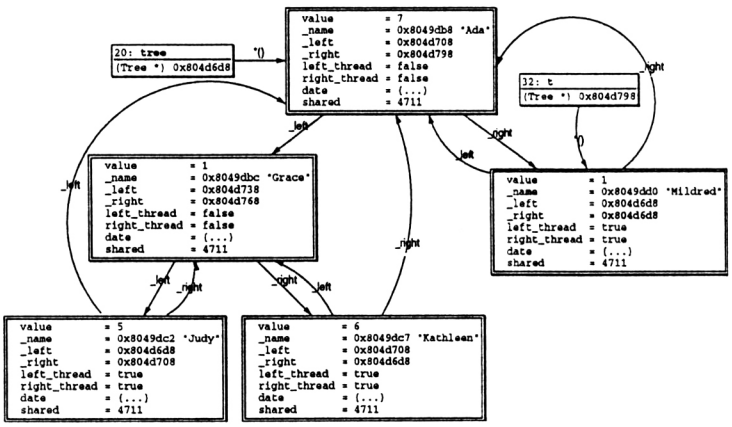


Abb. 5. Ein gefädelter Baum in DDD

Neben dem einfachen Dereferenzieren von Zeigern unterstützt DDD beliebige andere Verweis-Operationen. Erzeugt der Benutzer etwa einen Tabelleneintrag $t[i]$ aus einem Index i, so wird eine Kante von i nach $t[i]$ gezogen. Die Operationen werden in einem Menü gespeichert und dienen so zur Erkundung und Darstellung beliebiger Verweise.

3 Layoutalgorithmen

In DDD erkundet der Programmierer Datenstrukturen *interaktiv* und *inkrementell* – er entscheidet selbst, welchen Querverweisen er folgen möchte. Damit entfällt die Notwendigkeit, eine Datenstruktur am Stück zu erkennen und zu layouten. DDD kennt lediglich vier einfache Regeln, anhand der neue Displays positioniert werden, dargestellt in Abbildung 6 auf der nächsten Seite.

Ausgangslage ist, daß der Anwender ein neues Display d' aus einem bestehenden Display d erzeugt – etwa, indem er d dereferenziert und d' erhält. Dann lassen sich vier Fälle unterscheiden:

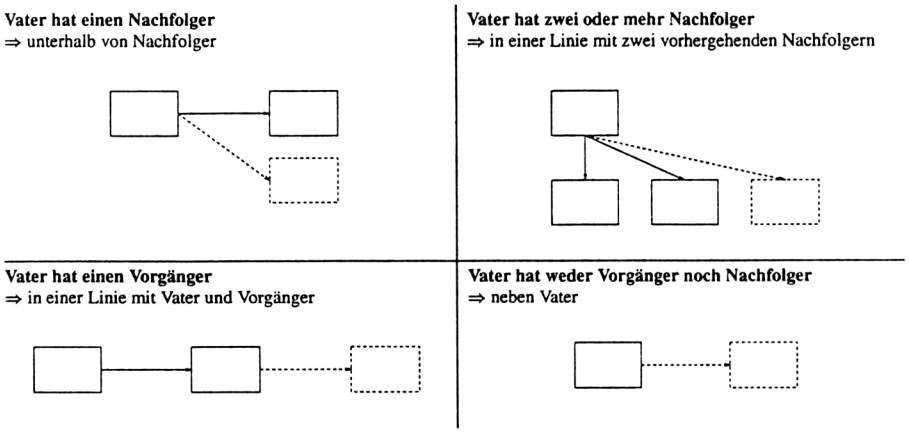


Abb. 6. Positionierungsregeln in DDD. Das neu zu positionierende Display ist gestrichelt.

- Hat d (der „Vater") bereits einen Nachfolger, so wird d' neben d plaziert. Dies ermöglicht das „Auffächern" mehrerer Nachfolger.
- Hat d mehr als einen Nachfolger, wird d' in Verlängerung des Abstands zwischen den beiden letzten Nachfolgern plaziert. Dies führt dazu, daß alle Nachfolger aneinander ausgerichtet sind.
- Hat d keinen Nachfolger, aber einen Vorgänger, wird d' in Verlängerung des Abstands zwischen d und dessen Vorgänger plaziert. Hiermit werden verkettete Strukturen ausgerichtet.
- Hat d weder Vorgänger noch Nachfolger, wird d' neben d plaziert. Die Vorgabe für diese Ausrichtung ist „rechts von d"; der Benutzer kann dies durch Rotieren des Graphen ändern.

Nach der ursprünglichen Positions-Vorgabe kann der Anwender das Layout nach Belieben manuell verändern. DDD bietet auch die Möglichkeit, den kompletten Graphen am Stück neu zu layouten. Hierfür setzt DDD das Verfahren von Sugiyama und Misue [8] ein. In jedem Layoutvorgang geht die bisherige Ortsinformation verloren – jedes Display erhält einen neuen Platz. Daher muß sich der Anwender komplett neu orientieren, was belastend wirkt. Unserer Erfahrung nach bevorzugen die Benutzer das manuell-inkrementelle Layout.

Nichtdestotrotz bleibt auch das inkrementelle Layout mühsam, wenn große Strukturen Stück für Stück erkundet werden müssen. Für künftige DDD-Versionen ist deshalb eine *automatische Entfaltung* von Datenstrukturen geplant: Nach dem Positionieren der ersten zwei oder mehr Displays werden die Operationen solange auf die Nachfolger angewandt, bis ein Fixpunkt erreicht ist. Auf diese Weise lassen sich beliebige Datenstrukturen durch einfaches automatisches Wiederholen der Zugriffsmechanismen entfalten.

Die konkrete Darstellung der Daten und Beziehungen in DDD hat den Nachteil, daß wenig Raum für abstrakte (Text-)Daten bleibt. Neuere DDD-Versionen bieten deshalb die Möglichkeit, Displays in *Cluster* zusammenzufassen. Werden grundsätzlich alle Displays geclustert, erhält man eine herkömmliche platzsparende Textdarstellung wie auch in traditionellen Debuggern.

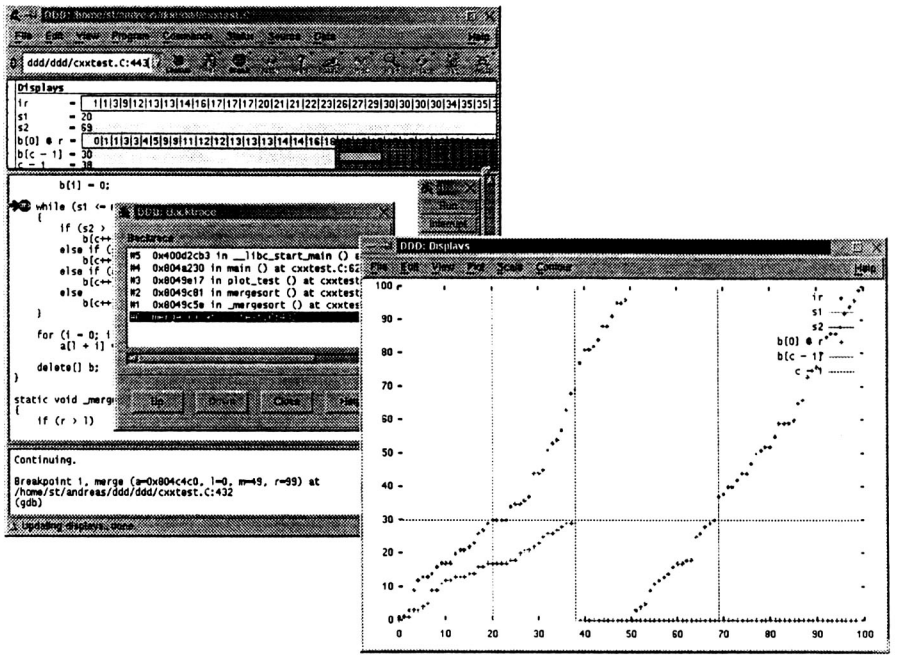


Abb. 7. Plots in DDD

4 Visualisierung numerischer Werte

Neben der Darstellung von Datenstrukturen ermöglicht es DDD, numerische Werte in Form von *Plots* darzustellen. Hierfür greift DDD auf die Dienste des externen Werkzeugs GNUPLOT zurück. Das Grundverfahren ist ganz einfach:

1. Der Benutzer wählt ein Datenfeld aus, das geplottet werden soll.
2. DDD erfragt vom Kommandozeilen-Debugger die Werte und sendet sie an GNUPLOT.
3. DDD führt die von GNUPLOT generierten Zeichenanweisungen in einem eigenen Fenster aus.

Abbildung 7 zeigt einen so generierten Plot. Insgesamt kann DDD folgende Datentypen als Plot darstellen:

- Eindimensionale Felder werden als 2D-Plot über den Index gezeichnet.
- Zweidimensionale Felder werden als 3D-Plot über beide Indizes gezeichnet.
- Einfache numerische Werte werden als Linien gezeichnet. Wird der Wert zusammen mit einem Feld gezeichnet und liegt der Wert im Indexbereich des Feldes, so zeichnet DDD den Wert als vertikale x-Konstante, ansonsten als horizontale y-Konstante. Der Benutzer kann die Ausrichtung selbst ändern.
- Zusammengesetzte Werte (Cluster, Records, usw.) werden in einem Plot zusammengefaßt. DDD berücksichtigt hierbei alle auftretenden numerischen Werte – Zeichenketten oder Zeiger werden übergangen.

Der letzte Punkt ist besonders wichtig, wenn man mehrere Werte zueinander in Beziehung setzen will. In Abbildung 7 etwa sehen wir einen Ausschnitt aus der *merge*-Funktion des *mergesort*-Algorithmus. Die äußeren vertikalen Linien sind die Indizes über die beiden Hälften, aus denen gemischt wird; die mittleren Linien zeigen die obere Grenze des bereits gemischten Feldes. DDD stellt Menüs zur Verfügung, mit denen die wichtigsten Aspekte des Plots verändert werden können; die Plots können auch gedruckt und an andere Programme exportiert werden.

Als besondere Eigenschaft erlaubt es DDD, die *Historie* eines Wertes zu plotten – also die Kette der früheren Werte während der Programmausführung. Bei jedem Programmhalt (etwa beim schrittweisen Ausführen) merkt sich DDD den dargestellten Programmzustand einschließlich aller Variablenwerte in einem *Historien-Speicher;* der Anwender kann durch *Undo* und *Redo* nach Belieben frühere Programmzustände erneut darstellen. Dies ist besonders praktisch, wenn man frühere Werte von Variablen noch einmal sehen möchte. Beim Plotten der Historie wird nun die Folge der Variablenwerte über die Anzahl der Programmhalte geplottet. Hiermit lassen sich ungewöhnliche Wertentwicklungen „auf einen Blick" erkennen.

5 Algorithmenanimation

Bei jedem Halt des Programms bringt DDD alle angezeigten Werte auf den neuesten Stand. Dies gilt für die dargestellten Datenstrukturen und Plots; auch die Alias-Analyse wird bei jedem Programmhalt neu gestartet.

Neben dem eigentlichen Ziel des Debuggens läßt sich diese Eigenschaft besonders gut nutzen, um *Algorithmen zu animieren* – ganz ohne zusätzliches Werkzeug, Hilfsfunktionen oder auch nur eine Neuübersetzung. Abbildung 8 etwa zeigt das Löschen von Elementen aus einer verketteten Liste. Nach jedem Löschen stoppt ein Haltepunkt das Programm, und DDD stellt den neuen Wert der Liste dar.

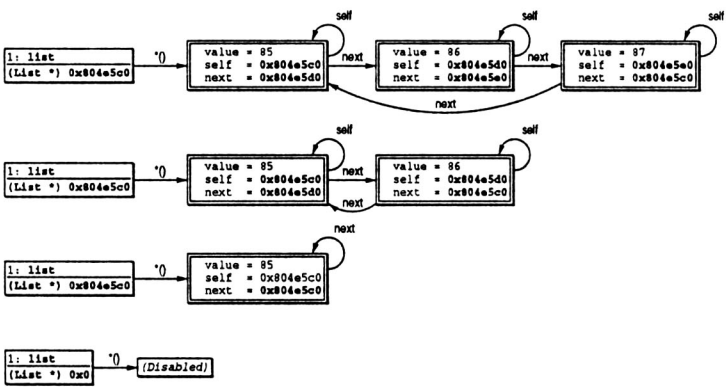


Abb. 8. Elemente aus einer verketteten Liste löschen[1]

[1] Für eine optimale Darstellung in diesem Beitrag zeigen wir statt DDD-Screenshots von DDD exportierte Grafiken. Alle Bilder können ebenso interaktiv am Bildschirm betrachtet werden.

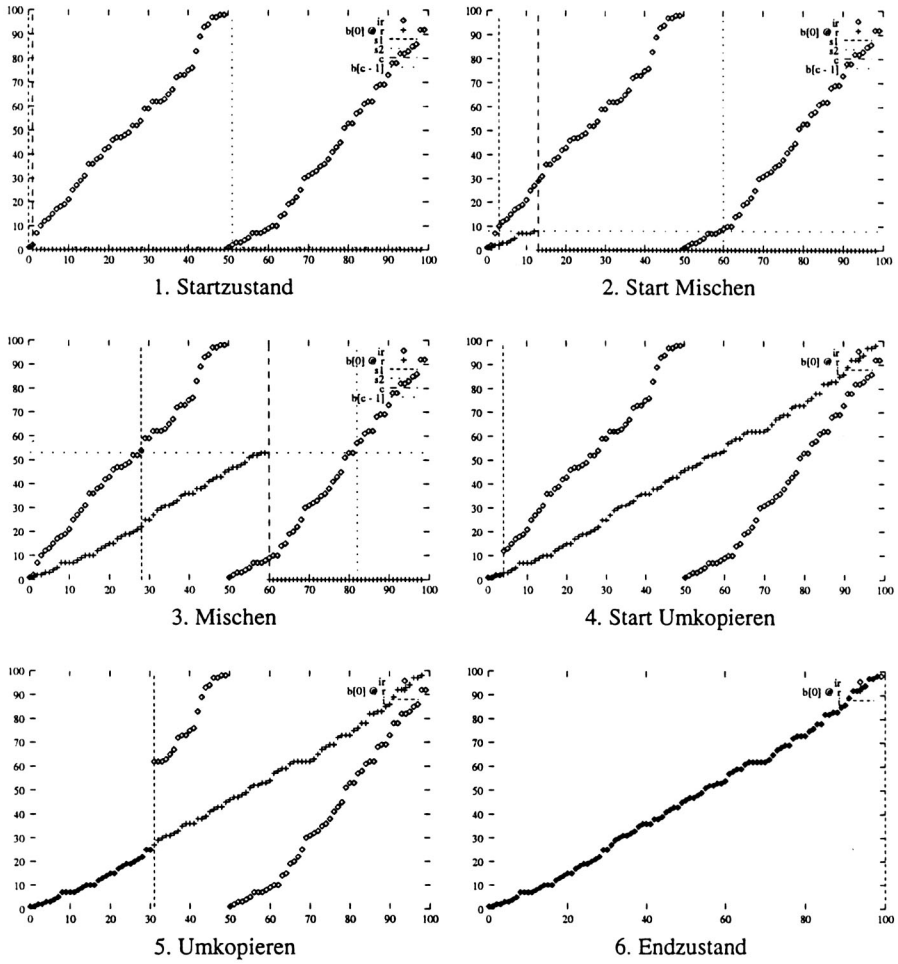


Abb. 9. Animation eines *mergesort*-Algorithmus in DDD

Will man in der Lehre einen Algorithmus zeigen, ist insbesondere die *Undo/Redo*-Funktion nützlich. Durch *Undo/Redo* kann zwischen den einzelnen Zuständen in Abbildung 8 hin- und hergesprungen werden; gleichzeitig wird die entsprechende Anweisung im Programm dargestellt. Auch die Möglichkeit, eine *DDD-Sitzung* abzuspeichern – den Programmzustand einschließlich aller Displays und Haltepunkte – ist hier praktisch.

Arbeitet ein Algorithmus auf numerischen Werten, empfiehlt es sich, seine Daten als Plot zu animieren. Abbildung 9 etwa zeigt Ausschnitte aus dem Ablauf eines *mergesort*-Algorithmus, der bereits im DDD-Bild in Abbildung 7 gezeigt wurde. Deutlich ist zu erkennen, wie die Werte aus den beiden Hälften zusammengemischt und schließlich umkopiert werden.

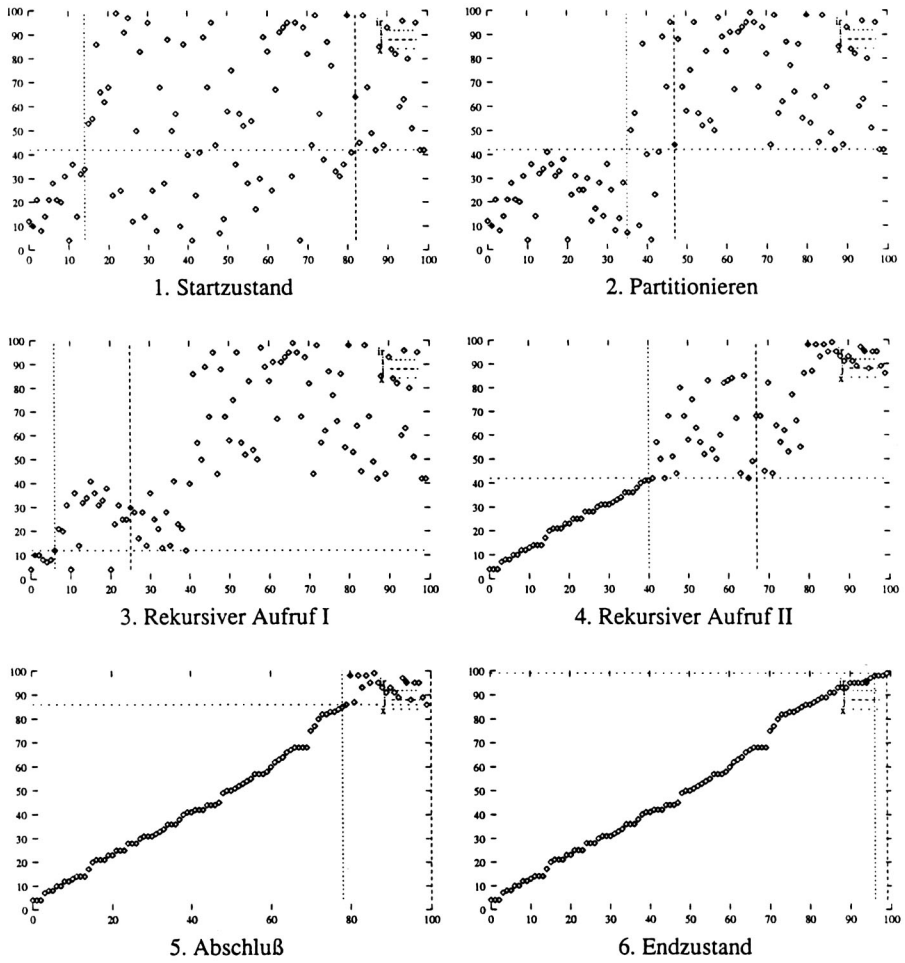


Abb. 10. Animation eines *quicksort*-Algorithmus in DDD

Ein zweites Beispiel zeigt Abbildung 10: Hier animiert DDD einen *quicksort*-Algorithmus. Dargestellt ist der Partitionierungsvorgang – alle Werte, die kleiner sind als das Pivot-Element (horizontale Linie), werden im Feld nach links bewegt; alle Werte, die größer sind, werden nach rechts bewegt. Die vertikalen Linien wandern von außen nach innen und stellen den Fortschritt des Verfahrens dar; treffen sich die Linien, ist die Partitionierung auf dieser Ebene abgeschlossen. In den späteren Schritten sieht man, wie der rekursive Aufruf der Partitionierung das gesamte Feld sortiert.

Noch wirkungsvoller werden diese Darstellungen, wenn man sie *vollautomatisch* ablaufen läßt. Hierzu setzt man einen Haltepunkt, der mit einem *continue*-Kommando gekoppelt ist. Beim Erreichen des Haltepunkts bringt DDD die Darstellung auf den neuesten Stand und führt das *continue*-Kommando aus, wodurch der Programmlauf fortgesetzt wird. Mit *bedingten Haltepunkten* kann die Ausführung an ausgewählten

Stellen unter definierten Bedingungen angehalten werden – etwa im erstmaligen rekursiven Aufruf oder bei Unterschreiten einer gewissen Schranke für den Index.

6 Verwandte Arbeiten

Neben DDD gibt es eine Reihe von Projekten, die ebenfalls Fehlersuche mit Visualisierung verbinden.

VCC [1] visualisiert Programme, indem es beim Übersetzen automatisch Aufrufe für eine Animations-Bibliothek in den Programmcode hinein übersetzt. Im Gegensatz zu DDD ist somit ein eigener Übersetzungsschritt erforderlich; auch ist VCC auf die Programmiersprache C beschränkt. VCC bietet Sichten für den Code, die aktive Funktion, den Baum der Funktionsaufrufe sowie Datenstrukturen.

ZStep95 [6] ist eine Visualisierungsumgebung für Common Lisp. Wie auch VCC fügt ZStep95 dem Programm spezielle Animations-Aufrufe hinzu; wie DDD speichert ZStep95 die Historie von Daten während eines Programmlaufs, die dann mittels eines „Videorekorders" vor- und zurückgespult werden können. Im Gegensatz zu DDD, der nur dargestellte Daten speichert, merkt sich ZStep95 jegliche Veränderungen aller Daten, was erheblich Speicherplatz kostet.

Lens [7] ist wie DDD ein graphisches Debugger-Frontend, in diesem Fall für den DBX-Debugger. Lens bietet ein Entwicklungswerkzeug, in dem interaktiv Animationen zum Code erstellt werden können. Im Gegensatz zu DDD, wo die Animation über den Programmlauf erfolgt, müssen in Lens Animationsanweisungen manuell eingefügt werden. Dafür erlaubt Lens vielfältige Eingriffe in die Animation – etwa Farben, Strukturen, usw. – die bei DDD fest vorgegeben sind.

DEET [5] ist ebenfalls eine graphische Erweiterung für Kommandozeilen-Debugger. In DEET liegt der Schwerpunkt auf *Einfachheit:* Für die graphische Datendarstellung werden externe Layouter und Programme benutzt. Da DEET keine Alias-Analyse durchführt, ist die Darstellung auf Baumstrukturen beschränkt (die aber im Gegensatz zu DDD automatisch entfaltet werden); ein interaktives Layout ist in DEET nicht möglich.

DUEL [4] ist eine spezielle Programmiersprache für Debugger, mit der Daten auf einer höheren Abstraktionsstufe analysiert werden können. So bietet DUEL Quantoren, Filter und Mengenoperationen. Die Erweiterung XDUEL visualisiert die aus DUEL gewonnenen Daten. Der Hauptvorteil von DUEL ist gleichzeitig sein Nachteil: Die speziellen DUEL-Operatoren und -Befehle lösen sich von der Programmiersprache und erschweren so Anfängern die Bedienung.

ISVL [3] ist eine Arbeitsumgebung zur verteilten kollaborativen Fehlersuche. Die Visualisierungskomponente visualisiert eine Ausführungshistorie (ähnlich ZStep95), die dann über einen Web-Server einer Gruppe von Programmierern verfügbar gemacht wird. ISVL ist leider auf die Programmiersprache PROLOG beschränkt.

Von den beschriebenen Projekten ist keines über einen frühen Prototyp-Status hinausgekommen. Es wäre jedoch wünschenswert, wenn zukünftige praktisch eingesetzte Werkzeuge die hier realisierten Anregungen aufgriffen; auch DDD böte sich als Plattform an.

7 Zusammenfassung und Ausblick

DDD ist ein Universalwerkzeug zur dynamischen Softwarevisualisierung. DDD kann verzeigerte Strukturen (wie Listen oder Bäume) als Graphen darstellen, aber auch numerische Felder in zwei- oder dreidimensionalen Plots darstellen und im Programmlauf animieren. Herausragend in DDD ist der schnelle Zugang zu visualisierten Daten – es sind weder aufwendige Neuübersetzungen noch komplexe Befehlsstrukturen noch aufwendige Interaktionen vonnöten.

So angenehm und hilfreich die Datenvisualisierung in DDD sein mag, so schnell werden aber auch die *Grenzen* sichtbar. Ein Graph mit wenigen Dutzend Displays paßt noch gut auf den Bildschirm und ist leicht zu überblicken. In größeren Strukturen wird es aber schnell aussichtslos, Fehler durch simples Betrachten zu finden. Die Visualisierung von Daten zeigt hier dasselbe Problem wie jede andere Visualisierung: die *kognitive Aufnahmefähigkeit* des Menschen ist eine obere Schranke, die auch durch die beste Visualisierung nicht überwunden werden kann.

Man könnte versuchen, das Problem durch „intelligentere" Darstellungen zu lösen – etwa durch Zusammenfassung von Daten oder Datentyp-spezifische Zeichenverfahren. Leider setzen solche Darstellungen Wissen über die Datenstrukturen voraus, das nicht ohne weiteres aus dem Programm abgeleitet werden kann. Wüßte DDD etwa, daß eine Datenstruktur einem gefädelten Baum entspricht, könnte DDD den Baum entsprechend zeichnen. Steht aber diese Intelligenz bereits zur Verfügung, könnte man genausogut entsprechende Prüfprogramme generieren, die jegliche Abweichung vom Lehrbuch-Baum aufdecken. Warum dann nicht gleich eine fertige, gut getestete Komponente wählen und Fehlern von vorneherein aus dem Weg gehen?

Zusammenfassend stößt man mit Visualisierung also offensichtlich an Grenzen, die nicht leicht zu überwinden sind; zusätzliche bereichsspezifische Intelligenz steht im Gegensatz zur angestrebten Universalität von Werkzeugen. Gerade bei Debuggern als Universalwerkzeugen, die schnell und unkompliziert die freie Erkundung des Programmzustands ermöglichen sollen, wird dieser Gegensatz deutlich.

Gelingt es aber, die Problemgröße klein genug zu halten, bringt Visualisierung großen Gewinn – in der Fehlersuche wie auch in der Lehre. Daß Debuggen Spaß machen kann, haben viele Menschen erst mit DDD gelernt, und das liegt nicht zuletzt an seinen universellen und eingängigen Visualisierungstechniken.

Danksagung. Holger Cleve, Torsten Robschink und die anonymen Gutachter gaben wertvolle Kommentare zu diesem Beitrag. Dorothea Krabiell (geb. Lütkehaus) ist Co-Autorin der ersten DDD-Fassung.

DDD, Dokumentation und Online-Verweise auf verwandte Projekte finden Sie unter

`http://www.gnu.org/software/ddd/` .

Literatur

1. BAEZA-YATES, R., G. QUEZADA und G. VELMADRE: *Visual Debugging and Automatic Animation of C Programs*. In: EADES, P. und K. ZHANG (Hrsg.): *Software Visualisation*. World Scientific Press, 1996.

2. DIEHL, S.: *GI-Workshop Softwarevisualisierung – Call for Papers*, Jan. 2000.
3. DOMINGUE, J. und P. MULHOLLAND: *Fostering Debugging Communities on the Web*. Communications of the ACM, 40(4):65–71, Apr. 1997.
4. GOLAN, M. und D. R. HANSON: *DUEL—a very high-level debugging language*. In: *Proceedings of the Winter USENIX Technical Conference*, S. 107–117, San Diego, California, USA, Jan. 1993.
5. HANSON, D. R. und J. L. KORN: *A Simple and Extensible Graphical Debugger*. In: *Proceedings of the USENIX 1997 Annual Technical Conference*, Bd. 183-174, Anaheim, California, USA, Jan. 1997.
6. LIEBERMANN, H. und C. FRY: *ZStep95: A Reversible, Animated Source Code Stepper*. In: STASKO, J., J. DOMINGUE, B. PRICE und M. BROWN (Hrsg.): *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1997.
7. STASKO, J. T. und S. MUKHERJEA: *Toward Visual Debugging: Integrating Algorithm Animation Capabilities Within a Source-Level Debugger*. ACM Transactions on Computer-Human Interaction, 1(3):215–244, Sep. 1994.
8. SUGIYAMA, K. und K. MISUE: *Visualization of Structural Information: Automatic Drawing of Compound Digraphs*. IEEE Transactions on Systems, Man, and Cybernetics, SMC-21(4):876–892, Juli 1991.

Über generisches visuelles Programmieren *

Berthold Hoffmann[1] und Mark Minas[2]

[1] Fachbereich Mathematik/Informatik, Universität Bremen
Postfach 33 04 40, 28334 Bremen
hof@informatik.uni-bremen.de
[2] Lehrstuhl für Programmiersprachen, Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen
minas@informatik.uni-erlangen.de

Zusammenfassung Graphen sind nützlich für die abstrakte Darstellung visueller Spezifikationen. In diesem Papier umreißen wir eine Sprache, die geschachtelte Graphen unterstützt sowie regelbasiert und objektorientiert ist. Insbesondere gehen wir auf die *Generizität* der Sprache ein. Dadurch lassen sich Graphen als verschiedene Arten von Diagrammen visualisieren; Programme können so Notationen für die Ein-Ausgabe verwenden, wie sie im jeweiligen Anwendungsgebiet gebräuchlich sind. Die Sprache ist selbst visuell und benutzt den Diagramm-Editor-Generator DiaGen um Generizität zu unterstützen, sowie zur Implementierung ihrer Benutzungsschnittstelle und zur Typüberprüfung. .

1 Einleitung

Graphen sind nützlich für die abstrakte Darstellung visueller Spezifikationen. Deshalb ist die regelbasierte Transformation von Graphen [20, 7] ein interessantes Berechnungsmodell für visuelle Programmiersprachen. Existierende Graphtransformationssprachen wie PROGRES [21] waren in dieser Anwendung jedoch nicht so erfolgreich wie erwartet werden konnte. Wir glauben, daß dies hauptsächlich zwei Gründe hat: Zum einen wird die *Schachtelung* von Graphen (analog derer von *Objekten*) und damit auch die *objektorientierte Programmierung* nicht unterstützt; außerdem ist es nur schwer möglich, die "Standard"-Notation von Graphen an die visuellen Notationen bestimmter Anwendungsgebiete *anzupassen.*

Für das visuelle Programmieren sind schon einige objektorientierte Sprachen entwickelt worden [2], insbesondere *Prograph* [4]. Aber auch alle visuellen Sprachen, die wir kennen, sind mehr oder weniger an eine Diagrammnotation gebunden.

In diesem Papier geht es um eine Programmiersprache, die auf Graphtransformation basiert, geschachtelte Graphen und objektorientiertes Programmieren unterstützt und visuell (*graph*-isch) und regelbasiert ist. Wir diskutieren insbesondere die *Generizität* der Sprache, dank derer die intern verarbeiteten Graphen

* Diese Arbeit wurde von der ESPRIT-Arbeitsgruppe *Applications of Graph Transformation* (APPLIGRAPH) unterstützt.

in einer Vielfalt von Diagrammnotationen visualisiert werden können. In dieser Sprache implementierte Systeme können dadurch unmittelbar die Darstellungsweisen nutzen, die im jeweiligen Anwendungsgebiet gebräuchlich sind. Die konzipierte Programmier- und Ausführungsumgebung gestattet es dem Programmierer und dem späteren Anwender, anwendungsspezifische Diagramme sowohl in der Eingabe als auch der Ausgabe des Systems zu verwenden. Damit unterscheidet sich die Programmiersprache mit ihrer zugehörigen Umgebung deutlich von anderen Systemen (wie Smalltalk- oder Prolog-Umgebungen), die zwar auch graphische Ein- und Ausgaben erlauben, die aber getrennt von den intern verwendeten Datenstrukturen zu programmieren sind.

Das Papier gliedert sich wie folgt: In Abschnitt 2 umreißen wir exemplarisch einige grundlegende Konzepte einer graph- und regelbasierten Programmiersprache; auf die anwendungsspezifische Visualisierungen der verwendeten Graphen wird dabei noch nicht eingegangen. Dies ist Thema von Abschnitt 3, das rekapituliert, wie Graphen als Datenstruktur bei der Analyse von Diagrammen benutzt werden können, und das erklärt, wie dies ausgenutzt wird, um das Programmieren mit Graphen *generisch* an viele verschiedene Diagrammnotationen anzupassen. In Abschnitt 4 beschreiben wir, wie so eine generische visuelle Programmiersprache implementiert werden kann. Abschließend machen wir in Abschnitt 5 einige Bemerkungen zu verwandten und zukünftigen Arbeiten.

Schon aus Platzgründen kann unsere Darstellung nur informell sein. Wir erklären die Konzepte an einem Beispiel, in dem es um die Darstellung von Listen als Graphen geht.

2 Programmieren mit Graphen und Regeln

In diesem Abschnitt beschreiben wir exemplarisch, wie auf der Basis eines Modells *geschachtelter Graphen* regelbasiert und objektorientiert programmiert werden kann. Eine ausführlichere Darstellung der hier umrissenen Programmierkonzepte findet sich in [10].

2.1 Graphen

Graphen repräsentieren Relationen zwischen Größen (*entities*) als *Kanten* zwischen *Knoten*. Üblicherweise verbinden Kanten zwei Knoten eines Graphen und repräsentieren zweistellige Relationen. Wir erlauben dagegen, daß Kanten *beliebig* viele Knoten verbinden, und unterscheiden verschiedene *Kantentypen* durch Markierungen, so dass in einem Graphen verschiedene Relationen beliebiger Stelligkeit repräsentiert werden können. Außerdem zeichnen wir eine Folge von Knoten als *Punkte* aus, an denen Graphen miteinander verknüpft werden können. Solche Graphen sind in der Literatur als *punktierte Hypergraphen* bekannt [5].

Üblicherweise sind Graphen flach: Sie können keine *geschachtelten* Graphen enthalten. Wir dagegen zeichnen eine Teilmenge der Kanten in einem Graphen H als *Rahmen* (*frames*) aus: Jeder Rahmen f *enthält* einen Untergraphen H_f, der

selbst geschachtelt sein kann. Solche Graphen werden in [6] *hierarchisch* genannt und sind sind für das objektorientierte Programmieren mit Graphen unerläßlich.

Abbildung 1 zeigt, wie Listen als Graphen dargestellt werden können: Ein

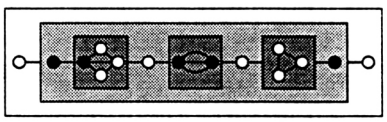


Abbildung 1. Ein Listenrahmen mit drei geschachtelten Elementrahmen

zweistelliger *Listenrahmen* enthält eine Kette von ebenfalls zweistelligen Elementrahmen. Jeder *Elementrahmen* enthält einen *Elementgraphen.*

Knoten werden als Kreise gezeichnet und sind schwarz, wenn es sich um Punkte handelt. Kanten werden als Kästen um ihre Markierung gezeichnet, und mit Knoten durch Linien verbunden. (Die Kästen einfacher binärer Kanten "verschwinden", so daß sie als Linien zwischen zwei Knoten gezeichnet werden.) Rahmen sind Kästen (wie normale Kanten auch), in die der Inhalt hineingezeichnet wird. Wir unterscheiden Listen- und Elementrahmen durch ihren Grauton und lassen dafür ihre Markierungen weg. Der Graph in Abbildung 1 enthält einen Listenrahmen, der drei Elementrahmen enthält, deren Inhalt in diesem Beispiel keine besondere Bedeutung hat.

Kanten über Rahmengrenzen hinweg, wie andere Hierarchiebegriffe [8] sie erlauben, sind in diesen Graphen *verboten*, damit sie modular transformiert werden können. Die Verbindungen eines Rahmens, und die Punkte seines Inhalts können jedoch eine *indirekte Beziehung* zwischen dem Inhalt und der Umgebung des Rahmens ausdrücken. Der Listenrahmen in Abbildung 1 drückt so eine indirekte Beziehung zwischen den Punkten des in ihm enthaltenen Listengraphen und seinen Verbindungen aus.

Die hier betrachteten Graphen sind *attributiert*: Ihre Knoten und Kanten können mit Werten assoziiert werden, die mittels semantischer Funktionen berechnet werden. Attribute sind größtenteils einfache Werte wie Zahlen (von Bildschirmkoordinaten u. dgl.). Attribute werden jedoch der Einfachheit zuliebe in diesem Papier nicht weiter behandelt und tauchen daher auch nicht in den Beispielen auf.

2.2 Abstraktion

Die Abbildungen 2 und 3 zeigen *Prädikate*, die Graphen transformieren. Die Definition des Prädikats remove in Abbildung 2 enthält zwei Graphen: Der linke ist ein *Muster* (*pattern*), das in dem zu transformierenden *Wirtsgraphen* gesucht wird (*matching*). Dabei können die *Variablen* X und L an beliebige Untergraphen gebunden werden. Das Muster mit der remove-Kante und dem Listenrahmen wird an den Punkten aus dem Wirtsgraphen *herausgeschnitten*, und eine

Kopie des rechten Graphen dafür *eingeklebt*, wobei dessen Variable L durch den an sie gebundenen Untergraphen *instanziiert* wird.

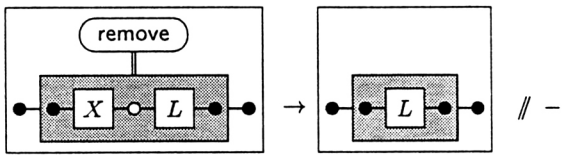


Abbildung 2. Das Prädikat remove

Ein Prädikat kann durch mehrere Ersetzungsregeln definiert werden, die sich nicht ausschließen müssen; für remove reicht eine. Falls diese Regel nicht angewendet werden kann, legt die *otherwise*-Definition "// -" fest, daß das Prädikat *scheitert* und *Backtracking* eingeleitet werden soll. Das Prädikat remove löscht also den ersten Elementgraphen X in einem Listenrahmen und scheitert, wenn es auf den leeren Listengraphen angewendet wird.

Im allgemeinen können die rechten Seiten von Regeln in Prädikaten weitere (oval gezeichnete) Prädikatkanten enthalten, mit denen weitere Prädikate aufgerufen werden. Die *Auswertung* eines Prädikats verlangt es daher im allgemeinen, zunächst eine seiner Regeln anzuwenden, und dann rekursiv alle dadurch eingefügten Prädikataufrufe auszuwerten.

Die Verbindungen einer Prädikatkante geben *Parameter* an; sie sind als Doppellinien dargestellt. Ein Prädikatparameter kann einfach ein *Knoten*, aber auch eine *Kante* sein. Insbesondere kann diese Kante ein Rahmen sein, der einen *Graphparameter* angibt (wie in diesem Beispiel), oder ein *Prädikatparameter*, wie im nächsten Beispiel.

2.3 Steuerung

Abbildung 3 zeigt den Kombinator normalize, der auf einen mit der Variablen T bezeichneten *Prädikatparameter* angewendet wird und diesen solange auswertet, bis dies fehlschlägt. Für den Fall signalisiert ein "+" in der *otherwise*-Definition, daß normalize *gelingt*, und die Auswertung weiter gehen kann.

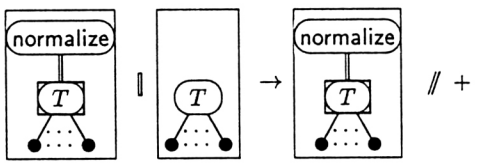


Abbildung 3. Der Kontrollkombinator normalize

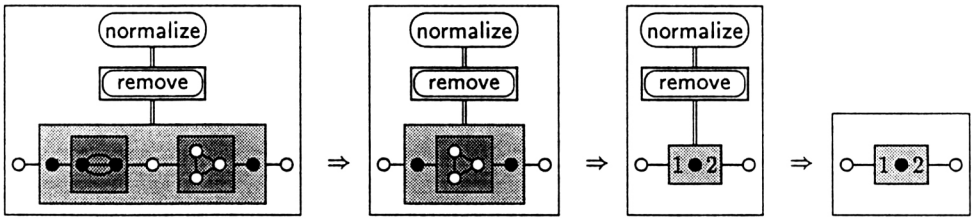


Abbildung 4. Eine Auswertung von normalize

Das Prädikat ist mit einer *bedingten Regel* der Form $P \parallel A \rightarrow R$ definiert. Die *Anwendbarkeitsbedingung* A "bewacht" die Regel: Nur wenn sie vollständig ausgewertet werden kann, ohne zu scheitern, wird die Regel angewendet.

Da T ein Platzhalter für Prädikate mit beliebig vielen Parametern sein soll, benutzen wir Punkte, um eine veränderliche Anzahl von Knoten anzudeuten. Wo die T-Kante als Prädikatparameter benutzt wird, wird sie außerdem als einfache Kante *getarnt*, indem ein Kasten um das Oval gezogen wird. Das verhindert, dass sie ausgewertet wird, während sie "umhergetragen" wird (nämlich im Muster- und Ersetzungsgraph der Regel).

Abbildung 4 zeigt die Auswertung von normalize mit einem (getarnten) Aufruf von remove. Jede Anwendung von normalize entfernt einen Elementrahmen, indem remove als Anwendbarkeitsbedingung ausgewertet wird, bis der Listenrahmen leer ist, und remove fehlschlägt. (Der leere Listengraph besteht aus einem einzelnen Knoten, wobei die Ziffern 1 und 2 angeben, dass dieser Knoten gleichzeitig Start- und Endpunkt des leeren Listengraphen ist.)

Prädikate stellen also ähnliche Steuerungsmechanismen zur Verfügung wie funktionale und logische Sprachen: *Pattern matching* und *otherwise*-Definitionen erlauben Fallunterscheidungen; mit Anwendbarkeitsbedingungen kann die Reihenfolge von Prädikatanwendungen gesteuert werden. Zusammen mit Rekursion und Kombinatorprädikaten wie normalize reicht das schon aus, um in der Sprache Steuerung zu spezifizieren.

2.4 Kapselung

Wir sehen eine datenorientierte Art der Modularisierung vor. Abbildung 5 zeigt die globale Sicht einer Klasse L_T. Die Klasse L_T ist eine Vorlage (*template*) für Listenrahmen. Ihre Definition enthält öffentliche Methoden enter oder remove. (In diesem einfachen Fall gibt es keine verborgenen Methoden.)

Wir fassen Listenrahmen als *Objekte* auf, die als Instanzen der Klasse L_T gebildet werden und auf *Nachrichten* (Aufrufe der öffentlichen Prädikate enter oder remove) reagieren. Die Struktur der Listengraphen ist jedoch nur in der Klasse, für deren Methoden sichtbar. Andere Objekte, die einen Listenrahmen l enthalten, können auf dessen Inhalt nur zugreifen, indem sie eine Nachricht enter oder remove an l schicken. So wird das Prinzip der *Datenabstraktion* bewahrt.

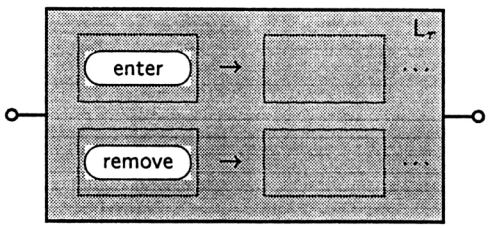


Abbildung 5. Die Listen-Klasse

2.5 Typisierung

Typen klassifizieren die Objekte eines Programms und geben Regeln für deren Benutzung vor. Über die übliche Typisierung von Graphen durch Markierung und Beschränkung des Grades von Knoten und Kanten hinaus erlauben wir, die *Struktur* (*shape* [9]) der in Rahmen enthaltenen Graphen zu definieren.

Abbildung 6 spezifiziert die Struktur von Listengraphen. Nur so aufgebaute

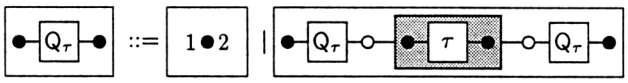


Abbildung 6. Die Struktur von Listengraphen

Graphen dürfen in Listenrahmen enthalten sein. Der Typ L_τ der Listengraphen ist *polymorph*. Die *Typvariable* τ kann mit beliebigen Typen instanziiert werden. In unseren Beispielen hat die Variable L den Typ L_τ, und die Variable X den Typ τ, der in den Transformationen von Abbildung 3 mit dem Typ der Elementgraphen instanziiert wird.

Jede Variable, jede Klasse, jede Regel und jeder Prädikatparameter hat einen so spezifizierten Typ, und es wird überprüft, ob alle Variablensubstitutionen, alle Rahmeninhalte, und die Muster- und Ersetzungsgraphen von Regeln diesen Typen *entsprechen*. Dann sind alle Graphen eines Programms wohlstrukturiert.

Die Strukturdefinitionen sind gut untersucht als *kontextfreie Graphtransformationen* [5]. Typüberprüfung bedeutet deshalb *kontextfreies Graphparsieren*, wie es in DiaGen implementiert ist (vgl. Abschnitt 3). Auch wenn Graphparsieren teuer ist, wird es *vor* Ausführung des Programms gemacht, und verringert somit den Suchraum für das *Graphmatching* zur Laufzeit.

3 Diagramme

Dieser Abschnitt ruft kurz in Erinnerung, wie Diagramme als Graphen repräsentiert werden können (beispielsweise in Diagrammeditoren), und wie dieses Modell dazu benutzt werden kann, Graphen *generisch* als Diagramme darzustellen.

3.1 Repräsentation und Analyse von Diagrammen

Andries *et al.* [1] haben ein Modell für die Repräsentation von Diagrammen mir Graphen vorgeschlagen, das modifiziert auch in DIAGEN realisiert ist, einem Werkzeug zur Erzeugung von Diagrammeditoren, die sowohl freies als auch strukturiertes Editieren unterstützen [12, 14, 15]. Abbildung 7 zeigt die Ebenen der Diagramm-Repräsentation und die Schritte, in denen ein Diagramm analysiert wird. (Beachte die Analogie zur Analyse textueller Sprachen [26].)

Der *Scanner* baut einen *Layoutgraph* (*spatial relationship graph* [1]) auf, um die lexikalische Struktur des Diagramms festzustellen. Dieser Schritt benutzt Kanten, um die *Komponenten* einer Diagrammsprache zu repräsentieren (wie Kreise, Kästen, Pfeile und Text). Die Komponentenkanten sind mit Knoten verbunden, die die *Anknüpfungsbereiche* der Komponenten repräsentieren (wie *Rand* und *Fläche* von Kreisen und Kästen, *Quell-* und *Zielenden* von Pfeilen).

Die Komponenten können in ihren Anknüpfungsbereichen verbunden sein. Dies wird explizit durch binäre *Anordnungskanten* repräsentiert. Die Art einer Anordnungskante berücksichtigt die Typen der verbundenen Anknüpfungsbereiche und die Art ihrer Verbindung (wie *Überschneidung* oder *Enthaltensein*). Beispielsweise können sich *Quell-* und *Zielbereiche* von Pfeilen mit den *Randbe-*

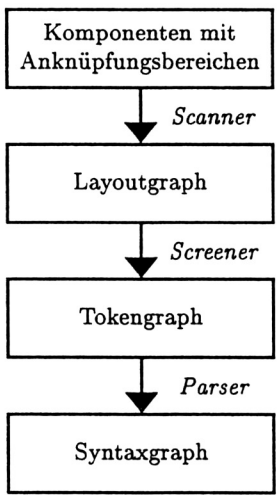


Abbildung 7. Diagrammrepräsentation und -erkennung in DIAGEN

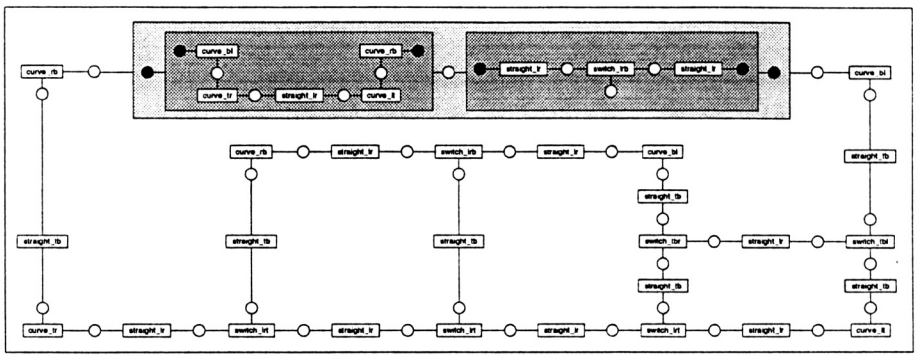


Abbildung 8. Der Syntaxgraph eines Schienendiagramms

reichen von Kreisen und Kästen überschneiden, was zu einer bestimmten Anordnungskante zwischen den betreffenden Knoten der Komponentenkanten führt.

Die weitere Analyse vollzieht sich in zwei Schritten: Der Layoutgraph wird zuerst vom *Screener* in einen kompakteren *Tokengraphen* überführt, indem Kantengruppen zu Einzelkanten zusammengefasst werden. Der Tokengraph wird dann gemäß der Syntax der Diagrammsprache *parsiert*. Diese zweistufige Analyse transformiert den Layoutgraph in die logischen Relationen des (*abstrakten*) *Syntaxgraphen* des Diagramms. Die Syntax von Diagrammen wird mit graphischen Regeln beschrieben, wie sie auch für die Typisierung von Graphen in Abschnitt 2.5 verwendet werden. Abbildung 8 repräsentiert den Layoutgraphen des Diagramms in Abbildung 9.

Mit diesem Modell können viele verschiedene Arten von Diagrammen repräsentiert und analysiert werden: endliche Automaten und Kontrollflußdiagramme [16], Nassi-Shneiderman-Diagramme (Struktogramme) [13], *message sequence charts* und *visual expression diagrams* [15], *sequential function charts* [12], und Kontaktpläne aus der SPS-Programmierung [14]. Derzeit sind uns keine Diagrammsprachen bekannt, die so nicht modelliert werden können.

3.2 Generische Visualisierung von Graphen

Da das oben beschriebene Modell zur Repräsentation und Analyse von Diagrammen Graphen benutzt, können wir es umgekehrt auch dazu benutzen, die Daten einer graphbasierten Programmiersprache als Diagramme zu visualisieren. Graphen sind zwar auch eine visuelle Darstellung, die aber für viele Anwendungen nicht angemessen ist. Statt dessen können wir für die *externe* Darstellung von Graphen beliebige visuelle Darstellungen nehmen. Die Benutzungsschnittstelle eines Programms kann so an die visuelle Notation seines Anwendungsgebietes angepasst werden. Damit kann die in hier vorgestellte Programmiersprache als *generische visuelle Programmiersprache* benutzt werden. Indem Graphen als verschiedene Diagrammarten repräsentiert werden, können verschiedene Stile visu-

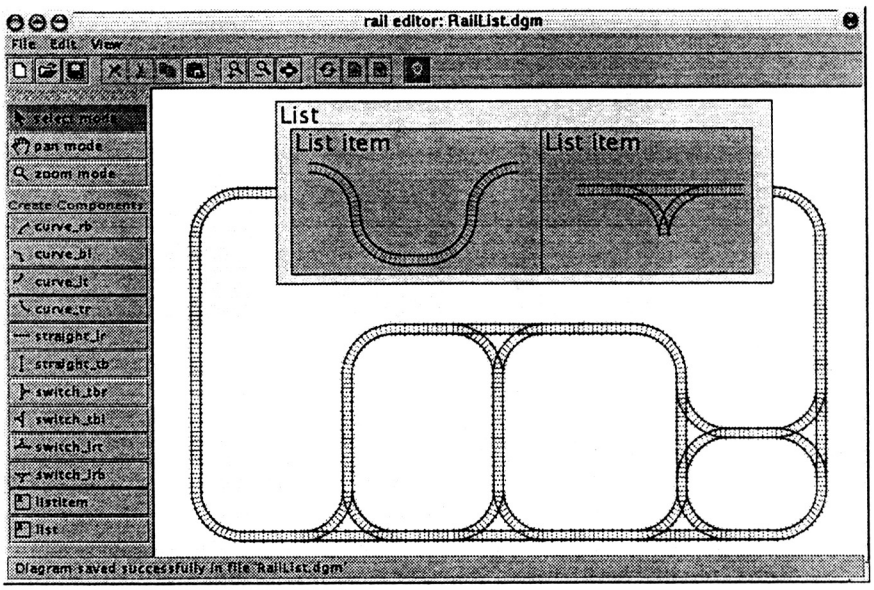


Abbildung 9. Ein Schienendiagramm mit einem Listendiagramm

eller (Programmier-) Sprachen beschrieben und implementiert werden. Naheliegende Beispiele wären *Pictorial Janus* [11] (dessen Agenten und *ports* unserem Begriff von typisierten Kanten direkt entsprechen) oder *KidSim* [22].

Abbildung 9 illustriert, wie die Generizität des Diagrammmodells zum Bau einer *KidSim*-artigen Umgebung benutzt werden kann. Der Bildschirmausschnitt zeigt einen Diagramm-Editor, der mit DIAGEN erzeugt wurde: Datenstrukturen sind Eisenbahnschienen, erweitert um abstrakte Datentypen (Listen). Der Editor ist Teil der Benutzungsschnittstelle zu einem (noch nicht realisierten) Programm, das es erlaubt, Schienennetze zu bearbeiten. Abstrakte Datentypen (Listen etc.) werden dabei unter anderem benötigt, den Vorrat an Schienen oder bestimmte Konfigurationen (z. B. Bahnhöfe) verwalten zu können. Das Programmierwerkzeug (das derzeit noch nicht Bestandteil des Editors ist), manipuliert diese Eisenbahnschienen visuell. Intern, aber für den Benutzer nicht sichtbar, arbeitet das Werkzeug auf der Graph-Repräsentation in Abbildung 8. DIAGEN benutzt ohnehin Graphen als interne Repräsentation, und ist daher für die generische Implementierung derartiger Umgebungen besonders gut geeignet.

4 Implementierung

Die in diesem Papier vorgestellte Programmiersprache ist noch nicht fertig entworfen. Ihre Implementierung steht erst ganz am Anfang und kann deshalb hier nur skizziert werden (vgl. Abbildung 10).

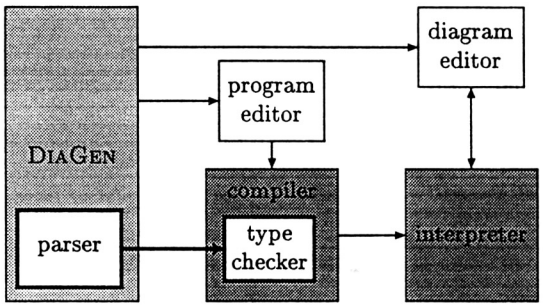


Abbildung 10. Systemarchitektur

- Der *Interpreter* führt Programme der Sprache aus, indem in einer Schleife – gesteuert vom Benutzer – ein Eingabegraph gelesen, gemäß dem Programm transformiert, und der modifizierte Graph wieder dargestellt wird.
- Der *Compiler* liest Programme und formt sie so um, dass die vom Interpreter leicht und effizient ausgeführt werden können, und überprüft, ob das Programm irgendwelche Regeln der Sprache verletzt. Dabei kommt der Typisierung (wie in Abschnitt 2.5 diskutiert) eine besondere Rolle zu: Die Typüberprüfung soll mithilfe des Graphparsers von DiaGen [13] realisiert werden.
- Mit einem interaktiven *Diagramm-Editor* werden die Eingabedaten für das Programm konstruiert. Dieser Editor soll mit DiaGen erzeugt werden, damit die intern verarbeiteten Graphen flexibel als Diagramme dargestellt werden können. Ein solcher Editor für eine *KidSim*-artige Diagrammnotation wurde bereits als Prototyp realisiert (vgl. Abschnitt 3.2). Abbildung 9 zeigt einen Bildschirmausschnitt.
- Die Programme werden mit einem interaktiven *Programm-Editor* für die visuelle Syntax der Sprache konstruiert. Auch dieser Editor soll mit DiaGen erzeugt werden.

Alles in allem muss also nur das Herz des Systems – Übersetzer und Interpreter – neu implementiert werden, während wir für die Erzeugung der Benutzungsschnittstelle ganz auf DiaGen bauen.

DiaGen muss noch in einigen Punkten ergänzt werden, um den Anforderungen dieser Anwendung gerecht zu werden. Insbesondere braucht das System selbst einen visuellen Editor. Es ist sicher keine Überraschung, dass auch dafür DiaGen benutzt werden soll.

Die Implementierung von Übersetzer und Interpreter ist eine anspruchsvolle Aufgabe. Auch wenn wir den Leser hoffentlich davon überzeugt haben, dass alle vorgeschlagenen Sprachkonzepte implementierbar sind, heißt das lange noch nicht, dass dies auch *effizient* geht, und zu *effizienten Systemen* führt. Die Effizienz der Implementierungen logischer und funktionaler Sprachen müsste erreicht werden können, weil die Fragen der Benutzungsschnittstelle vom Kern des Systems abgekoppelt werden können.

5 Schlußbetrachtungen

In diesem Papier haben wir eine auf geschachtelten Graphen basierende Programmiersprache vorgestellt, die regelbasiert, typisiert und objektorientiert ist. Wir haben insbesondere ihre Generizität in Hinblick auf Diagrammnotationen diskutiert.

Geschachtelte oder *hierarchische* Graphen wurden im Zusammenhang mit Graphtransformation bereits in [19, 8] vorgeschlagen, und auch in graphbasierten Datenbanksprachen [18] und in Systemmodellierungssprachen [25]. Graph-*Strukturen* gibt es in *Structured Gamma* [9] (für ungeschachtelte Graphen).

Wir kennen jedoch keine andere Sprache (auch keinen Entwurf), die geschachtelte Graphen, Transformationen, Typisierung und Klassen mit Generizität kombiniert.

Die präzise Definition der in diesem Papier genannten Konzepte wurde in [6] begonnen, und muss weiter geführt werden. Einige weitere Konzepte, wie *Nebenläufigkeit* und *Verteiltheit*, sollen noch untersucht werden. Da diese Konzepte schon in [23] und [24] in ähnlichem Zusammenhang studiert wurden, gibt es berechtigte Hoffnung darauf, dass sie sich auf unsere Sprache übertragen lassen.

Literatur

1. ANDRIES, M., G. ENGELS und J. REKERS: *How to Represent a Visual Specification*. In: MARRIOTT, K. und B. MEYER (Hrsg.): *Visual Language Theory*, Kap. 8, S. 245–260. Springer, New York, 1998.
2. BURNETT, M. M., A. GOLDBERG und T. G. LEWIS (Hrsg.): *Visual Object-Oriented Programming*. Manning, Greenwich, CT, 1994.
3. CORRADINI, A. und U. MONTANARI (Hrsg.): *Proc. Joint* COMPUGRAPH/SEMAGRAPH *Workshop on Graph Rewriting and Computation*, Nr. 2 in *Electronic Notes in Theoretical Computer Science*, `http://www.elsevier.nl/locate/entcs`, 1995. Elsevier.
4. COX, P. T., F. R. GILES und T. PIETRZYKOWSKI: *Prograph*. In: BURNETT, M. M. et al. [2], Kap. 3, S. 45–66.
5. DREWES, F., A. HABEL und H.-J. KREOWSKI: *Hyperedge Replacement Graph Grammars*. In: ROZENBERG, G. [20], Kap. 2, S. 95–162.
6. DREWES, F., B. HOFFMANN und D. PLUMP: *Hierarchical Graph Transformation*. In: TIURYN, J. (Hrsg.): *Foundations of Software Science and Computation Structures (FOSSACS 2000)*, Nr. 1784 in *Lecture Notes in Computer Science*, S. 98–113. Springer, März 2000.
7. ENGELS, G., H. EHRIG, H.-J. KREOWSKI und G. ROZENBERG (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Specification and Programming*. World Scientific, Singapore, 1999.
8. ENGELS, G. und A. SCHÜRR: *Encapsulated Hierachical Graphs, Graph Types, and Meta Types*. In: CORRADINI, A. und MONTANARI [3].
9. FRADET, P. und D. L. MÉTAYER: *Structured Gamma*. Science of Computer Programming, 31(2/3):263–289, 1998.
10. HOFFMANN, B.: *From Graph Transformation to Rule-Based Programming with Diagrams*. In: NAGL, M. und SCHÜRR [17].

11. KAHN, K. M. und V. SARASWAT: *Complete Visualizations of Concurrent Programs and Their Executions*. In: *Proc. IEEE Workshop on Visual Languages (VL'90)*, S. 7–15, 1990.
12. KÖTH, O. und M. MINAS: *Generating Diagram Editors Providing Free-Hand Editing as well as Syntax-Directed Editing*. In: *Proc. International Workshop on Graph Transformation (GRATRA 2000), Berlin*, März 2000.
13. MINAS, M.: *Diagram Editing with Hypergraph Parser Support*. In: *Proc. 13th IEEE International Symposium on Visual Languages (VL'97), Capri, Italy*, S. 230–237. IEEE Computer Society Press, 1997.
14. MINAS, M.: *Creating Semantic Representations of Diagrams*. In: NAGL, M. und SCHÜRR [17].
15. MINAS, M.: *Hypergraphs as a Uniform Diagram Representation Model*. In: EHRIG, H., G. ENGELS, H.-J. KREOWSKI und G. ROZENBERG (Hrsg.): *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, Nr. 1764 in *Lecture Notes in Computer Science*, S. 281–295. Springer, 2000.
16. MINAS, M. und G. VIEHSTAEDT: *DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams*. In: HAARSLEV, V. (Hrsg.): *Proc. 11th IEEE International Symposium on Visual Languages (VL'95), Darmstadt, Germany*, S. 203–210. IEEE Computer Society Press, 1995.
17. NAGL, M. und A. SCHÜRR (Hrsg.): *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'99), Selected Papers*, Lecture Notes in Computer Science. Springer, Mai 2000.
18. POULOVASSILIS, A. und M. LEVENE: *A Nested-Graph Model for the Representation and Manipulation of Complex Objects*. ACM Transactions on Information Systems, 12(1):35–68, 1994.
19. PRATT, T. W.: *Pair Grammars, Graph Languages and String-to-Graph Translations*. Journal of Computer and System Sciences, 5:560–595, 1971.
20. ROZENBERG, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.
21. SCHÜRR, A., A. WINTER und A. ZÜNDORF: *The PROGRES Approach: Language and Environment*. In: ROZENBERG, G. [20], Kap. 13, S. 487–550.
22. SMITH, D. C., A. CYPHER und J. SPOHRER: *KidSim: Programming Agents Without a Programming Language*. Communications of the ACM, 37(7):54–67, 1994.
23. TAENTZER, G.: *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. Dissertation, TU Berlin, 1996. Shaker Verlag.
24. TAENTZER, G. und A. SCHÜRR: *DIEGO, another step towards a module concept for graph transformation systems*. In: CORRADINI, A. und MONTANARI [3].
25. TAPKEN, J.: *Implementing Hierarchical Graph Structures*. In: FINANCE, J.-P. (Hrsg.): *Proc. Formal Aspects of Software Engineering (FASE'99)*, Nr. 1577 in *Lecture Notes in Computer Science*. Springer, 1999.
26. WILHELM, R. und D. MAURER: *Übersetzerbau — Theorie, Konstruktion, Generierung*. Springer, 2. Aufl., 1992.

KIEL – Ein Computersystem zur Visualisierung der Auswertung von funktionalen Programmen

Rudolf Berghammer

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Olshausenstraße 40, D-24098 Kiel

Zusammenfassung Wir stellen das Computersystem KIEL vor, welches die interaktive Auswertung von funktionalen Programmen erster Stufe erlaubt, die in einer einfachen Teilsprache von Standard ML geschrieben sind. KIEL wurde für den Unterricht im Grundstudium konzipiert und hier insbesondere im Hinblick auf die Visualisierung von Programmauswertungen und die Auswirkungen von Berechnungsregeln.

1 Einleitung

Seit etwa zwei Jahrzehnten findet das funktionale Programmierparadigma immer größeren Anklang und wird mittlerweile auch gerne verwendet, um an Universitäten den Informatikstudierenden die grundlegenden Konzepte des Programmierens nahezubringen, oder sogar, wie in [2] beschrieben, jene der gesamten Informatik. Ein Grund dafür ist sicher, daß funktionale Programme in der Regel einfacher, leichter zu erstellen und deshalb auch weniger fehlerhaft sind als ihre imperativen Gegenstücke. Ein weiterer Grund ist, daß bei ihnen Syntax und Semantik sehr nahe zusammenliegen, denn dies impliziert, daß Eigenschaften meist einfacher zu verifizieren sind als im imperativen Fall.

Eine bekannte operationale Art, die Semantik von funktionalen Programmen zu definieren, verwendet Berechnungssequenzen $e_1, e_2, \ldots$ zur Beschreibung der Einzelschritte einer Auswertung. Dabei sind die e_i Terme und werden, ausgehend vom auszuwertenden Originalterm e_1, nacheinander dadurch erzeugt, daß man e_{i+1} aus e_i durch einen Expansionsschritt gefolgt von einem Vereinfachungsschritt gewinnt. Der Expansionsschritt ersetzt dabei gewisse Aufrufe einer Rechenvorschrift jeweils durch den Rumpf, wobei in diesem noch die formalen Parameter durch die jeweiligen Argumente substituiert werden. Welche Aufrufe zu ersetzen sind, wird durch eine Berechnungsregel (auch Parameterübergabemechanismus genannt) bestimmt; Beispiele sind Leftmost-innermost-Ersetzung (Call-by-value), Leftmost-outermost-Ersetzung (Call-by-name) und volle Ersetzung (Herbrand-Regel). Der an die Expansion sich anschließende Vereinfachungsschritt führt die Operationen der primitiven Datentypen und Fallunterscheidungen soweit wie möglich aus. Bezüglich der Details verweisen wir auf [6].

Die eben skizzierte Semantik wertet ein funktionales Programm imgrundegenommen dadurch aus, daß bedeutungserhaltende Transformationen angewendet

werden. Sie ist nicht nur leicht verständlich, sondern auch einfach zu implementieren. Die Vorteile, die sich aus einem entsprechenden Computersystem ergeben, sind vielfältig. Dies gilt insbesondere, wenn man mittels moderner GUI-Toolkits Terme geeignet graphisch darstellt und dem Benutzer die volle Kontrolle über den Auswertungsprozeß überläßt. So ein System kann etwa dazu dienen, Programmauswertungen und die Auswirkungen der Berechnungsregeln zu verdeutlichen. Man kann damit aber auch Programmabläufe schrittweise nachvollziehen, z.B. um Schwachstellen und logische Fehler zu erkennen. Eine Visualisierung der Auswertung kann schließlich noch dazu verwendet werden, die Ideen zu erkennen, die dem durch das funktionale Programm implementierten (abstrakten) Algorithmus zugrunde liegen, sowie den durch sie verursachten Aufwand.

Die Vorteile einer interaktiven Auswertung funktionaler Programme wurden auch andererorts erkannt, und mittlerweile existieren einige Computersysteme zu diesem Zweck. In [3] wird beispielsweise das System MIRACALC beschrieben, welches die schrittweise Auswertung von Miranda-Programmen erlaubt. Ein weiteres solches System ist π-RED. Es unterstützt die interaktive Auswertung von funktionalen Programmen, welche in KIR (der "Kiel reduction language") geschrieben sind; siehe [4]. Auch für viele Lisp-Systeme existieren Debugger, die ein schrittweises Arbeiten erlauben. Abgesehen von einigen nicht-publizierten ad hoc Ansätzen ist dies jedoch nicht der Fall für alle bekannten Implementierungen von modernen funktionalen Sprachen wie Haskell, Miranda und ML.

In diesem Artikel beschreiben wir ein Computersystem, das es erlaubt, die Auswertung von funktionalen Programmen erster Stufe zu visualisieren, die in einer einfachen Teilsprache von ML geschrieben sind. Es heißt KIEL, was ein Akronym ist für "Kiel Interactive Evaluation Laboratory", und wurde speziell für den Unterricht im Informatik-Grundstudium an Universitäten konzipiert. Der Rest des Artikels ist wie folgt gegliedert: In Abschnitt 2 stellen wir die Teilsprache von ML vor, die das System unterstützt. Abschnitt 3 beschreibt zuerst KIEL überblicksartig und geht dann etwas genauer auf seine graphische Benutzeroberfläche ein. In Abschnitt 4 wird ein bekanntes Beispiel mit KIEL behandelt. Abschließende Bemerkungen findet man schließlich in Abschnitt 5.

2 Die von KIEL unterstützte Teilsprache von ML

Die von KIEL unterstützte Teilsprache von ML, im folgenden MLKIEL genannt, entspricht in etwa dem Umfang, wie er in den Kapiteln 1 bis 5 von [9] beschrieben wird. Dieser soll nun skizziert werden. Wir verzichten auf eine formale Beschreibung. Die entsprechenden Syntaxdiagramme findet man in [8] und mit Ausnahme der Parameterübergabe, deren freie Wahl ja geradezu charakteristisch für KIEL ist, entspricht die Semantik aller Sprachkonstrukte der von ML.

Es gibt fünf primitive Typen in MLKIEL, nämlich den einelementigen Typ `unit` und die Typen `bool`, `int`, `real` und `string` für die Wahrheitswerte, ganzen Zahlen, reellen Zahlen und Zeichenreihen. Jeder primitive Typ m ist ein "Typ mit Gleichheit" im Sinne von ML, was heißt, daß Gleichheitstest `=` : $m * m$ `->bool` und Ungleichheitstest `<>` : $m * m$ `->bool` definiert sind.

Die Darstellung aller Konstanten ist wie in ML. Der Typ `unit` besitzt also nur `()` als Konstante, die Wahrheitswerte werden mit `true` und `false` bezeichnet, die Konstanten des Typs `int` sind Ziffernfolgen, ev. mit einer vorangestellten unären Negation "~", die Konstanten des Typs `real` beginnen mit einem optionalen "~" gefolgt von einer Ziffernfolge und dann entweder dem Dezimalpunkt und einer Ziffernfolge oder dem Symbol `E`, einem optionalen "~" und wiederum einer Ziffernfolge und die Konstanten des Typs `string` sind in Anführungszeichen eingeschlossene Zeichenreihen. Im Gegensatz zu ML erlaubt MLKIEL in Zeichenreihen keine speziellen Zeichen wie "newline". Die Maximallänge der Zeichenreihen kann mittels einer Konfigurationsdatei festgelegt werden.

In KIEL werden sequentielle Konjunktion und Disjunktion als Operationen auf den Wahrheitswerten eingeführt und nicht als Schreibweisen für spezielle Fallunterscheidungen. Dies führt zu

`not : bool -> bool`	Negation
`andalso, orelse : bool*bool->bool`	Konjunktion / Disjunktion

als den primitiven Operationen auf den Wahrheitswerten. Die primitiven Operationen auf den ganzen Zahlen, reellen Zahlen und Zeichenreihen bilden jeweils eine Teilmenge der entsprechenden ML-Operationen. Zur Behandlung von ganzen Zahlen bietet KIEL an:

`+, -, * : int*int->int`	Arithmetik
`div, mod : int*int->int`	Ganzzahl. Division mit Rest
`min, max : int*int->int`	Minimum / Maximum
`abs : int->int`	Absolutbetrag
`<, <=, >, >= : int*int->bool`	Vergleiche

Die primitiven Operationen des KIEL-Systems für die reellen Zahlen sind:

`+, -, *, / : real*real->real`	Arithmetik
`abs : real->real`	Absolutbetrag
`sqrt : real->real`	Quadratwurzel
`sin, cos : real->real`	Sinus / Cosinus
`exp, ln : real->real`	e-Funktion / natürlicher Log.
`real : int->real`	Konvertierung
`truncate : real->int`	Rundung nach unten
`<, <=, >, >= : real*real->bool`	Vergleiche

Auf Zeichenreihen kann man schließlich die folgenden Operationen verwenden:

`^ : string*string->string`	Konkatenation
`size : string->int`	Länge
`ord : string->int`	ASCII-Code Anfangsz.
`chr : int->string`	Inversion ASCII-Code
`substring : string*int*int->string`	Teilzeichenreihe
`<, <=, >, >= : string*string->bool`	Vergleiche (lexikog.)

Listen sind in funktionalen Programmen ein extrem hilfreicher Datentyp. Deshalb stellt ML einen primitiven polymorphen Datentyp für Listen mit vielen Operationen zur Verfügung. In MLKIEL gibt es (derzeit noch) keinen Polymorphismus. Wegen der Bedeutung der Listen haben wir uns jedoch dazu entschlossen, für jeden der obigen fünf Typen m den Typ m `list` für Listen mit Objekten des Typs m zu implementieren. Die Konstanten werden dabei wie in ML dargestellt, was heißt, daß die leere Liste mit `nil` oder `[]` bezeichnet wird und die Elemente einer nichtleeren Liste durch Kommata getrennt und mittels "`[`" und "`]`" geklammert werden. Neben dem Gleichheits- und Ungleichheitstest sind in MLKIEL noch die folgenden Listenoperationen vorhanden:

`::` : m * m `list -> `m `list`	Links anfügen
`hd` : m `list -> `m	Anfangselement
`tl` : m `list -> `m `list`	Entfern. Anfangselement
`@` : m `list` * m `list -> `m `list`	Konkatenation
`length` : m `list -> int`	Länge
`null` : m `list -> bool`	Test auf leere Liste
`rev` : m `list -> `m `list`	Spiegelung

Wie bei den Zeichenreihen kann auch die Maximallänge der Listen mittels der schon erwähnten Konfigurationsdatei festgelegt werden.

Ein Programm in MLKIEL besteht aus einer Deklarationsfolge, wobei jede Deklaration entweder eine einzelne Rechenvorschrift oder ein System von (möglicherweise verschränkt-rekursiven) Rechenvorschriften definiert und mit einem Semikolon abgeschlossen ist. Die Deklaration einer einzelnen Rechenvorschrift geschieht wie in ML mit der Ausnahme, daß, wegen des fehlenden Polymorphismus, eine MLKIEL-Rechenvorschrift vollständig typisiert sein muß. Dabei wird der Typ m eines formalen Parameters x innerhalb der Parameterliste in der Form x : m angegeben und der Resultattyp wird, ebenfalls mit dem Doppelpunkt als Trennzeichen, der Parameterliste nachgestellt. Insgesamt geschieht die Deklaration einer einzelnen Rechenvorschrift F mit formalen Parametern x_i des Typs m_i $(1 \leq i \leq k)$, Resultattyp m und Rumpf E in der folgenden Form:

$$\texttt{fun}\ F\ (x_1 : m_1, \ldots, x_k : m_k) : m = E\ ;$$

Dabei ist der Term E in der üblichen Weise aufgebaut mittels der formalen Parameter, der obigen Konstanten und Operationen, Aufrufen von schon deklarierten Rechenvorschriften, Fallunterscheidungen der Form `if` B `then` A_1 `else` A_2 und rekursiven Aufrufen von F. Die Anwendungen der Operationen werden wie in ML notiert mit der (aus methodischen Gründen) einzigen Ausnahme, daß, wenn verwendet, MLKIEL grundsätzlich die klassische mathematische Notation $F(A)$ für die Applikation verlangt und die in ML auch zulässigen Formen $F\,A$ oder $(F\,A)$ verbietet.

Es sollte ausdrücklich erwähnt werden, daß die derzeitige Version von KIEL weder höhere funktionale Konstrukte noch Deklarationen von Datentypen, Muster in Rechenvorschriften und lokale Deklarationen erlaubt.

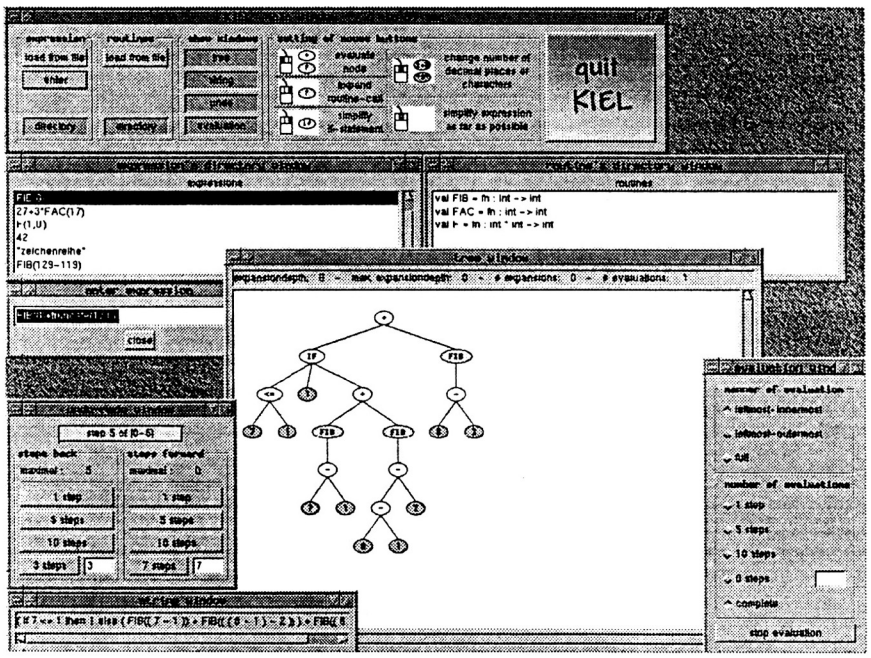
Nachfolgend ist ein einfaches Beispiel eines MLKIEL-Programms angegeben, wobei ein Aufruf von `sort` eine Liste von ganzen Zahlen sortiert. Dies geschieht unter Verwendung einer Hilfsrechenvorschrift `insort`, die ein Element in eine sortierte Liste einsortiert.

```
fun insort (y : int, w : int list) : int list =
  if null(w) then [y]
             else if y <= hd(w) then y :: w
                                else hd(w) :: insort(y,tl(w));
fun sort (s : int list) : int list =
  if null(s) then s
             else insort(hd(s),sort(tl(s)));
```

Die Syntax von Systemen von Rechenvorschriften stimmt mit der von ML überein. Dies heißt, daß das Schlüsselwort `fun` bei allen Rechenvorschriften mit Ausnahme der ersten weggelassen wird und `and` zur Trennung dient.

3 Struktur und Benutzerschnittstelle von KIEL

Das Computerssystem KIEL ist in C geschrieben und verwendet die Bibliotheken GTK, GDK und GLib für die graphische Benutzerschnittstelle. Obwohl der Benutzer die Oberfläche des Systems mittels einer Konfigurationsdatei in weiten Teilen selbst gestalten kann (vergl. [8] wegen der Details), zeigt das nachfolgende Bild eine typische Benutzeroberfläche mit den wichtigsten Fenstern.

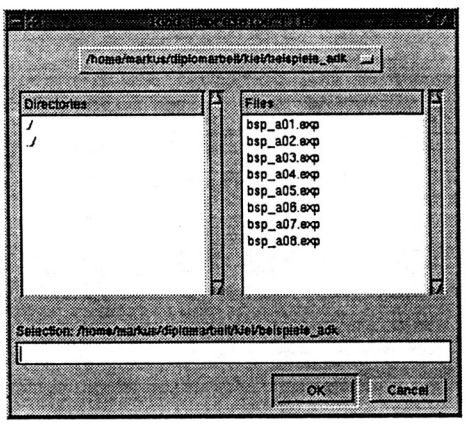


Aus der Sicht des Benutzers besteht KIEL, grob gesehen, aus zwei Komponenten. Die erste Komponente ist der Arbeitsbereich. Er enthält MLKIEL-Programme und Terme über ihnen. Dabei ist angenommen, daß die Programme in Dateien mit der Endung ".prg" enthalten sind. Aus diesen werden sie in das System mit Hilfe des Fensters "load program file" geladen. Auch Terme können aus Dateien mit der Endung ".exp" in KIEL geladen werden; das entsprechende Fenster hat den Namen "load expression file". Sie werden jedoch üblicherweise direkt mit Hilfe des Fensters "enter expression" eingegeben.

Die zweite Komponente des Systems ist die Auswertungseinheit. Durch sie werden Terme des Arbeitsbereichs berechnet. Dies geschieht hauptsächlich mit Hilfe des Fensters "tree window". Wir haben diesen Namen gewählt, weil Terme graphisch als Bäume dargestellt werden. Die Auswertung kann schrittweise, semiautomatisch oder vollautomatisch erfolgen. Ersteres geschieht mittels einzelner Mausklicks im Fenster "tree window"; in den beiden anderen Fällen kommen noch zusätzliche Fenster ins Spiel, mit denen beispielsweise eine spezielle Berechnungsregel und die maximale Schrittanzahl festgelegt werden. Auf Details gehen wir später noch genauer ein.

Wir beschreiben nun die Benutzeroberfläche des KIEL-Systems etwas genauer und beginnen mit dem Hauptfenster "KIEL main window". Wie man der obigen Gesamtübersicht entnehmen kann, ist es konzeptuell in drei Teile gegliedert. Der linke Teil enthält drei Spalten von Bedienknöpfen, welche zum Laden bzw. zur Eingabe von Termen, zum Laden von Programmen und zum Öffnen bzw. Schließen der vier mit der Auswertung zu tun habenden Fenster dienen. Im mittleren Teil findet man fünf kleine Bilder, welche die verschiedenen Aktionen beschreiben, die durch das Drücken von Maustasten ausgelöst werden. Der rechte Teil des Hauptfensters besteht nur aus einem einzigen großen Knopf zum Beenden von KIEL.

Wenn man den Knopf "load from file" der linken Knopfspalte des Hauptfensters betätigt, so öffnet sich das Fenster "load expression file", mit dem man Terme aus Dateien ins System laden kann. Da dieses Fenster in der obigen Gesamtübersicht nicht geöffnet ist, zeigen wir in dem nachfolgenden Bild, wie es typischerweise aussieht:



Die linke Liste dieses Fensters enthält die Unterverzeichnisse und die rechte Liste die exp-Dateien des Verzeichnisses, in dem man sich gegenwärtig befindet. Mit der Maus ist es möglich, in ein anderes Verzeichnis zu wechseln oder eine Datei auszuwählen; der Ladevorgang wird durch Betätigung von "OK" ausgelöst. Üblicherweise erfolgt jedoch die Eingabe der auszuwertenden Terme unter Verwendung des Fensters "enter expression", welches durch den Knopf "enter" geöffnet wird und sich in der obigen Gesamtübersicht in der Mitte am linken Bildrand befindet. Der Knopf "directory" der linken Knopfspalte des Hauptfensters öffnet schließlich ein Fenster, welches die derzeitigen Terme des Arbeitsbereichs zeigt. Dieses Fenster sieht man in der obigen Gesamtübersicht direkt über dem Fenster "enter expression".

Das Laden von Programmen aus prg-Dateien ins System geschieht analog zum Laden von Termen mit Hilfe des schon erwähnten Fensters "load program file", das durch den Knopf "load from file" der mittleren Knopfspalte des Hauptfensters geöffnet wird. Der Knopf "directory" dieser Spalte hat für Programme die gleiche Bedeutung wie der entsprechende Knopf der linken Spalte für Terme.

Der primäre Zweck von KIEL ist die benutzerkontrollierte interaktive Auswertung von funktionalen Programmen. Dies geschieht mit Hilfe des Fensters "tree window", das sich öffnet, wenn man den Knopf "tree" im Hauptfenster betätigt. Die Auswahl des auszuwertenden Terms aus der Liste aller Terme des Arbeitsbereichs geschieht mit der Maus und stellt diesen anschließend im Fenster "tree window" als Baum dar. In der Gesamtübersicht ist ein Baum zu sehen, der sich während der Auswertung einer Rechenvorschrift FIB für die Fibonacci-Zahlen ergeben hat. Ein Knoten eines Baums steht für eine Konstante bzw. Operation aus Abschnitt 2, eine Fallunterscheidung, eine Rechenvorschrift oder einen Auswertungsfehler. Die nächsten vier Bilder zeigen für jede von diesen vier Klassen ein Beispiel. Vielleicht fallen dem Leser die unterschiedlichen Grauwerte auf; bei einem Farbbildschirm entsprechen diese vier unterschiedlichen Farben, die vom Benutzer mittels einer Konfigurationsdatei frei wählbar sind.



Normalerweise ist die Größe der Knoten fest, und damit werden von großen Zahlen und langen Zeichenreihen und Listen nur die Anfangsstücke angezeigt. Mit Hilfe der mittleren Maustaste kann jedoch die Größe der Knoten der Größe der Objekte angepaßt werden, für die sie stehen.

Wie man der obigen Gesamtübersicht auch entnehmen kann, wird der Term des Fensters "tree window" im Fenster "string window" zusätzlich noch in der gewohnten textuellen Darstellung angegeben. Dieses letztgenannte Fenster wird mittels des Knopfs "string" der rechten Spalte des Hauptfensters geöffnet.

Die semi- oder vollautomatische Auswertung eines (Teil-)Terms geschieht, indem man mit der Maus auf den entsprechenden Wurzelknoten zeigt und anschließend die linke Maustaste drückt. Bei diesen Vorgehensweisen ist es zusätzlich möglich, eine spezielle Berechnungsregel und die maximale Anzahl der Auswertungsschritte mittels des Fensters "evaluation window" einzustellen, das durch den untersten Knopf "evaluation" der rechten Knopfspalte des Hauptfensters

geöffnet wird und sich in der Gesamtübersicht in der rechten unteren Ecke befindet. Die derzeitige Version von KIEL erlaubt die Auswahl zwischen genau den in der Einleitung genannten Berechnungsregeln, also zwischen Leftmost-innermost-Ersetzung, Leftmost-outermost-Ersetzung und voller Ersetzung.

Nachdem wir die semi- oder vollautomatische Auswertung von Termen mittels KIEL beschrieben haben, wenden wir uns nun deren schrittweiser Auswertung zu. Diese ist durch drei Aktionen möglich, nämlich Expandieren von Rechenvorschriftsaufrufen, Ausführen von Operationen von Abschnitt 2 und Vereinfachen von Fallunterscheidungen. Wie diese drei Aktionen angestoßen werden, wird durch die Bilder im mittleren Teil des Hauptfensters erklärt. Beispielsweise wird eine Expansion angestoßen, indem man mit der Maus auf den entsprechenden Rechenvorschriftsknoten zeigt und dann die rechte Maustaste drückt; man vergleiche mit dem mittleren Bild der linken Bilderspalte. Wie man aus der Gesamtübersicht ersehen kann, gibt es zusätzlich zu den eben aufgeführten drei Aktionen noch die Möglichkeit, den gesamten Term des Fensters "tree window" soweit wie möglich auszuwerten. Dazu hat man, wie das untere Bild der rechten Bilderspalte angibt, mit der Maus auf eine Position außerhalb des Baums zu zeigen und dann die mittlere Maustaste zu drücken. In Verbindung mit Expansionen, die jeweils in einer speziellen Weise durch die Auswertung des Gesamtterms mit eins als maximaler Schrittanzahl realisiert sind, kann man damit unmittelbar die operationale Semantik mittels Berechnungssequenzen visualisieren, die in der Einleitung skizziert wurde.

Schließlich sollte noch erwähnt werden, daß KIEL erlaubt, Auswertungen zu stoppen und Schritte rückgängig zu machen. Letzteres kann insbesondere benutzt werden, um durch eine Berechnungssequenz "zu wandern", beispielsweise zu Zwecken der Visualisierung oder um alternative Berechnungen zu testen. Das entsprechende "undo-redo window" wird durch die Betätigung des dritten Knopfs "undo" der rechten Knopfspalte des Hauptfensters geöffnet.

4 Ein Beispiel zur Anwendung von KIEL

In diesem Abschnitt verwenden wir eine sehr bekannte funktionale Rechenvorschrift, um zu demonstrieren, wie KIEL im Grundstudium eingesetzt werden kann, um Programmauswertungen zu visualisieren und die Unterschiede zwischen den Berechnungsregeln hervorzuheben.

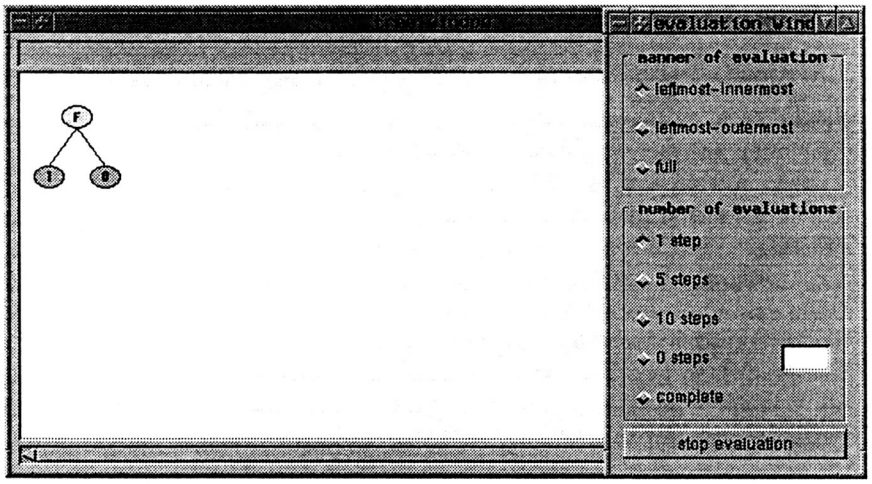
Die folgende geschachtelt-rekursive Rechenvorschrift wurde erstmals im Jahre 1968 von J.H. Morris im Rahmen seiner Dissertation [7] untersucht:

```
fun F (x : int, y : int) : int =
  if x <= 0 then 1
            else F(x-1,F(x-y,y));
```

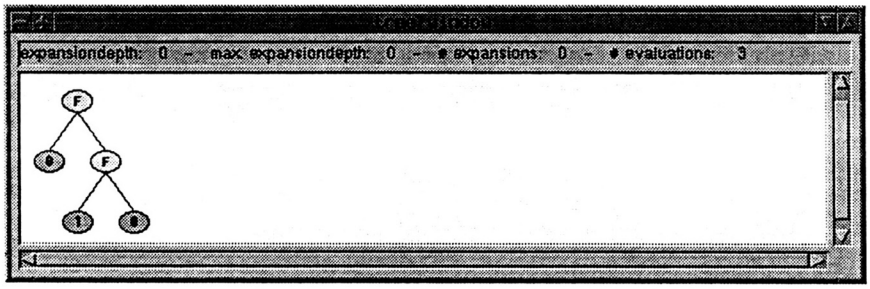
Seither ist bekannt, daß verschiedene Berechnungsregeln zu verschiedenen Resultaten führen können. Beispielsweise führt der Term `F(1,0)` zu einer unendlichen Berechnungssequenz, liefert also "undefiniert" als Resultat, wenn Leftmost-innermost-Ersetzung angewendet wird. Ist durch die Berechnungsregel hingegen

Leftmost-outermost-Ersetzung oder volle Ersetzung vorgeschrieben, so wertet sich `F(1,0)` zu `1` aus. Nachfolgend zeigen wir, wie man KIEL verwenden kann, um diesen Unterschied zu demonstrieren.

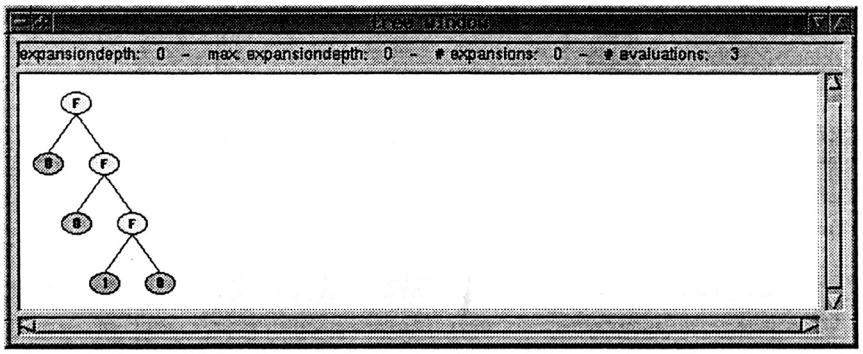
Wir starten mit dem auszuwertenden Term `F(1,0)` und wählen zuerst Leftmost-innermost als Ersetzungsstrategie und, da die Berechnung gemäß der in der Einleitung skizzierten Weise erfolgen soll, eins als maximale Schrittanzahl bei einer Auswertung. Das nachfolgende Bild zeigt die Fenster "tree window" und "evaluation window" von KIEL am Beginn der Auswertung:



Die ersten zwei Schritte des Auswertungsprozesses bestehen aus einer Expansion des Aufrufs `F(1,0)` und einer nachfolgenden Vereinfachung des durch sie entstehenden Terms soweit wie möglich. Das Ergebnis ist `F(0,F(1,0))` und dieser Term wird durch KIEL wie folgt im Fenster "expression window" dargestellt:

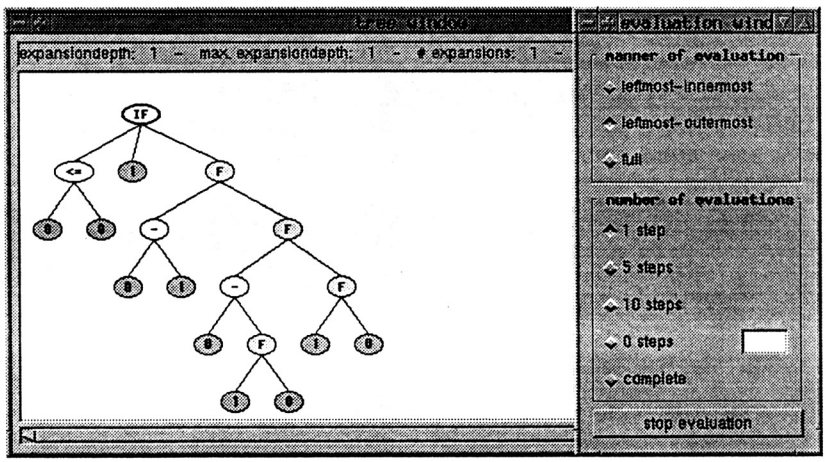


Leftmost-innermost-Ersetzung bedeutet, daß von allen Aufrufen einer Rechenvorschrift, deren Argumente keine rekursiven Aufrufe mehr enthalten, der linkeste (im textuellen Aufschreibungssinne) gewählt wird. Im vorliegenden Fall haben wir also den unteren der beiden Aufrufe zu expandieren. Führen wir dies durch und vereinfachen wir anschließend wiederum soweit wie möglich, so präsentiert uns KIEL den folgenden Baum:

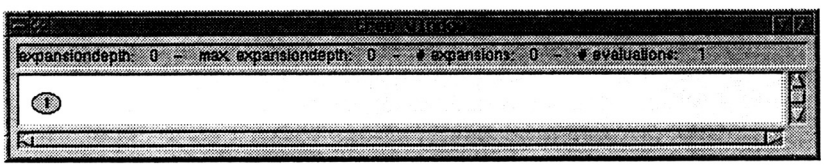


Schon nach diesen wenigen Schritten ist klar, daß eine Leftmost-innermost-Ersetzung zu einer unendlichen Berechnungssequenz führt.

Nun werten wir `F(1,0)` nach der Leftmost-outermost-Strategie aus. Dazu stellen wir zuerst diese Ersetzungsart ein und gehen dann an den Anfang der bisher aufgebauten Berechnungssequenz zurück. Von dort ausgehend sind die erste Expansion und Vereinfachung genau wie bei der Leftmost-innermost-Ersetzung. Dann verändert sich jedoch die Sachlage und wir zweigen im sogenannten Berechnungsdiagramm (siehe [6]) von der ersten Berechnungssequenz ab. Expandieren wir nämlich in `F(0,F(1,0))` den äußeren Aufruf von `F`, wählen also den oberen `F`-Knoten, so sehen wir in dem Fenster "tree window" den folgenden Baum (wobei, wegen der Einstellung, auch das "evaluation window" mit angeben ist):



Aufgrund der nichtstrikten Semantik der Fallunterscheidung liefert eine vollständige Vereinfachung die Konstante `1` und diese wird auch von KIEL berechnet:



Eine Besonderheit von KIEL ist, daß das System während einer semi- oder vollautomatischen Auswertung die Anzahl der Expansionen und Ausführungen von Operationen bzw. Fallunterscheidungen mitzählt. Dies erlaubt es insbesondere den Studierenden, die Effizienz von Berechnungsregeln zu bewerten. Intuitiv scheint klar zu sein, daß Leftmost-innermost-Ersetzung effizienter ist als Leftmost-outermost-Ersetzung, da sie viele redundante Berechnungen vermeidet. Jedoch weist schon J.H. Morris in [7] darauf hin, daß Leftmost-innermost-Ersetzung nicht immer überlegen sein muß. Und tatsächlich ist die auf ihn zurückgehende Rechenvorschrift `F` ein Gegenbeispiel. Die nachfolgende Tabelle zeigt einige Testergebnisse, die mittels KIEL erzielt wurden. In der zweiten und vierten Spalte sind dabei die zur Berechnung des Terms der ersten Spalte durchgeführten Expansionen unter Verwendung von Leftmost-innermost- bzw. Leftmost-outermost-Ersetzung angegeben und in der dritten und vierten Spalte die entsprechenden Ausführungen von Operationen auf den Typen.

	LI-Exp.	LI-Ausf.	LO-Exp.	LO-Ausf.
`F(5,1)`	63	125	6	21
`F(5,3)`	37	73	6	21
`F(5,5)`	33	65	6	21
`F(10,1)`	2047	4093	11	66
`F(10,5)`	1057	2113	11	66
`F(10,10)`	1025	2049	11	66

Wir haben auch einige Tests mit voller Ersetzung durchgeführt. Dabei wurden die Terme aber bald so riesig, daß beispielsweise auf einer modernen SUN ULTRA Workstation die Auswertung von `F(3,0)` mehr als 20 Minuten dauerte.

5 Abschließende Bemerkungen

In diesem Artikel wurde das Computersystem KIEL vorgestellt, das es erlaubt, funktionale Programme erster Stufe in ML-Syntax interaktiv auszuwerten, und es wurde seine Anwendung anhand eines bekannten Beispiels demonstriert. KIEL wurde für den Gebrauch im Informatik-Grundstudium entwickelt und ist primär vorgesehen für die Visualisierung von Programmauswertungen und die Auswirkungen von Berechnungsregeln. Zu einem gewissen Grad kann das System aber auch im Hauptstudium oder zur allgemeinen Programmierung verwendet werden, etwa um ML-Programme zu testen und/oder debuggen, sofern sie nach MLKIEL reduzierbar sind.

KIEL kann auf allen SUN Workstations mit SUN-OS 5.x installiert werden, sowie auf allen PCs mit INTEL-Prozessoren und LINUX, Kernel 2.2.13. Es ist via FTP frei erhältlich vom Server ftp.informatik.uni-kiel.de, wo man es im Verzeichnis pub/kiel/progdev/KIEL findet. Zusätzliche Information wird mittels der Web-Seite http://www.informatik.uni-kiel.de/~progsys/kiel.html verbreitet. Durch diese Web-Seite hat man auch Zugang zu einer Online-Version von KIEL. Diese steht allen Benutzern des Internets als Java-Applet zur Verfügung und

wurde, als beispielhafte Anwendung einer allgemeinen Methode, die in der Diplomarbeit [5] beschrieben ist, aus der in diesem Artikel beschriebenen C-Version entwickelt.

Seit der Fertigstellung von KIEL wurden mit dem System viele Fallstudien durchgeführt, u.a. auch bei einer Fortbildungsveranstaltung für Gymnasiallehrer. Dabei zeigte sich, daß das System eine große Hilfe ist bei einer Einführung in funktionale Programmierung (oder sogar in Informatik, wenn man den Ideen von [2] folgt). Aufgrund der gemachten positiven Erfahrungen glauben wir, daß eine Visualisierung von Programmauswertungen die Studierenden effizienter in die teils doch etwas verzwickte Materie der operationalen Semantik und Parameterübergabe einführt als Erklärungen an der Tafel oder in Lehrbüchern. Gleiches gilt etwa auch für Algorithmenentwurfsprinzipien (eine Visualisierung von Quicksort macht z. B. jedem sofort klar, wie das Prinzip "Teile und herrsche" funktioniert) und die Komplexität von Berechnungen (hierzu werte man etwa die Fibonacci-Rekursion mittels KIEL schrittweise aus).

Die Erfahrungen haben aber auch gezeigt, daß die Programmiersprache von KIEL zu restriktiv ist für fortgeschrittenere funktionale Programmiertechniken wie partielle Auswertung, Tabellierung, Supercompilation und Deforestation. Für die Zukunft ist deshalb geplant, MLKIEL um Sprachkonstrukte wie Funktionsabstraktion, Deklaration von (rekursiven) Datentypen, Muster in Rechenvorschriften und lokale Objekte zu erweitern. Als ein erster Schritt in diese Richtung werden derzeit im Rahmen der Diplomarbeit [1] Deklarationen von (rekursiven) Datentypen und Muster in Rechenvorschriften in MLKIEL integriert.

Literatur

1. Ernst H.: Erweiterung des KIEL-Systems um benutzerdefinierte Typen und Rechenvorschriften mit Mustern. Diplomarbeit, Inst. für Informatik und Praktische Mathematik, Universität Kiel (im Erscheinen)
2. Giegerich R., Hinze R., Kurtz S.: Straight to the heart of computer science via functional programming. In: Felleisen M., Hanus M., Thompson S. (eds.): Proc. FDPE '99, Report Rice COMP TR99-346, Dept. of Computer Science, Rice University, 1-5 (1999)
3. Goldson D.: A symbolic calculator for non-strict functional programs. The Computer Journal 37, 177-187 (1994)
4. Kluge W.E., Rathsack C., Scholz S.-B.: Using π-RED as a teaching tool for functional programming and program execution. In: Hartel P.H., Plasmeijer M.J. (eds.): Proc. FPLE '95, LNCS 1022, Springer, 231-250 (1995)
5. Lorenz O.: Vom C Programm zum Applet am Beispiel von KIEL. Diplomarbeit, Inst. für Informatik und Praktische Mathematik, Universität Kiel (2000)
6. Manna Z.: Mathematical theory of computation. McGraw-Hill (1974)
7. Morris J.H.: Lambda-calculus models of programming. Ph.D. Thesis, Mass. Inst. of Technology (1968)
8. Tiedt M.: Kiel interactive evaluation laboratory: Eine Arbeitsumgebung zur Visualisierung von Termersetzungssemantik. Diplomarbeit, Inst. für Informatik und Praktische Mathematik, Universität Kiel (1999)
9. Ullman J.D.: Elements of ML programming. Prentice Hall (1994)

Generierung interaktiver Animationen von Berechnungsmodellen

Stephan Diehl, Andreas Kerren

Universität des Saarlandes, FB 14 Informatik
Postfach 15 11 50, D-66041 Saarbrücken
{diehl,kerren}@cs.uni-sb.de

Zusammenfassung In diesem Artikel wird ein generativer Ansatz für animierte Berechnungsmodelle vorgestellt. Zuerst werden die technischen und pädagogischen Ziele diskutiert, dann wird der Ansatz im Kontext einer Lernsoftware für den Compilerbau angewandt und die Implementierung eines ersten Prototypen beschrieben. Basierend auf der Erfahrung mit diesem Prototypen wurde das GANIMAL Framework entworfen. Es handelt sich dabei um ein generisches Algorithmenanimationssystem, das eine für ein solches System einzigartige Fülle an Möglichkeiten durch sein graphisches Basispaket, das nebenläufige Laufzeitsystem mit graphischer Benutzeroberfläche und die Visualisierungskontrollsprache GANILA bietet.

1 Einleitung

Heutige *Lernsoftware* deckt größtenteils Themen außerhalb der Informatik ab und zielt meist auf die Vermittlung von Faktenwissen und weniger auf das Verständnis komplexer Abläufe. Hierzu zählen elektronische Bücher, Lexika und Wörterbücher. Schaut man sich z.B. die von Schulmeister beschriebene Lernsoftware [13] (u.a. 16 interaktive, hypermediale Lernprogramme und 23 tutorielle Programme) genauer an, so stellt man fest, daß nur eine geringe Zahl von diesen komplexe Vorgänge darstellen kann. Dies sind vor allem Systeme im Bereich der Physik. Im Gebiet der Informatik überwiegen Kurse für Programmiersprachen. Nur wenige Systeme behandeln Themen der theoretischen Informatik [1].

Häufig tritt im Informatikunterricht das Problem auf, daß der Lehrer ein dynamisches System nur unzureichend auf Papier, Folien oder an der Tafel darstellen kann. Er muß die Dimension Zeit in der Dimension Raum abwickeln, d.h. Auswischen, Überschreiben, Übereinanderlegen von Folien oder die Verwendung aufeinanderfolgender Diagramme, Tabellen oder Abbildungen. Für diesen Zweck ist die Computeranimation, insbesondere Algorithmenanimation, das Mittel der Wahl. Der größte Teil der Arbeiten zur *Algorithmenanimation* lassen sich grob in folgende Kategorien einteilen:

1. Darstellung einfacher Graphenalgorithmen [12].
2. Geometrische Algorithmen werden mit komplexen Basisprozeduren beschrieben, die sich direkt auf graphische Operationen abgebilden lassen (siehe z.B. GASP [17]).

3. Andere entwickeln meist einfache Animationssprachen, mit denen Algorithmen annotiert werden können (siehe z.B. Zeus, Samba, Polka, XTango [14, 15]). Zum Beispiel würde die Zuweisung `x:=y` annotiert mit `moveto(obj(y), obj(x))`. So kann z.B. der Entwickler einer Animation in Zeus [2, 3] graphische Sichten (views) auf Daten definieren und im Algorithmus geeigneten Stellen mit sogenannten 'interesting events' annotieren. Während der Ausführung eines annotierten Programmpunktes werden diese Ereignisse an alle Sichten geschickt, die dann darauf reagieren können.

Im Folgenden betrachten wir Algorithmenanimationen in Zusammenhang mit Lernsoftware für den Compilerbau. Dort spielt die Visualisierung von Generatoren eine wesentliche Rolle. Wir stellen diesbezüglich in Abschnitt 3 zwei Methoden vor. Dann betrachten wir eine Implementierung, die auf der ersten Methode beruht, analysieren diese und entwerfen ein Framework, das als Grundlage für die zweite Methode verwendet werden kann.

1.1 Technische Ziele

Im Software-Engineering gewinnen generative und generische Techniken immer mehr an Bedeutung. Im Projekt GANIMAL [1] entwickeln wir Generatoren, die interaktive Visualisierungen und Animationen verschiedener Compilerphasen, aber auch anderer Algorithmen erzeugen. Diese sollen Teil einer web-basierten Lernsoftware für den Compilerbau werden. Die Ergebnisse, d.h. insbesondere die Beschreibungssprache und das Basispaket, könnten in späteren Projekten auch zur Entwicklung von Generatoren für andere prozeßorientierte Lerninhalte, wie z.B. Algorithmen, Rechnerarchitektur, chemische Reaktionen oder gar der Modellierung ökonomischer Prozesse eingesetzt werden.

1.2 Pädagogische Ziele

Der generative Ansatz erlaubt neue Übungsformen. Als Teil einer Übungsaufgabe schreibt der Lerner Spezifikationen von Vorgängen oder Berechnungsmodellen. In konventioneller Lernsoftware werden solche Antworten oft auf Korrektheit geprüft. Fehler werden dem Lerner angezeigt und er kann seine Antwort überarbeiten. Als Folge des Halteproblems ist jedoch die automatische Überprüfung der Korrektheit für viele Aufgaben, die sich auf Berechnungsmodelle beziehen, nicht möglich. Daher werden in unserem Ansatz interaktive Animationen aus der eingegebenen Spezifikation des Lerners erzeugt. Dann kann er diese mit Hilfe eigener oder vorgegebener Beispiele testen. Auf diese Weise kann er eigenständig Fehler entdecken. Die Software tritt in diesem Fall nicht als nüchterne, allwissende Autorität auf, die ihm seine Fehler zeigt.

Viele Autoren sind der Meinung, daß Lernsoftware, in der der Computer als Korrektor auftritt, entmutigend wirkt und daher wenig erfolgreich ist. Die

[1] Das Projekt wird von der DFG unter Aktenzeichen WI 576/8-1 und WI 576/8-3 gefördert.

existierenden Untersuchungen hierzu sind jedoch teilweise widersprüchlich, siehe [13]. Unser Ansatz ermöglicht exploratives, selbst gesteuertes Lernen. Der Lerner kann bestimmte Aspekte in den erzeugten, interaktiven Animationen fokussieren und herausfinden, welche Auswirkungen kleine Änderungen in der Spezifikation haben. Aufgrund solcher Beobachtungen kann er Hypothesen formulieren und diese empirisch überprüfen. Dabei soll jedoch nicht das mathematische Beweisen von Hypothesen ersetzt, sondern ein tieferes Verständnis der Berechnungsmodelle gefördert und damit eine spätere Beweisführung erleichtert werden [7].

Die Akzeptanz und Effektivität einer solchen explorativen Lernsoftware kann nur in der Praxis, d.h. im Unterricht, belegt werden. In Zusammenarbeit mit kognitiven Psychologen haben wir einige Lernversuche durchgeführt und entwickeln derzeit neue.

2 Lernsoftware für den Compilerbau

In der Informatik und insbesondere im Compilerbau sind die Theorie und Algorithmen sehr abstrakt und meist auch komplex. Daher ist ihre visuelle Darstellung ein wichtige Hilfe im Informatikunterricht [10, 11]. Obgleich der Compilerbau meist als praktisches Gebiet innerhalb der Informatik gesehen wird, beruhen seine Techniken auf Ergebnissen der theoretischen Informatik wie z.B. formalen Sprachen, Automatentheorie und formaler Semantik.

Im Compilerbau werden Techniken entwickelt, mit denen Programme höherer Programmiersprachen in effiziente und korrekte Programme einer realen oder abstrakten Maschinen übersetzt werden können [18]. Die Übersetzung eines Programms kann in verschiedene Phasen unterteilt werden. Im GANIMAL Projekt werden Generatoren entwickelt, die Visualisierungen und interaktive Animationen aus Spezifikationen dieser Compilerphasen erzeugen. In der folgenden Tabelle werden Compilerphasen, ihre Spezifikationssprachen und zugrundeliegende Berechnungsmodelle aufgeführt:

Compilerphase	**Spezifikationssprache**	**Berechnungsmodell**
Lexikalische Analyse	Reguläre Ausdrücke	Endliche Automaten
Syntaktische Analyse	Kontextfreie Grammatiken	Item-Kellerautomaten
Semantische Analyse	Attributgrammatiken	Attributauswerter
Codeerzeugung	Baumgrammatiken	Baumautomaten
Optimierung	Gleichungssysteme	Fixpunktlöser
Laufzeitsystem	**Spezifikationssprache**	**Berechnungsmodell**
Abstrakte Maschine	Kontrollflußsprache	Maschine mit Kellern, Halde und Registern

Häufig werden abstrakte Maschinen als plattformunabhängige Zwischenarchitekturen bei der Übersetzung höherer Programmiersprachen verwendet. Die Anweisungen einer abstrakten Maschine sind für die Übersetzung einer bestimmten Quellsprache oder für Quellsprachen des selben Sprachenparadigmas (impe-

rativ, funktional, logisch, objektorientiert) maßgeschneidert. In Abbildung 3 ist eine interaktive Animation einer abstrakten Maschine zu sehen.

Es gibt nur wenige Arbeiten zur Interaktion von Programmen und multimedialen Dokumenten, d.h. wie Programme zur Betrachtungszeit eines Dokuments Teile des Dokuments modifizieren oder gar ganz neu erzeugen können. Das DOM (Document Object Model) ist ein erster Versuch zur Standardisierung einer eingeschränkten Programmierschnittstelle für diesen Zweck. Der Bedarf ist groß, da Technologien wie Java-Applets, Dynamic HTML, XML oder Schnittstellen wie das EAI (External Authoring Interface) von VRML, Netscape's LiveConnect, derzeit völlig unterschiedliche Ansätze zur Kommunikation mit dem Rest eines Dokuments verfolgen.

3 Die generativen Methoden von GANIMAL

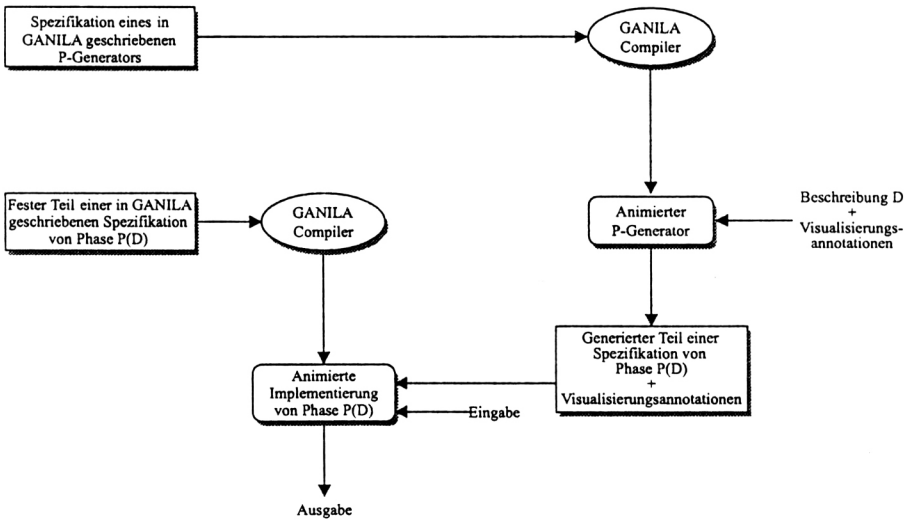
3.1 Methode I: Erweitere existierende Generatoren

Eine etablierte Spezifikationssprache für eine Compilerphase, wie z.B. reguläre Ausdrücke für die lexikalische Analyse, wird um Animationsannotationen erweitert. Zusätzlich werden Komponenten des Berechnungszustands mit graphischen Strukturen (Fenster, Graphen, Textfelder, ...) assoziiert. Dann erzeugt ein erweiterter Generator eine interaktive Animation der Compilerphase aus einer annotierten Spezifikation. Abschnitt 4 stellt eine Implementierung vor, die auf dieser Methode beruht.

3.2 Methode II: Implementiere Generatoren in einer Visualisierungssprache

Die erste Methode erlaubt es nicht, den Generator selbst zu visualisieren. In der zweiten Methode nutzen wir eine Eigenschaft aus, die viele Generatoren im Compilerbau haben. Diese Generatoren erzeugen Tabellen, die zusammen mit einem festen Treiber die Implementierung einer Compilerphase ergeben. Zuerst wurde eine Visualisierungskontrollsprache GANILA (siehe Abschnitt 6) entwickelt, in der die Generatoren und Treiber spezifiziert werden. Der GANILA Compiler erzeugt daraus eine Implementierung des Generators und des Treibers.

Aus der erweiterten Spezifikation erzeugt der GANILA Compiler eine Animation des Generators und der generierten Compilerphase, siehe Abbildung 1. Zusätzlich zur Annotierung der Spezifikation der Compilerphase, wie in der ersten Methode beschrieben, annotieren wir die Spezifikation des Generators und des Treibers durch Markieren von Programmpunkten mit sogenannten 'interesting events' und Definition von Sichten auf deren Datenstrukturen, d.h. u.a. die generierte Tabelle. Für jede Sicht muß festgelegt werden, wie sie auf jeden Event reagiert. Auf dieser Methode beruhen die interaktiven, animierten Visualisierungen im elektronischen Textbuch zur Theorie der Generierung endlicher Automaten [9].



Als Beispiel betrachten wir einen lexikalischen Analysatorgenerator. P ist die lexikalische Analyse und D eine regulärer Ausdruck. Der feste Teil ist der Treiber für die lexikalische Analyse, der generierte Teil ist eine Lookup-Tabelle und die Implementierung von P(D) ist ein lexikalischer Analysator, auch Scanner genannt.

Abbildung1. Generierung animierter Generatoren und Compilerphasen

4 Ein erster Prototyp: GANIMAM

Als Beispiel der ersten Methode betrachten wir die Entwicklung von GANIMAM, unseres web-basierten Generators für interaktive Animationen abstrakter Maschinen [5, 6]. Abbildung 3 zeigt eine Momentaufnahme einer solchen Animation. Im Folgenden beschreiben wir, wie GANIMAM verwendet werden kann und was das System generiert. Abschließend diskutieren wir dann die Vorteile von GANIMAM und der erzeugten interaktiven Animationen sowohl als Entwicklungswerkzeug, als auch als Teil einer Lernsoftware.

4.1 Technischer Überblick

GANIMAM kann über die Webseite des GANIMAL-Projekts [8] aufgerufen werden. Der Benutzer gibt die Spezifikation einer abstrakten Maschine ein, die dann zum Server geschickt wird. Ein CGI-Skript auf dem Server erzeugt Java-Quellcode, der durch einen Java-Compiler in Klassendateien übersetzt wird. In Kombination mit den Klassen des Basispakets von GANIMAM ergeben die Klassendateien ein interaktives Java-Applet, siehe Abbildung 2.

Das Applet kann über das Internet geladen werden und der Benutzer kann dann Maschinenprogramme eingeben, das Layout der verschiedenen Teile der

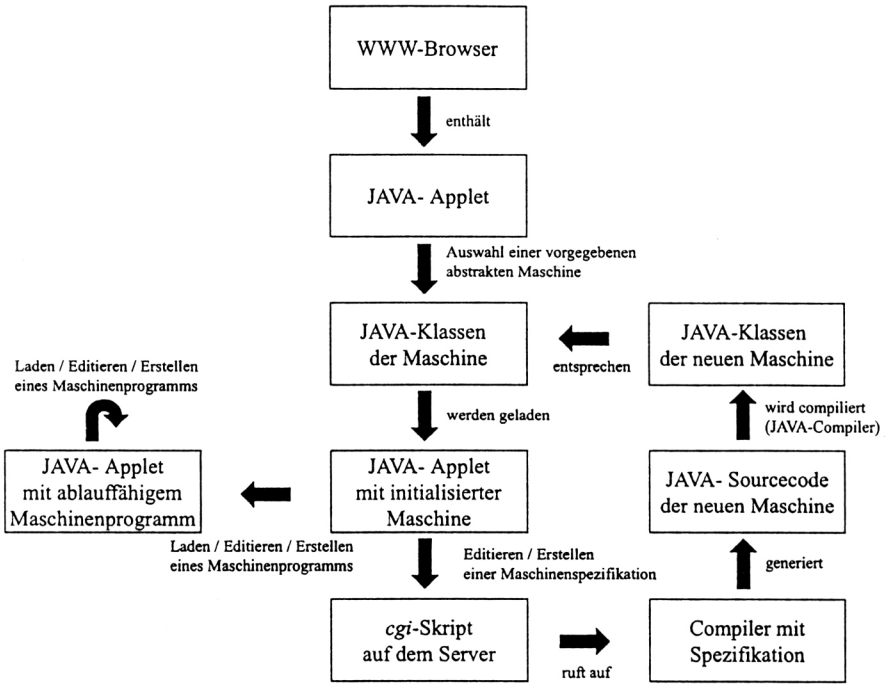


Abbildung2. Interaktion der Systemkomponenten

visualisierten abstrakten Maschine ändern und die animierte Ausführung seines Maschinenprogramms steuern.

Abstrakte Maschinen können eine unterschiedliche Anzahl von Registern, Kellern, Halden haben, daher wird für jede generierte Maschine ein automatisches Layout benötigt. Das automatische Layout gruppiert die verschiedenen Speicherarten um den Akkumulator, eine konzeptionelle Recheneinheit, herum (der Chip in der Mitte von Abbildung 3). Programmcode und Keller werden links, die Halde rechts, lokale Variablen über und Register unter dem Akkumulator plaziert. Mit dem Akkumulator ist ein Akkumulatorfenster verbunden, das den Ausdruck anzeigt der gerade im Akkumulator berechnet wird, sowie die Definition der Instruktion bzw. Funktion, die gerade ausgeführt wird. Durch einen zweifachen Mausklick mit der rechten Maustaste auf eine Instruktion im Programmcode wird die Definition der Instruktion in das Akkumulatorfenster geladen. Ein zweifacher Mausklick mit der linken Maustaste auf eine Instruktion im Programmcode setzt den Wert des Programmzählers auf die Adresse der Instruktion, d.h. die Ausführung des Programms wird an dieser Stelle fortgesetzt. Ein Klick auf eine Zelle des Kellers, der Halde oder auf ein Register öffnet ein Fenster. In diesem Fenster kann der Benutzer den Wert und den Typ, bei Registern nur den Wert ändern.

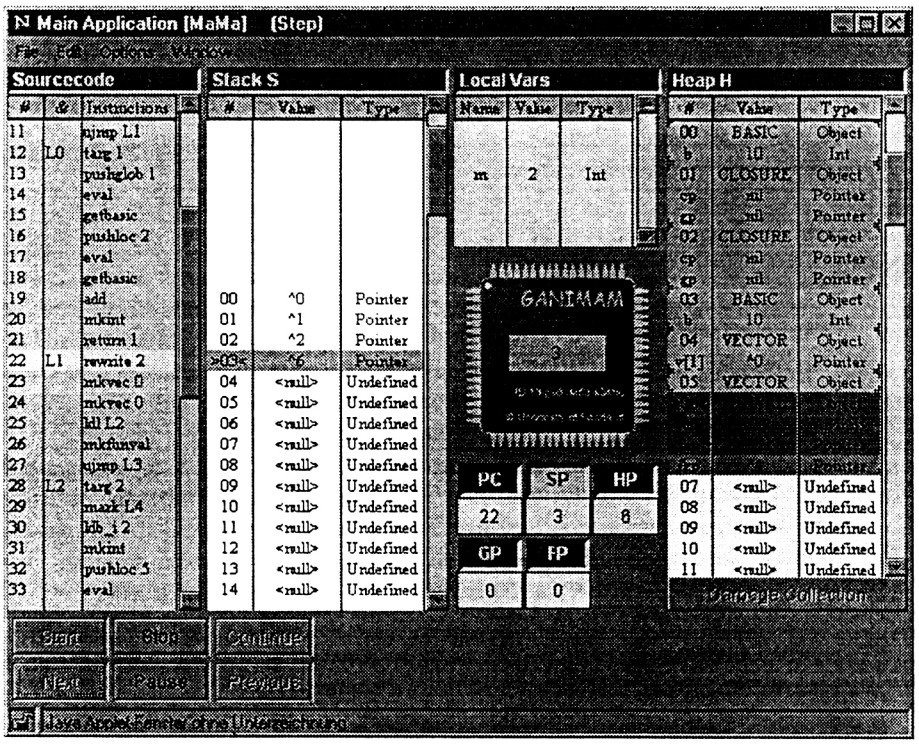


Abbildung3. Momentaufnahme einer animierten abstrakten Maschine

4.2 Vorteile interaktiver Animationen

GANIMAM bietet verschiedene Arten der Interaktion. Erstens kann der Benutzer Spezifikationen einer abstrakten Maschine eingeben oder ändern [4]. Beim Visualisieren einer abstrakten Maschine ist es wichtig, daß man nicht nur Daten und Vorgänge auf dem geringen Abstraktionsniveau und der Granularität der Spezifikationssprache darstellt, sondern auch Abstraktionen auf höherem Abstraktionsniveau. Um dies zu ermöglichen, wurden Visualisierungsannotationen zur Spezifikationssprache hinzugefügt. Diese Annotationen ähneln den 'interesting events' einschlägiger Algorithmenanimationssysteme, siehe Abschnitt 1.

Nach der Erzeugung der Implementierung der abstrakten Maschine kann der Benutzer Programme in der Sprache der abstrakten Maschine eingeben, diese Schritt für Schritt ausführen und den Inhalt jedes Registers und jeder Speicherzelle inspizieren. Während der Ausführung einer Instruktion zeigt eine Animation den Fluß der Information von Registern oder Speicherzellen in den Akkumulator und vom Akkumulator zurück zu Register oder Speicherzellen. Die Berechnung, die im Akkumulator durchgeführt wird, wird in einem separaten Fenster angezeigt.

Annotationen helfen nur solche Prinzipien sichtbar zu machen, die wir schon vorher kennen. GANIMAM kann auch dazu verwendet werden, neue Prinzipien durch Experimentieren mit Maschinenprogrammen oder Spezifikationen von abstrakten Maschinen zu entdecken. Dieses experimentelle Vorgehen kann u.a. für folgende Zwecke eingesetzt werden:

- Als Teil einer explorativen Lernsoftware ermöglicht es Studenten, neue Hypothesen zu formulieren und diese durch Änderung der Spezifikation oder des Maschinenprogramms zu überprüfen. Zusätzliche Hinweise in textueller Form sollte dabei sicherstellen, daß der Lerner auch die wesentlichen Punkte untersucht. Solche könnten z.B. der Vergleich zwischen caller-save-registers und callee-save-registers oder zwischen lazy und eager Auswertung sein oder das Finden des Kellerrahmens des statischen Vorgängers [18].
- Als Entwicklungswerkzeug kann es helfen, Fehler oder Optimierungsmöglichkeiten zu entdecken. Eine solche Optimierung könnte z.B. die Endrekursion betreffen: Durch das Verfolgen der Ausführung von Beispielprogrammen könnte der Benutzer feststellen, daß bestimmte Informationen in einem Kellerrahmen nach rekursiven Aufrufen nicht mehr benötigt werden.

GANIMAM kann auch von Forschern eingesetzt werden, um ihre neusten Implementierungstechniken vorzuführen oder einfach für Rapid Prototyping.

5 Das GANIMAL Framework

Basierend auf den mit GANIMAM und verwandten Arbeiten zur Softwarevisualisierung [16] gesammelten Erfahrungen haben wir das GANIMAL Framework für die Erstellung generativer Lernsoftware entwickelt, das in Abbildung 4 schematisch dargestellt ist. In diesem Framework wird aus einer in GANILA geschriebenen Spezifikation ein Algorithmenmodul automatisch erzeugt. Während der Ausführung des Algorithmus sendet dieses Modul ein sogenanntes "interesting event" `IE`, den Programmpunkt `pp` an dem das Event aufgetreten ist, sowie den aktuellen Animationsmodus (`RECORD`, `PLAY`, `REPLAY`) zum Kontrollobjekt, welches in seinen Einstellungen überprüft, ob für diesen speziellen Programmpunkt das interesting event zu allen Views weitergeleitet werden muß. In diesem Fall wird ein Broadcast des Events `IE` zu allen Views initiiert. Jede View hat ihre eigenen Einstellungen und überprüft nun wiederum selbst, ob sie den Eventhandler für `IE` aufrufen soll oder nicht. Abhängig vom aktuellen Modus produziert dieser Eventhandler graphische Ausgaben oder ändert lediglich den internen Zustand. Die graphische Benutzerschnittstelle GUI kann nun verwendet werden, um die Einstellungen für jeden Programmpunkt zu ändern, die Ausführung des Algorithmus zu kontrollieren und die Variablenwerte während der Ausführung zu ändern. Weiterhin können Programmpunkte, die über GANILA für parallele Ausführung markiert wurden, in der GUI so markiert werden, daß sie sequentiell ausgeführt werden.

Sämtliche Views sollten das objektorientierte Basispaket verwenden, das grafische Basisfunktionen enthält. Das umschließende Dokument ist ein Hypermediadokument, welches die verschiedenen Compilerphasen beschreibt. Es enthält Übungen, die teilweise mit Hilfe der erzeugten Generatoren lösbar sind.

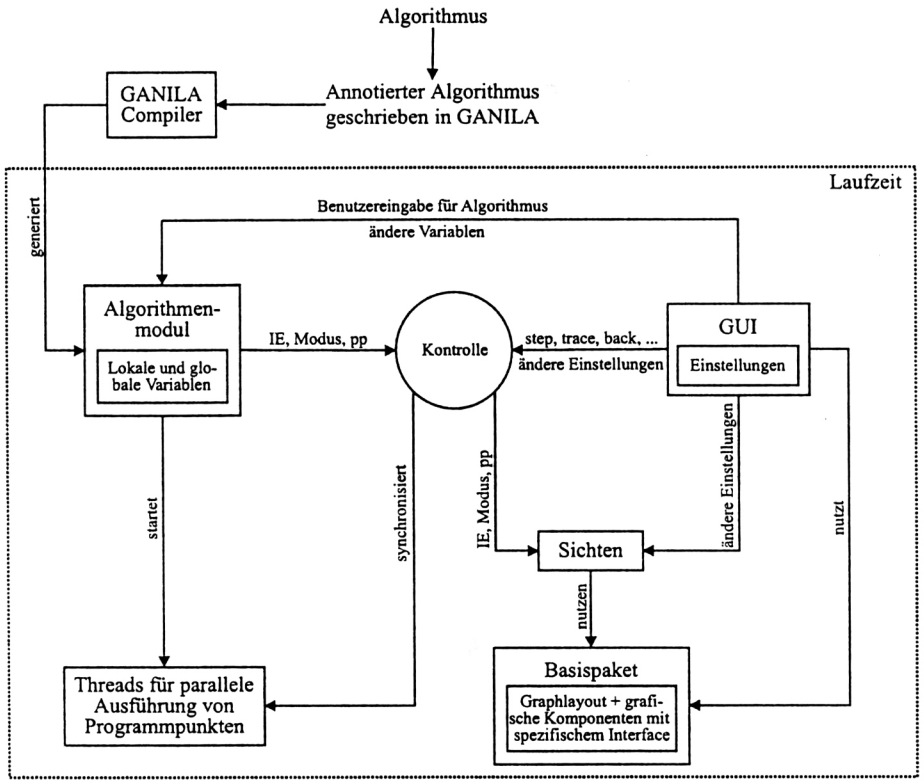


Abbildung4. Das GANIMAL Framework

Das Basispaket besteht aus einer Menge von Java Klassen, die primitive Methoden zur Kommunikation, zum Zeichnen und für Animationen beinhalten. Dies fördert zudem ein konsistentes Erscheinungsbild (look-and-feel) der verschiedenen Abschnitte der Lernsoftware. Somit erlaubt das GANIMAL Framework auch klassische Algorithmenanimationen in einfacher Weise zu implementieren, etwa die Animation des Heapsortalgorithmus, siehe Abbildung 5.

Die Plattformunabhängigkeit der Programmiersprache Java ermöglicht es, unabhängig vom lokal verfügbaren Computersystem, die Lernsoftware in allen Gebieten der Lehre zu verwenden. Die mächtige Java API (Klassenbibliothek) erleichtert die Benutzerinteraktion ebenso, wie die Entwicklung des graphischen Frontends.

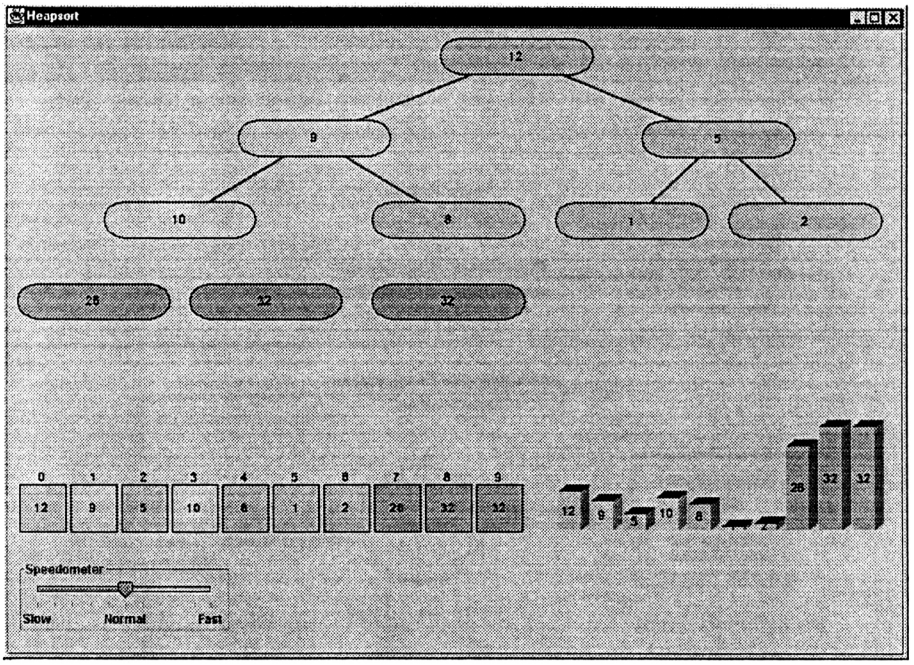


Abbildung5. Algorithmanimation von Heapsort

5.1 Vordefinierte Sichten

GANILA und das Basispaket stellen eine Menge von vordefinierten Sichten zur Verfügung, die durch ein GANILA-Sprachkonstrukt einfach in das Programm eingebunden werden. Dabei sind die Methoden der Sichten die Eventhandler der Events, die sie selbst verstehen. Andere Events werden einfach ignoriert.

HTMLView: In GANILA gibt es die Möglichkeit, Programmpunkte mit HTML-Dokumenten zu verbinden. Einerseits können HTML-Seiten über eine URL an diesem Programmpunkt durch einen kleinen HTML-Browser (IceBrowser-Java-Bean) angezeigt werden. Andererseits können Informationen, die zur Laufzeit an diesen Programmpunkten zur Verfügung stehen, durch Parameterübergabe an CGI-Skripten zu einem Server transferiert werden.

GraphView: Die GraphView bietet mehrere Graphlayoutalgorithmen, siehe auch Abschnitt 6.3. Knoten und Kanten können beliebig hinzugefügt bzw. gelöscht werden. Hierbei können die Knoten (fast) beliebige AWT-Komponenten sein.

CodeView: Die CodeView ermöglicht eine textuelle Sicht auf das auszuführende Programm und zeigt den aktuell ausgeführten Programmpunkt an.

Aura: Erhält über einen Event die URL einer Sounddatei und spielt diese ab. Start, Stop, Schleifen, Lautstärkeregelung, etc. werden ebenfalls über das Senden von Interesting Events gesteuert.

6 Design der visualisierenden Programmiersprache GANILA

Für die Spezifikation und Animation der Generatoren wurde die Sprache GANILA entworfen. GANILA erweitert Java um Interesting Events, parallele Ausführung von Programmpunkten, Annotationen für vorausschauendes Graphenlayout, Einbinden von Views und um Break- und Backtrackpunkte. GANILA wird durch einen Compiler (GAJA) nach Java übersetzt. Für jeden annotierten Programmpunkt kann diese Annotation zur Laufzeit der Animation über eine Benutzerschnittstelle ein- und ausgeschaltet werden. Die resultierenden Einstellungen können sowohl für den ganzen Algorithmus, als auch für jede View einzeln definiert werden, falls keine Parallelausführung eingestellt ist.

6.1 Interesting Events

Programm 1 zeigt ein GANILA Programm für eine einfache Operation: das Vertauschen der Inhalte zweier Feldelemente `a[i]` und `a[j]`. Interesting Events haben den Präfix `*IE_` und übertragen lokale Informationen über ihre Argumente zu den einzelnen Views, die diese Events bearbeiten. Ein solcher Event wird zuerst zu einer zentralen Kontrolle gesendet. Jede View registriert sich bei dieser Kontrolle und die Kontrolle leitet alle Interesting Events zu den registrierten Views weiter. Ein Eventhandler einer View kann wiederum neue Views kreieren und muß diese bei der Kontrolle registrieren.

Programm 1 GANILA-Code einer Vertauschungs-Methode

```
*{ *IE_MakeTemporary(1,i);
   help1 = a[i];
*} *||
*{ *IE_MakeTemporary(2,j);
   help2 = a[j];
*}

*{ *IE_MoveTemporary(j,1);
   a[j] = help1;
*} *||
*{ *IE_MoveTemporary(i,2);
   a[i] = help2;
*}
```

6.2 Parallele Ausführung

In GANILA ist es möglich, Programmpunkte durch `*{ ... }*` zu gruppieren und Programmpunkte bzw. Gruppen von Programmpunkten durch den Operator `*||` parallel auszuführen. In Programm 1 werden zuerst die Zuweisungen zu `help1` and `help2` parallel ausgeführt, dann die Zuweisungen zu `a[i]` und `a[j]` und ebenso ihre entsprechenden Interesting Events. Als Resultat werden ihre Animationen parallel dargestellt. Zu Beachten ist, daß wenn wir nur eine Hilfsvariable `help` zum Vertauschen verwenden würden, Datenabhängigkeiten eine parallele Ausführung nicht erlauben würden. D.h. der Algorithmus muß hier leicht umgeschrieben werden, damit die gewünschte Animation möglich ist.

6.3 Vorausschauendes Graphenlayout

Oft führen Animationen für Algorithmen, die Graphen verändern, d.h. Knoten oder Kanten hinzufügen oder löschen, zu Konfusionen, weil nach jeder Änderung ein neues Layout des aktuellen Graphen berechnet wird. In diesem neuen Layout werden Knoten zu unterschiedlichen Koordinaten bewegt, ohne daß der Algorithmus diese Knoten verändert hat. Daher ist es für den Anwender nicht sofort klar, welche Änderungen des Graphen durch den Algorithmus vorgenommen wurden. GANILA unterstützt Mechanismen für vorausschauendes Graphenlayout, d.h. ein Graph wird zur Zeit t_1 gezeichnet, wobei Informationen darüber verwendet werden, wie dieser Graph zu einem späteren Zeitpunkt t_2 aussehen wird.

Programm 2 Skizze des GANILA-Code für vorausschauendes Graphenlayout einer "RA→NEA" Algorithmenanimation

```
:
*RECORD;
  Code fuer die Umwandlung eines regulaeren
  Ausdrucks in einen nichtdeterministischen Automaten
*REPLAY;
*PLAY;
  Rest des Programms
:
```

Abbildung 6 zeigt, wie dieser Mechanismus verwendet werden kann, um die Umwandlung eines regulären Ausdrucks zu einem nichtdeterministischen Automaten (RA→NEA) zu animieren. In Programm 2 ist eine Skizze des entsprechenden GANILA-Codes angegeben. Die Instruktionen `*RECORD` und `*PLAY` werden verwendet, um einen bestimmten Modus zur Ausführung der graphischen Primitive auszuwählen. Im PLAY Modus werden alle interesting events unmittelbar ausgeführt. Im RECORD Modus werden Änderungen im internen Zustand vorgenommen, allerdings ohne graphische Ausgabe. Alle Interesting Events werden

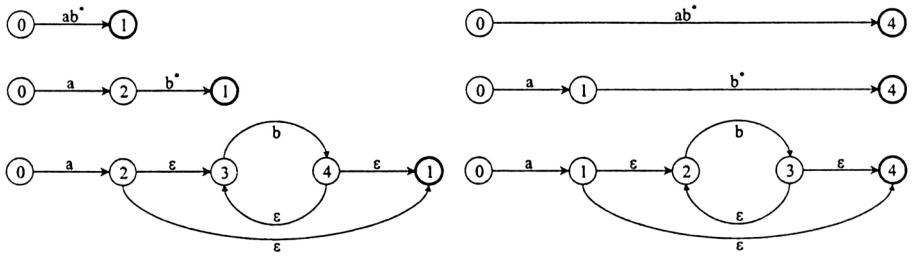


Abbildung6. Ad-Hoc und vorausschauendes Graphenlayout für endliche Automaten

dabei gespeichert. Die Instruktion `*REPLAY` ruft alle gespeicherten Events mit dem endgültigen Zustand als zusätzliches Argument wieder auf. Nun werden alle Zustandsänderungen ausgeführt und graphischer Output erzeugt.

6.4 Break- und Backtrackpunkte

Programmpunkte können mit `*BREAK` als Breakpunkte markiert werden. Wenn die Ausführung des Algorithmus solch einen Breakpunkt erreicht, wird die Ausführung unterbrochen und der Benutzer kann z.B. den gegenwärtigen Zustand untersuchen, fortfahren oder den Algorithmus schrittweise ausführen.

Backtrackpunkte werden mit `*SAVE` markiert. Wenn die Ausführung eines Algorithmus einen solchen Punkt erreicht, kopiert das System den aktuellen Zustand und speichert ihn in einer History.

Backtrackpunkte sind ein Mittel für Rückwärtsausführung eines Algorithmus oder Wiederholung eines Algorithmus von einem vorhergehenden Punkt aus, unter Umständen mit geänderten Einstellungen. Ein anderes Mittel ist die Wiederholung aller vom Startpunkt des Algorithmus aufgezeichneten Interesting Events. Dies ist zwar eine zeitintensivere, aber weniger speicherverzehrende Alternative, die vom System angeboten wird.

7 Abschlußbemerkungen

Wir haben die meisten Teile des Basispakets implementiert und eine formale Beschreibung des GANILA Compilers GAJA erstellt. Die GANILA Quellen des Heapsort Beispiels wurden durch den GAJA-Compiler in ein Java Programm konvertiert. Eine prototypische Implementierung des Compilers, sowie Beispielvisualisierungen, die durch den Compiler erzeugt werden (d.h. Heapsort und Berechnungsmodell der endlichen Automaten) werden zum Zeitpunkt des Workshops verfügbar sein. Mehr Informationen über das GANIMAL Projekt und weitere Beispiele sind unter [8] zu finden.

Das GANIMAL Framework und insbesondere GANILA bieten eine Menge von mächtigen Features. Wir erwarten, daß die beachtliche Investitionen an Zeit

und Anstrengungen bei deren Design und Implementierung sich auszahlen werden, wenn wir beginnen, Inhalte für die Lernsoftware als solche zu kreieren. Letztes Jahr haben wir in Experimenten mit Studenten eine Lernsoftware über endliche Automaten ohne generativen Teil evaluiert. Wir beabsichtigen, die gesammelten Ergebnisse mit denen der Evaluation unserer neuen Lernsoftware zu vergleichen, die wir mit Hilfe des GANIMAL Frameworks erstellt haben.

Literatur

1. B. Braune, S. Diehl, A. Kerren, R. Wilhelm. *Animation of the Generation and Computation of Finite Automata for Learning Software.* To appear in Proceedings of Workshop of Implementing Automata WIA'99, Potsdam, Germany, July, 1999.
2. M. H. Brown. *Algorithm Animation.* MIT Press, 1987.
3. M. H. Brown. *Zeus: A System for Algorithm Animation and Multiview Editing.* In IEEE Workshop on Visual Languages, pp. 4-9, 1991. Also appeared as SRC Research Report 75.
4. S. Diehl, T. Kunze, A. Placzek. *GANIMAM: Generation of Interactive Animations of Abstract Maschines.* In Proceedings of "Smalltalk und Java in Industrie und Ausbildung STJA'97" (in German), pp. 185-190, Erfurt (Germany), 1997.
5. S. Diehl, T. Kunze. *Visualizing Principles of Abstract Maschines by Generating Interactive Animations.* In Proceedings of Workshop on Principles of Abstract Machines, Pisa, Italy, 1998.
6. S. Diehl, T. Kunze. *Visualizing Principles of Abstract Machines by Generating Interactive Animations.* To appear in Future Generation Computer Systems, Elsevier, 1999.
7. S. Diehl, A. Kerren. *Increasing Explorativity by Generation.* To appear in Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, EDMEDIA-2000, AACE, Montreal, Canada, 2000.
8. http://www.cs.uni-sb.de/GANIMAL
9. http://www.cs.uni-sb.de/RW/users/ganimal/GANIFA
10. A. Kerren. *Animation of the Semantical Analysis.* Master Thesis (in German), University of Saarland, Saarbrücken (Germany), 1997.
11. A. Kerren. *Animation of the Semantical Analysis.* In Proceedings of „8. GI-Fachtagung Informatik und Schule INFOS99" (in German), pp. 108-120, Informatik aktuell, Springer, 1999.
12. http://www.mpi-sb.mpg.de/~marco/java.html.
13. R. Schulmeister. *Foundations of Hypermedial Learning Systems.* Addison Wesley (in German), Bonn (Germany), 1996.
14. J. T. Stasko. *A Framework and System for Algorithm Animation.* Computer, 18(2), pp. 258-264, 1990.
15. J. T. Stasko. *The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces.* Journal of Visual Languages and Computing (1), pp. 213-236, 1990.
16. J.T Stasko, J. Domingue, M.H. Brown, B.A. Price. *Software Visualization.* The MIT Press, 1997.
17. A. Y. Tal, D. P. Dobkin. *Visualization of Geometric Algorithms.* IEEE Transactions on Visualization and Computer Graphics, Vol. 1, Num. 2, 1995.
18. R. Wilhelm, D. Maurer. *Compiler Design: Theory, Construction, Generation.* Addison-Wesley, 1995.

*Vi*sualisierung *v*on *a*usgewählten *L*ehrinhalten *d*er *I*nformatik - Beispielanimationen aus dem Projekt *Vivaldi*

Rainer Oechsle, Dieter Jahn

Fachhochschule Trier, Fachbereich Angewandte Informatik

Kurzfassung

Im Rahmen des Projekts ***Vivaldi*** wurden ausgewählte Lehrinhalte der Informatik visualisiert. Die Visualisierungen werden unterstützend zur traditionellen Vorlesung verwendet. Außerdem wird den Studierenden eine Plattform zur Verfügung gestellt, mit der sie eigene Programme mit kontinuierlichen Bewegungen visualisieren können. Grundlage der Visualisierungen ist der ***Vivaldi***-Kern, der eine Reihe von Klassen umfasst, mit deren Hilfe grafische Objekte auf dem Bildschirm dargestellt werden können. Ferner ist es möglich, für die grafischen Objekte Vorschriften anzugeben, wie sich ihr Aussehen oder ihre Position im Laufe der Zeit verändert. Damit können dann kontinuierliche Bewegungen von Objekten spezifiziert werden.

1. Einleitung

Im Rahmen des verhältnismäßig kleinen Projekts ***Vivaldi*** (**V**isualisierung von ausgewählten **L**ehrinhalten **d**er **I**nformatik) im Fachbereich Angewandte Informatik der FH Trier war es nicht möglich, eine Lernsoftware zu entwickeln, die einen gewissen Lehrbereich komplett abdeckt. Die Ziele dieses Projekts, das vom 1. April 1998 bis zum 31. Dezember 1999 an der FH Trier mit zwei halben Stellen durchgeführt wurde, waren statt dessen wesentlich bescheidener:

- Es sollten Lehrinhalte der Informatik punktuell ausgewählt werden, die sich besonders gut für eine Visualisierung eignen.
- Die Visualisierungen sollten real ablaufende Algorithmen illustrieren, statt einen solchen Ablauf nur vorzutäuschen. Das heißt zum Beispiel, dass ein Sortieralgorithmus tatsächlich abläuft und dabei über eine Visualisierung anzeigt, was gerade getan wird. Damit kann durch Veränderung von Parametern die Anwendung und damit auch die Visualisierung unterschiedlich ablaufen und interaktiv gesteuert werden.
- Die entwickelten Visualisierungen haben nicht den Anspruch, dass sie einen zuvor gänzlich unbekannten Sachverhalt erklären können, sondern die Visualisierungen sollen unterstützend zur traditionellen Vorlesung verwendet werden. Den Studierenden sollen die Visualisierungen im Anschluss an die Vorlesung zugänglich sein, so dass sie diese mit eventuell anderen Parametern nochmals selber durchspielen können, um ihr Verständnis dadurch zu vertiefen.

- Im Projekt sollte eine Beobachtung berücksichtigt werden, die wir im Vorfeld des Projekts gemacht haben: Häufig haben die Entwickler einer Visualisierung mehr Nutzen davon als die späteren Betrachter. Aus diesem Grund sollte das Projekt ***Vivaldi*** eine Plattform zur Verfügung stellen, mit der die Studierenden eigene Programme visualisieren können, wobei in diesem Fall die Visualisierung nicht primär für andere gedacht ist. Stehen den Studierenden z.B. entsprechende Visualisierungsoperationen zum Vergleich zweier Elemente und zum Vertauschen zweier Elemente zur Verfügung, so können sie selber unterschiedliche Sortieralgorithmen programmieren und an entsprechender Stelle die zur Verfügung gestellten Visualisierungsoperationen in ihr Programm einbauen (d.h. sie können ihr Sortierprogramm zum Zwecke der Visualisierung instrumentieren). In diesem Fall kann die Visualisierung zum schnellen Erkennen von Programmierfehlern dienen. Es sollte nicht unterschätzt werden, dass eine derartige Nutzung von Visualisierungen den Spaß am Programmieren erhöht und dadurch die Studierenden (und auch die Autoren dieses Beitrags) in besonderer Weise motiviert.
- In den Visualiserungen sollten „sanfte Übergänge" (kontinuierliche Bewegungen) statt lediglich eine Folge von Schnappschüssen vorkommen. Dadurch werden Veränderungen auf dem Bildschirm besser erkennbar. Die Autoren haben selber die Beobachtung gemacht, dass viele Visualisierungen aufgrund abrupter Übergänge schwer zu verfolgen sind.

Das Projekt ***Vivaldi*** lief in zwei Phasen ab: In der ersten Phase wurde eine Klassenbibliothek, der ***Vivaldi***-Kern, entwickelt. Der ***Vivaldi***-Kern besteht aus einer Reihe von Klassen, mit deren Hilfe grafische Objekte wie Rechtecke und Kreise auf dem Bildschirm dargestellt werden können. Ferner ist es möglich, für diese grafischen Objekte Vorschriften anzugeben, wie sich ihr Aussehen oder ihre Position im Laufe der Zeit verändert. Damit können dann u.a. kontinuierliche Bewegungen von Objekten spezifiziert werden. In der zweiten Phase wurden Visualisierungen aus unterschiedlichen Bereichen der Informatik unter Nutzung des ***Vivaldi***-Kerns entwickelt. Zum größten Teil handelt es sich dabei um Algorithmenanimationen; nur eine Visualisierung ist dem Gebiet der Programmanimationen zuzurechnen (zur Klassifikation der Software-Visualisierung siehe z.B. [Price, Baecker, Small 93]).
Dieser Beitrag ist wie folgt gegliedert: Im nächsten Abschnitt gehen wir kurz auf den ***Vivaldi***-Kern ein, der bereits an anderer Stelle ausführlicher beschrieben wurde [Oechsle 99; Oechsle, Becker 99]. Im dritten Abschnitt werden einige ausgewählte Animationen vorgestellt, die mit Hilfe des ***Vivaldi***-Kerns entwickelt wurden. Der vierte Abschnitt enthält Hinweise auf verwandte Arbeiten. Abschließend schildern wir im fünften Abschnitt die Erkenntnisse, die wir gewonnen und deren Konsequenzen wir bei der Entwicklung der Animationen berücksichtigt haben, und geben einen Ausblick auf zukünftige Aktivitäten.

2. Der *Vivaldi*-Kern

Die im folgenden Abschnitt beschriebenen Visualisierungen besitzen in der Regel folgende Struktur: Ein zu visualisierender Algorithmus (die eigentliche Anwendung

wie z.B. ein Sortierverfahren) wird mit Methodenaufrufen zur visuellen Darstellung instrumentiert (z.B. Methoden, die das Vergleichen oder Vertauschen zweier Elemente grafisch anzeigen). Die Klassen, welche diese visuelle Darstellung realisieren, bezeichnen wir als "Script". Der Name "Script" soll dabei an ein Drehbuch für einen Film erinnern; ein Visualisierungsscript hat nichts mit einem Script im Sinne eines interpretierten Programms wie z.B. einem Shell-Script in UNIX zu tun. Bei der Entwicklung der anwendungsspezifischen visuellen Darstellung als Script wird auf die Klassenbibliothek ***Vivaldi***-Kern zurückgegriffen, die es erlaubt, grafische Objekte darzustellen, die sich nach anzugebenden Vorschriften bewegen und ihr Aussehen verändern. Abbildung 1 fasst den soeben geschilderten Sachverhalt zusammen.

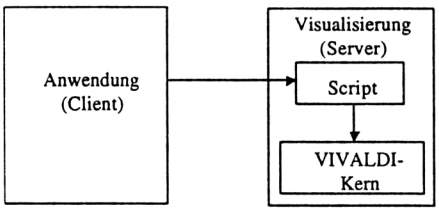


Abbildung 1: Trennung von Anwendung und visueller Darstellung

Der ***Vivaldi***-Kern basiert auf dem AWT (Abstract Window Toolkit) von Java JDK 1.1. Er besteht aus Klassen zur Repräsentation grafischer Objekte auf dem Bildschirm und zur Festlegung, wie sich das Aussehen und die Position grafischer Objekte im Laufe der Zeit verändern sollen.

Grafische Objekte: Alle Klassen zur Repräsentation grafischer Objekte werden von der Basisklasse `GraphicElement` abgeleitet. Die Klassen bilden eine echt objektorientierte Verkleidung der in Java grafisch darstellbaren Objekte. So ist z.B. die Farbe ein Attribut der grafischen Objekte, während in Java die Farbe als eine Art von globaler Variable im Grafikkontext vor dem Zeichnen eines grafischen Objekts gesetzt werden muss. Ferner gibt es einige Klassen, deren Funktionalität nicht direkt in Java verfügbar ist. Die Klasse `ArrowElement` z.B. erweitert die Klasse `LineElement` um das Konzept von Pfeilen; ein Pfeil ist eine Linie mit einer Pfeilspitze. Die Klasse `PhaseImageElement` ist eine Folge von Pixelbildern im GIF- oder JPEG-Format. Diese werden im Konstruktor durch ein Feld von Dateinamen spezifiziert. Die Bilder sind entsprechend dieses Dateinamenfelds von 0 an durchnummeriert. Das `int`-Attribut `phase` beschreibt, welches der Bilder aktuell angezeigt wird. Durch eine rasche Veränderung dieses Attributwerts entsteht bei entsprechend geeigneten Bildern der Eindruck eines Films. Mit Hilfe der Klasse `GraphicElementGroup` können mehrere `GraphicElement`-Objekte in einem Objekt zu einer Gruppe zusammengefasst werden. Da `GraphicElementGroup` selbst aus `GraphicElement` abgeleitet ist, kann eine Gruppe u.a. wieder eine Gruppe enthalten. Damit kann ein Baum von `GraphicElement`-Objekten generiert werden. Der Entwurf der graphischen Elemente ist gemäß dem Entwurfsmuster "Composite" [Gamma et al. 95] gestaltet. Die Auswahl der hier beschriebenen Klassen von grafischen Elementen sind in Abbildung 2 mit ihren Vererbungsbeziehungen und einigen ausgewählten Attributen zusammenfassend dargestellt.

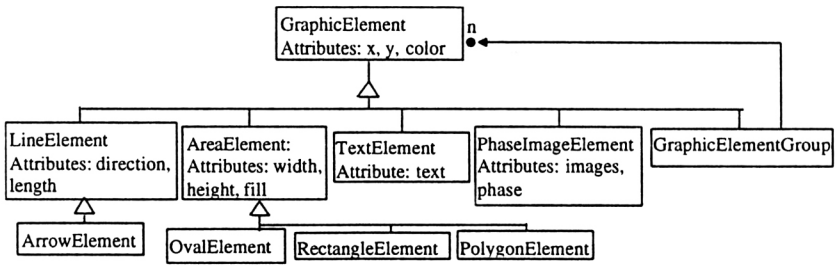


Abbildung 2: Teil des Klassendiagramms der grafischen Elemente

Jedes grafische Objekt kann auf Mausereignisse (Klicken, Bewegen usw.) reagieren, indem - wie in Java entsprechend dem Observer-Observable-Muster [Gamma et al. 95] allgemein üblich - ein sogenannter Listener an das Objekt gebunden wird. Dieser Listener muss eine Reihe von Methoden implementieren, die bei Eintreffen der entsprechenden Mausereignisse aufgerufen werden. Die Pfeile in Abbildung 1 können daher durch Doppelpfeile ersetzt werden, da Rückmeldungen vom ***Vivaldi***-Kern über das Script an die Anwendung möglich sind.

Attributtransitionen: Die Grundidee für die Spezifikation von kontinuierlichen Veränderungen der grafischen Objekte des ***Vivaldi***-Kerns ist, dass Vorschriften für die zeitliche Veränderung der Attribute von grafischen Objekten angegeben werden. Diese Vorschriften werden Attributtransitionen genannt. Eine Attributtransition bezieht sich dabei immer auf *genau ein* Attribut von *genau einem* grafischen Objekt. Will man also z.B. eine Bewegung eines einzigen Objekts beschreiben, so benötigt man zwei Attributtransitionen, eine für die x- und eine für die y-Koordinate dieses Objekts. Eine Transition besitzt im wesentlichen zwei Referenzen: eine auf das Objekt, auf das sie sich bezieht, und eine auf eine Funktion, welche die Zeit auf einen Attributwert abbildet. Auf welches Attribut des grafischen Objekts sich eine Transition bezieht, wird durch die Klasse des Transitionsobjekts festgelegt. So gibt es für jedes Attribut der grafischen Objekte eine entsprechende Transitionsklasse.
Abbildung 3 stellt die wichtigsten Objekte und ihre Beziehungen in einem Beispiel dar. Im oberen Teil der Abbildung sieht man die Enthaltenseinshierarchie der grafischen Objekte. Der untere Teil zeigt die Menge von Transitionen, wobei jede Transition eine Referenz auf ein grafisches Objekt und eine Zeitfunktion besitzt. Zum leichteren Umgang des Programmierers mit Transitionen gibt es wie bei den Grafikobjekten auch Transitionsgruppen. Damit können mehrere Transitionen auf einfache Art zusammengefasst werden. Durch das Konzept der Transitionsgruppe erhält man für die Transitionen ebenfalls einen Baum von Transitionen wie für die Grafikobjekte.

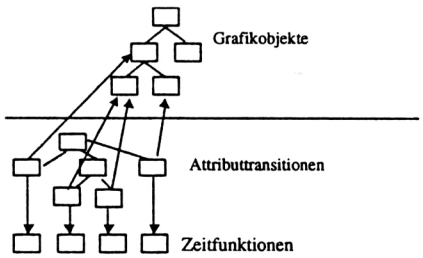


Abbildung 3: Beispiel für Beziehungen zwischen Baum von Grafikobjekten, Baum von Attributtransitionen und Zeitfunktionen

Aus der Abbildung können folgende Sachverhalte gefolgert werden:

- Die Transitionen können sich auf primitive Grafikobjekte oder auf Gruppen von Objekten beziehen. Damit kann z.B. eine Gruppe von Objekten bewegt werden.
- Es kann mehrere Transitionen geben, die sich auf dasselbe grafische Objekt beziehen. Dies macht allerdings nur dann Sinn, falls sich die Transition auf unterschiedliche Attribute dieses Objekts bezieht (z.B. x- und y-Koordinate).
- Nur die primitiven Transitionen (also die Nicht-Gruppentransitionen) besitzen Referenzen auf grafische Objekte und Zeitfunktionen.

Weitere Eigenschaften der Attributtransitionen sind:

- Die Transitionen können in die Liste aller Transitionen dynamisch eingefügt und daraus gelöscht werden. Damit können z.B. Bewegungen gestartet und beendet werden.
- Die Transitionen können endlich oder unendlich sein.
- Eine endliche Transition entfernt sich aus ihrer Gruppe selbst, sobald sie beendet ist. Dieses Prinzip wird rekursiv angewendet: Eine Transitionsgruppe ist dann beendet, falls alle ihre Mitglieder beendet sind. Dies ist genau dann der Fall, falls die Gruppe keine Mitglieder mehr hat.
- Jede Transition hat eine `synch`-Methode, mit der man sich auf das Ende einer Transition synchronisieren kann. Das heißt, ein Thread, welcher die `synch`-Methode auf eine Transition anwendet, wird solange blockiert, bis die entsprechende Transition beendet ist. Die `synch`-Methode kann auch auf eine Transitionsgruppe angewendet werden.

3. Visualisierungsbeispiele

Im Folgenden werden einige Beispiele für Animationen aus unterschiedlichen Bereichen der Informatik vorgestellt, die im Rahmen des *Vivaldi*-Projekts entstanden sind.

Bereich Software-Entwicklung

Mit Hilfe eines Editors können Methodenaufrufe innerhalb eines beliebigen Java-Programms markiert und mit einem Kommentar versehen werden. Wird aus diesem

Editor heraus das Programm übersetzt, so entsteht beim Ablauf dieses Programms eine Animation, welche die Methodenaufrufe im Stil eines Sequenzdiagramms nach der Unified Modeling Language (UML) dynamisch zusammen mit den Kommentaren darstellt. Das heißt, dass für ein beliebiges Java-Programm automatisch eine Animation generiert werden kann, welche die Interaktionen der Objekte des gerade ablaufenden Programms darstellt. Das System kann sogar rekursive Aufrufe korrekt und übersichtlich darstellen – eine Fähigkeit, die vielen kommerziell verfügbaren CASE-Tools immer noch fehlt. Das entwickelte System eignet sich sowohl zur Unterstützung der Lehre als auch als Hilfe bei der Software-Entwicklung.
Eine automatische Visualisierung des gesamten Programms ist in der Regel nicht ratsam, da man dann beim Ablauf des Programms "den Wald vor lauter Bäumen nicht mehr sieht". Deshalb haben wir uns dafür entschieden, dass der Entwickler spezifizieren muss, was er beim Ablauf sehen möchte.

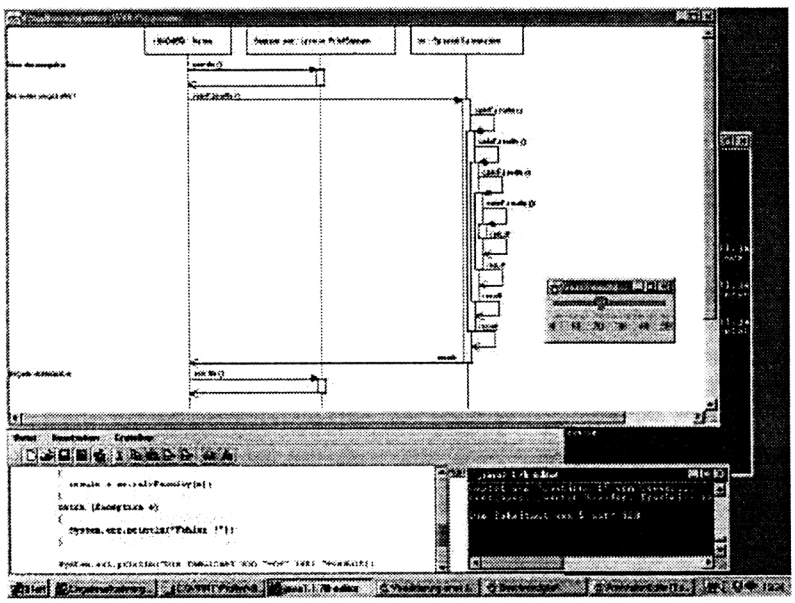


Abbildung 4: Dynamische Sequenzdiagramm-Visualisierung

In Abbildung 4 ist ein Schnappschuss der ablaufenden Visualiserung zu sehen. Das untere linke Fenster enthält den Editor, in den das zu visualisierende Programm geladen wurde. Die Methodenaufrufe, die visualisiert werden sollen, sind entsprechend farbig markiert worden. Das kleine Fenster rechts unten enthält die Ausgabe des rein zeilenorientierten Programms, nämlich das Ergebnis der Fakultätsberechnung. Das obere große Fenster beinhaltet die Visualisierung des gerade ablaufenden Programms. Die dargestellten Pfeile und Balken werden nicht auf einen Schlag, sondern mit kontinuierlichen Bewegungen allmählich auf den Bildschirm gebracht. Durch die Visualisierung verzögert sich der Ablauf des Programms entsprechend, da diese Visualisierung synchron zum ablaufenden Programm durchgeführt wird. Als Beispiel wurde die Berechnung der Fakultätsfunktion in rekursiver Weise gewählt,

um zu zeigen, wie rekursive Aufrufe visualisiert werden. Diese Visualisierung ist die einzige Programmanimation, die im Rahmen des *Vivaldi*-Projekts entwickelt wurde; alle anderen Beispiele stellen Algorithmenanimationen dar.

Bereich Grundlagen der Informatik

Sortieralgorithmen: Sortieralgorithmen wie Selection-, Insertion-, Bubble-, Quick- und Shell-Sort sind klassische Beispiele für Algorithmenanimationen. Die Animationen dienten als erster Test des *Vivaldi*-Kerns und wurden als Java-Applets implementiert. Jedes Applet stellt dem Benutzer eine Auswahlmöglichkeit bezüglich der Menge der zu sortierenden Elemente zur Verfügung. Der Benutzer kann jeweils zwischen einer Menge mit zufällig erzeugten Elementen oder zwei unterschiedlichen, fest vorgegebenen Beispielmengen wählen. Ebenso kann die Visualisierungsgeschwindigkeit bei jedem Applet vom Benutzer selbst eingestellt werden.
Damit die erstellten Visualisierungen der Sortieralgorithmen im Grundstudium der Angewandte Informatik in den Übungsstunden eingesetzt werden können, wurde zu jedem Sortieralgorithmus eine ausgesuchte Menge von Visualisierungsmethoden bereitgestellt. Die Studierenden erhalten zu jeder Methode eine Erklärung über deren Funktion. Dann sollen sie die Sortieralgorithmen programmieren und den Aufruf der Visualisierungsmethoden an der richtigen Stelle im Sortieralgorithmus einfügen. Die Visualisierungen auf den Internetseiten sollen hierbei dazu beitragen, dass die Funktionsweise der einzelnen Algorithmen von den Studierenden besser verstanden werden können.

Synchronisations- und Kommunikationsverfahren: Ebenfalls als Java-Applets wurden Visualisierungen verschiedener Synchronisations- und Kommunikationsverfahren mit Hilfe von Semaphorgruppen, Dateisperren, Pipes und Message Queues realisiert. Die Semantik dieser grundlegenden Konzepte entspricht derjenigen des Betriebssystems UNIX.
Die Realisierung dieser Visualisierungen erfolgte, indem die aus UNIX bekannten Konzepte in Java nachimplementiert wurden. Dabei wurden die Aufrufe für die Visualisierung lediglich auf die Methoden der Synchronisations- und Kommunikationsobjekte beschränkt. Dennoch werden die beteiligten aktiven Komponenten, die Threads, ebenfalls dargestellt. Dies geschieht dadurch, dass bei einem Aufruf einer Methode zuerst geprüft wird, ob der aufrufende Thread schon dargestellt ist oder nicht. Ist er noch nicht zu sehen, so erfolgt seine Darstellung zu diesem Zeitpunkt.
Der Vorteil dieses Konzepts ist, dass Studierende unter Verwendung der Synchronisations- und Kommunikationsobjekte Anwendungen programmieren können und beim Ablauf dieser Anwendungen eine Animation ihres Programms sehen, ohne dass sie sich selbst um die Animation kümmern müssen.
Die auf den Internet-Seiten zu sehenden Animationen sind zum einen solche, die vom Benutzer manuell gesteuert werden müssen: Es werden mehrere Threads gestartet, wobei für jeden Thread ein Dialogfenster auf dem Bildschirm erscheint, so dass der Benutzer spezifizieren kann, was dieser Thread als nächstes tun soll (z.B. welche Methode [Senden oder Empfangen] soll ausgeführt werden). Zum anderen wurden bekannte Synchronisationsprobleme wie das Leser-Schreiber-Problem oder das

Problem der fünf essenden Philosophen (Dinning Philosophers) mit Hilfe der Semaphorgruppenobjekte visualisiert.
Die Visualisierungen der Sortieralgorithmen und der Synchronisations- und Kommunikationsverfahren sind öffentlich zugänglich. Als Einstiegspunkt empfehlen wir die Adresse http://www.informatik.fh-trier.de/projekte/vivaldi/vivaldi.html, die *Vivaldi*-Projektseite.

Bereich Rechnernetze

Visualisierung von Routing-Algorithmen: In einer weiteren Arbeit wurde das im Internet wichtige RIP-Verfahren (Routing Information Protocol) visualisiert. RIP ist ein sogenanntes Routing-Verfahren, mit dem die Router im Internet untereinander Informationen austauschen, um daraus ihre Wegewahltabellen zu berechnen. Damit werden die Wegewahltabellen ständig auf dem aktuellen Stand gehalten.
Um den RIP-Algorithmus visualisieren zu können, ist es zunächst nötig, ein Netzwerk zu erstellen. Die Anwendung stellt daher im Hauptfenster eine Visualisierungsfläche zur Verfügung. In dieser Fläche können Router frei platziert und anschließend durch Ziehen einer Linie verbunden werden. Auf diese Weise kann der Benutzer ein Netzwerk erstellen, welches genau auf seine Bedürfnisse zugeschnitten ist. Beispielsweise kann er ein Netz aus nur zwei oder drei Routern entwerfen, um die Komplexität gering zu halten und so ein grundsätzliches Verständnis der Abläufe zu erlangen. Andererseits kann aber auch ein Netz entworfen werden, um eine bestimmte Situation, wie z.B. das „Zählen bis unendlich", zu demonstrieren. Ist ein solches Netzwerk einmal erstellt, kann es in einer Datei abgespeichert werden.
Um die Simulation des RIP-Algorithmus zu starten, reicht es, einen Router doppelt anzuklicken. Daraufhin sendet dieser seinen Distanzvektor an alle seine Nachbarn, wobei jeder Distanzvektor als Rechteck auf einer Leitung dargestellt wird. Die Änderungen in der Wegewahltabelle eines Routers können nun in einem Dialogfenster, welches für jeden Router geöffnet werden kann, beobachtet werden. In diesem Fenster ist die neu berechnete Wegewahltabelle, der empfangene Distanzvektor, der die Neuberechnung verursacht hat, mit Absender und die Ausgangswegewahltabelle zu sehen. Durch diese Informationen ist es möglich, die Berechnung der Wegewahltabelle aus dem empfangenen Distanzvektor und der bisherigen Tabelle nachzuvollziehen. Durch wiederholtes Doppelklicken auf weitere Router kann nun Schritt für Schritt die Berechnung der Wegewahltabellen im ganzen Netz erfolgen. Diese sogenannte „manuelle" Vorgehensweise ist besonders dann günstig, wenn bestimmte „was-wäre-wenn"-Situationen erzeugt und beobachtet werden sollen. Es ist aber auch möglich, die Anwendung selber entscheiden zu lassen, welcher Router an der Reihe ist, seinen Distanzvektor zu versenden. In jedem Fall muss aber die Animation manuell gesteuert werden, denn ein automatischer Ablauf ist auch bei geringer Geschwindigkeit wegen der Komplexität der Anwendung vom Betrachter nicht nachvollziehbar. Während der Simulation können jederzeit Verbindungen als defekt gekennzeichnet oder Router gelöscht werden.
In ähnlicher Weise wurde das OSPF-Verfahren (OSPF: Open Shortest Path First) visualisiert.

Visualisierung des TCP-Protokolls: Die TCP-Visualisierung gliedert sich in zwei Bereiche, bei denen es sich zum einen um das TCP-Verbindungsmanagement und zum anderen um den Datentransfer handelt. Beim Verbindungsmanagement werden die Zustandsänderungen beider Partner mit endlichen Automaten dargestellt. Hier hat der Benutzer die Möglichkeit, den Verbindungsaufbau und -abbau selbst zu beeinflussen, z.B. gleichzeitiger Verbindungsaufbau von beiden Partnern. Beim Datentransfer wird weniger auf den zeitlichen Ablauf des datenstromorientrierten TCPs Wert gelegt, sondern auf die Sicherstellung der Zuverlässigkeit und der Flusskontrolle. Durch die manuellen Eingaben können die verschiedensten Szenarien durchgespielt werden, z.B. Paketverzögerung und -verlust im Netz, Timeoutbehandlung und Überflutung des Empfängers. Dabei wird der Benutzer nach jeder Eingabe über die Änderung der Sequenznummer, Quittungsnummer und Fenstergröße informiert. In Abbildung 5 ist die TCP-Visualisierung dargestellt.

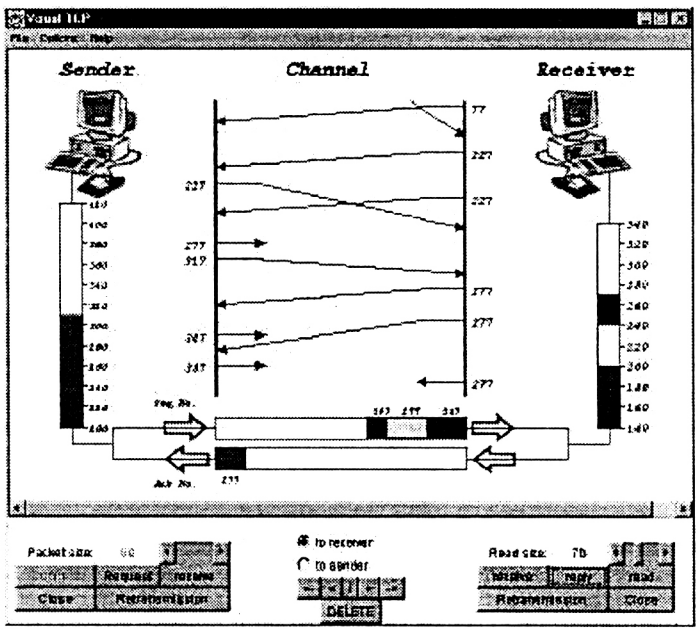


Abbildung 5: Programmfenster der TCP-Visualisierung

4. Verwandte Arbeiten

In vielen Veröffentlichungen werden immer wieder das Balsa-System [Brown 84] und dessen Nachfolger Balsa-II [Brown 88a; Brown 88b] und Zeus [Brown 91] als die ersten bedeutenden Arbeiten auf dem Gebiet der Algorithmenanimation angeführt. Diese und viele andere Artikel konzentrieren sich sehr stark auf die jeweiligen Anwendungen, so dass man nicht erkennen kann, inwieweit diesen Anwendungen ein Basis system zugrunde liegt, das mit dem ***Vivaldi***-Kern verglichen werden kann.

Der einzige uns bekannte Artikel, der ein Basissystem zur Programmierung von Animationen etwas detaillierter darstellt, ist ein von J. Stasko verfasstes Papier über das von ihm entwickelte Tango-System [Stasko 90], das auch Thema seiner Dissertation war [Stasko 89]. Der Artikel [Stasko 90] war eine wesentliche Inspiration für das ***Vivaldi***-System. Stasko beschreibt sehr deutlich die Trennung von Anwendung (Algorithmuskomponente) und Visualisierung (Animationskomponente). Die Animationskomponente beinhaltet das Pfad-Transitions-Paradigma (dies entspricht in ***Vivaldi*** dem Kern), das wie der ***Vivaldi***-Kern u.a. "sanfte Übergänge" (kontinuierliche Bewegungen) unterstützt. Der Ansatz unterscheidet sich dennoch stark vom ***Vivaldi***-Kern. Eine Schwäche des Pfad-Transitions-Paradigmas liegt unserer Meinung nach in der fehlenden Objektorientierung, was u.a. bewirkt, dass das System schwerer verständlich und programmierbar ist. Auch sind einige Funktionalitäten des ***Vivaldi***-Kerns wie unendliche Transitionen und Synchronisationsoperationen nicht vorhanden. Dennoch ist das Tango-System ein beeindruckendes System, anhand dessen wesentliche Basiskonzepte von Animationssystemen umfassend und systematisch dargelegt wurden, und für das einige interessante Erweiterungen wie z.B. die Integration mit einem Debugger [Mukherjea 94] entwickelt wurden.
In Übersichtsartikeln wird u.a. über weitere Aktivitäten im Bereich der Visualisierung von Informatiklehrinhalten berichtet [Price, Baecker, Small 93; Roman 93].

5. Gewonnenen Erkenntnisse, Ausblick auf zukünftige Aktivitäten

Bei der Betrachtung vieler über das Internet verfügbarer Animationen bekamen wir den Eindruck, dass für einen Teil dieser Animationen gilt:

- Einige Animationen stellen triviale Sachverhalte dar (Beispiel: Das Wandern von Daten durch die Schichten des OSI-Modells von oben nach unten und dann anschließend von unten nach oben). Solche Animationen bringen den Studierenden keine Erkenntnisse, die diesen nicht auch auf herkömmliche Art und Weise vermittelt werden könnten. Hier werden Animationen unserer Meinung nach nur der Effekthascherei wegen eingesetzt.
- Ein anderer Teil von Animationen zeigen nach dem Start eine verwirrende Fülle von Vorgängen, die von den betrachtenden Personen nicht verstanden werden können. Die durchgeführten Änderungen sind abrupt oder zu schnell. Oft erfolgen auch mehrere Veränderungen an unterschiedlichen Stellen gleichzeitig. Der Ablauf kann von den betrachtenden Personen nicht nachvollzogen werden. Dies sind sicherlich einige der Gründe, weshalb die Erfahrungen über den Einsatz von Animationen in der Lehre teilweise enttäuschend sind [Stasko 93].

Folgende Konsequenzen aus diesen Erkenntnissen versuchten wir, in den entwickelten Animationen umzusetzen:

- Man sollte sorgfältig die Sachverhalte auswählen, die durch Animationen visualisiert werden. Besonders gut geeignet sind Sachverhalte, die relativ komplex sind und die sich anhand von Beispielen gut erklären lassen:
  Auf der einen Seite lassen sich viele Sachverhalte ohne Verluste auch auf herkömmlichen Weg erklären. Wenn für jede Kleinigkeit eine Animation gezeigt

wird, dann stumpfen die Lernenden mit der Zeit ab; die Animationen haben ihren Reiz verloren, da die Studierenden sich daran gewöhnt haben. Auf der anderen Seite können Animationen unserer Meinung nach nicht die Beherrschung theoretischer und formaler Verfahren und Methoden "auf dem Königsweg" bieten. Die gründliche Auseinandersetzung der Studierenden mit dem Lernstoff, das eigenständige Denken und Grübeln kann den Studierenden nicht erspart bleiben.

- Animationen müssen sehr stark von den benutzenden Personen gesteuert werden. Eine Animation soll nicht wie ein Film nur betrachtet werden:
  Die manuelle Steuerung hat zwei Vorteile: Zum einen kann damit die Ablaufgeschwindigkeit reguliert werden. Es muss nach einem gesehenen Effekt die Animation beliebig lang gestoppt werden können, um über den gesehenen Effekt nachdenken zu können bzw. in einer Vorlesung den Effekt mit den Studierenden zu diskutieren. Zum anderen erzwingt eine explizite Angabe des nächsten Schrittes die betrachtenden Personen, sich aktiv mit der Animation auseinanderzusetzen. In vielen Diskussionen mussten die Entwickler der Animationen überzeugt werden, manche Automatismen, die technisch einfach machbar gewesen wären, nicht in die Animationen einzubauen, sondern diese Steuerungsaufgabe den späteren Nutzern zu überlassen. Hier zeigt sich die alte Erkenntnis, dass nicht alles, was technisch machbar ist, in einer konkreten Situation auch nützlich sein muss.
- Es sollen in der Lehre nicht nur fertige Animationen "konsumiert" werden, sondern die Studierenden sollen mit verhältnismäßig geringem Aufwand eigene Animationen "produzieren":
  Den Studierenden wird mit dem ***Vivaldi***-Kern eine Plattform zur Verfügung gestellt, mit der sie eigene Programme visualisieren können, wobei in diesem Fall die Visualisierung nicht für andere gedacht ist, sondern primär für sie selbst. Stehen den Studierenden z.B. entsprechende Visualisierungsoperationen zum Vergleich zweier Elemente und zum Vertauschen zweier Elemente zur Verfügung, so können sie selber unterschiedliche Sortieralgorithmen programmieren und an entsprechender Stelle die zur Verfügung gestellten Visualisierungsoperationen in ihr Programm einbauen (d.h. sie können ihr Sortierprogramm zum Zwecke der Visualisierung instrumentieren). In diesem Fall kann die Visualisierung zum schnellen Erkennen von Programmierfehlern dienen.

In Zukunft sind folgende weitere Aktivitäten geplant:

- Die Animationen sollen in unterschiedlichen Veranstaltungen an der FH Trier eingesetzt werden (z.B. Vorlesungen des Studiengangs "Angewandte Informatik", Präsenzphase des Fernstudiums "Allgemeine Informatik"). Dabei sollen zum einen vorbereitete Animationen während der Vorlesung präsentiert werden. Zum anderen sollen die Studierenden vorgegebene Visualisierungsoperationen in ihre selbst entwickelten Programme einbauen, um damit besser erkennen zu können, ob ihre Programme auch das ausführen, was sie sollen.
- In Software-Engineering-Projekten, Projekt- und Diplomarbeiten sollen weitere Animationen erstellt und bestehende verbessert werden.

Danksagung: Wir bedanken uns bei allen Studierenden, die Beiträge zum *Vivaldi*-Projekt geliefert haben, vor allem bei Bernd Greve, Torsten Martin, Christoph Mettler, Thomas Schmitt und Markus Sell. Außerdem bedanken wir uns bei den Förderern des *Vivaldi*-Projekts, der Nikolaus-Koch-Stiftung in Trier und dem Ministerium für Bildung, Wissenschaft und Weiterbildung Rheinland-Pfalz in Mainz.

6. Literatur

[Brown 84] M.H. Brown, R. Sedgewick: A System for Algorithm Animation, Computer Graphics, July 1984, pp. 177-186.

[Brown 88a] M.H. Brown: Algorithm Animation, MIT Press, Cambridge, Mass., 1988.

[Brown 88b] M.H. Brown: Exploring Algorithms Using Balsa-II, Computer, Vol. 21, No. 5, May 1988, pp. 14-36.

[Brown 91] M.H. Brown: Zeus: A System for Algorithm Animation and MultiView Editing, Proceedings IEEE Workshop Visual Languages, IEEE CS Press, Los Alamitos, California, 1991, pp. 4-9.

[Gamma et al. 95] E. Gamma, E. Helm, R. Johnson, J. Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[Mukherjea 94] S. Mukherjea, J.T. Stasko: Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger, ACM Transactions on Computer-Human Interaction, Vol. 1, No. 3, September 1994, pp. 215-244.

[Oechsle 99] R. Oechsle: Basiskonzepte des Visualisierungsprojekts ***Vivaldi***, Interner Bericht 99/01, Fachbereich Angewandte Informatik, FH Trier, Januar 1999.

[Oechsle, Becker 99] R. Oechsle, R. Becker: Das Projekt ***Vivaldi*** (*Vi*sualisierung *v*on *a*usgewählten *L*ehrinhalten *d*er *I*nformatik), 15. GI/ITG-Fachtagung "Architektur von Rechnensystemen" (ARCS '99) und "Arbeitsplatzrechensysteme" (APS '99), Friedrich-Schiller-Universität Jena, 4. – 7. Oktober 1999, pp. 263-270.

[Price, Baecker, Small 93] B.A. Price, R.M. Baecker, I.S. Small: A Principled Taxonomy of Software Visualization, Journal of Visual Languages and Computing 3, 1993, pp. 211-266.

[Roman 93] G.-C. Roman, K.C. Cox: A Taxonomy of Program Visualization Systems, IEEE Computer, Vol. 26, No. 12, December 1993, pp. 11-24.

[Stasko 89] J.T. Stasko: Tango: A Framework and System for Algorithm Animation, PhD Thesis, Brown University, Providence, RI, May 1989.

[Stasko 90] J.T. Stasko: Tango: A Framework and System for Algorithm Animation, IEEE Computer, Vol. 23, No. 9, September 1990, pp. 27-39.

[Stasko 93] J.T. Stasko, A. Badre, C. Lewis: Do Algorithm Animation Assist Learning? An Empirical Study and Analysis, Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems, Amsterdam, Netherlands, April 1993.

Ein System zur Visualisierung von Algorithmen und Datenstrukturen mit LEDA

Stefan Näher[1] and Oliver Zlotowski[2]

[1] Fachbereich IV-Informatik
Universität Trier
D-54286 Trier

[2] Martin-Luther-Universität Halle-Wittenberg
Fachbereich Mathematik und Informatik
Institut für Informatik
D-06099 Halle(Saale)

Zusammenfassung Diese Arbeit beschreibt einige grundlegende Konzepte und Implementierungsdetails des LEDA SV-Systems zur Visualisierung von Datenstrukturen und Algorithmen. Die wichtigsten Ziele dieses Projektes sind zum einen die Bereitstellung eines Systems zur Visualisierung von Verfahren, die für den Einsatz in der Lehre geeignet sind und zum anderen die Schaffung einer Plattform zur Unterstützung von Experimenten im Bereich des Algorithms Engineering. Zur Visualisierung werden neben der plattformunabhängigen Fensterklasse auch die Visualisierungswerkzeuge GraphWin[1] und GeoWin[2] von LEDA verwendet, so dass zusammen mit der Vielzahl von Algorithmen und Datenstrukturen in LEDA eine gute Ausgangsposition für die Entwicklung eines SV-Systems gegeben ist.

1 Einleitung

Die Leistung moderner Computersysteme hat die Entwicklung rechnergestützter Darstellungen komplizierter Sachverhalte in den letzten Jahren stark beschleunigt. So sind viele Bereiche der Industrie und Wirtschaft in hohem Maße von rechnergestützten Visualisierungen komplexer Sachverhalte abhängig, nicht zuletzt deshalb, weil „Computersysteme [...] Informationen als abstrakte Symbole [repräsentieren], [wobei] ihre Repräsentationsformen [...] in der Regel darauf ausgelegt [sind], Berechnungen und Analyseoperationen möglich zu machen und den Informationszugriff zu erleichtern." [RS99]

Die Visualisierung von Software ist ein Vertreter rechnergestützter Visualisierungen, wobei in den folgenden Betrachtungen ein Teilgebiet der Software Visualisierung (kurz: SV) im Vordergrund steht: die Animationen.

In der Lehre werden Animationen mit dem Ziel eingesetzt, das Verständnis der Funktionsweise von Algorithmen und Datenstrukturen zu fördern. Aus

[1] siehe Kapitel 2 in [MN99]
[2] siehe [Bae]

pädagogischer Sicht gehört zu den wichtigsten Vorteilen rechnergestützter Darstellungen die Möglichkeit des Einsatzes von bewegten Grafiken, die durch den Benutzer interaktiv gesteuert werden können [MH94]. Hierbei lassen sich die Vorteile eines Buches (Vor-/Zurückblättern und Bestimmung der persönlichen Lerngeschwindigkeit) mit denen eines Films (bewegte Grafiken, Töne und Spezialeffekte) auf sinnvolle und kreative Art verbinden.

Neben der Kreativität des Entwicklers von Animationen hat aber auch die Beachtung mediendidaktischer und psychologischer Aspekte bei der Erstellung einen entscheidenden Einfluss auf die Qualität und den Nutzen einer Animation. Aus dem Bereich der Pädagogik und der Informatik gibt es daher zahlreiche Erfahrungen mit dem Einsatz von Animationen in der Lehre. Der interessierte Leser sei hier beispielsweise auf [BCS96,LBS94,SDBP98,KST99] verwiesen.

Der nächste Abschnitt gibt auf Basis der in [PBS93] vorgestellten Taxonomie einen kurzen Überblick über die Teilbereiche der SV. Im Anschluss daran werden einige im Rahmen einer laufenden Diplomarbeit erstellten Grundkomponenten für die Entwicklung von Animationen unter Verwendung von LEDA[3] vorgestellt und an einem Beispiel ihre Verwendung demonstriert.

2 Teilgebiete der Software Visualisierung

Das Ziel der Software Visualisierung besteht im allgemeinen darin, den Verstehensprozess für die Arbeitsweise von Software zu erleichtern. Um dieses Ziel zu erreichen, werden häufig Techniken aus der Typographie und der Kinematographie angewandt und mit modernen Methoden der Mensch-Computer-Interaktion gekoppelt [SDBP98].

Die SV lässt sich, wie bereits erwähnt, in verschiedene Teilbereiche gliedern, die in der folgenden Abbildung dargestellt sind:

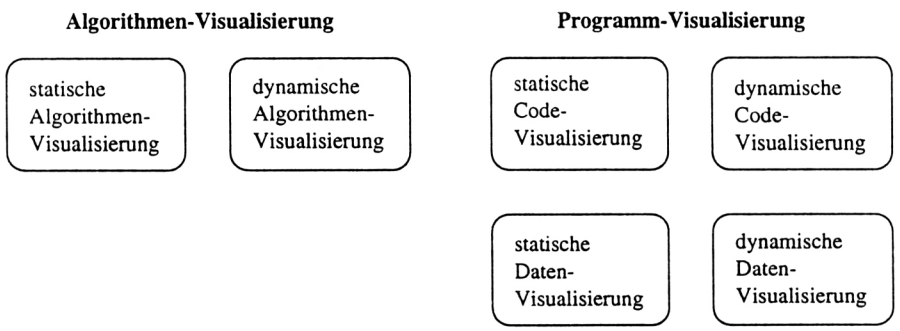


Abbildung1. Die Teilgebiete der Software Visualisierung [PBS93].

[3] Library of Efficient Data types and Algorithms [MN99]

Die Visualisierung von Algorithmen beinhaltet hierbei die Beschreibung von Software auf einem hohen Abstraktionsgrad, d.h. dass im wesentlichen nur strukturelle Ähnlichkeiten zwischen den Darstellungselementen und dem implementierten Verfahren bei Vertretern dieses Visualisierungstyps zu finden sind. Das Teilgebiet lässt sich weiter in statische- und dynamische Algorithmen-Visualisierung untergliedern, wobei letzterer Visualisierungstyp auch als Algorithmen-Animation bezeichnet wird. Das Video *Sorting Out Sorting* von BAECKER vorgestellt in [Bae81] gilt als eines der ersten Vertreter von Algorithmen-Animationen. In diesem Video wird die Arbeitsweise verschiedener Sortierverfahren mit Hilfe einfacher animierter Darstellungen erläutert.

Ein einfaches Beispiel für eine statische Algorithmen-Visualisierung ist ein Programmablaufplan. Im Sinne des Begriffes Abstraktion werden in der Regel nur wesentliche Verfahrensschritte in statischer Form unter Verwendung einer grafischen Symbolik dargestellt. Jeder Bezug auf eine konkrete Programmiersprache oder Implementierungsvariante wird dadurch vermieden. In Lehrbüchern findet dieser Visualisierungstyp oft Anwendung.

Auf der rechten Seite in der Abbildung 1 ist das zweite Teilgebiet der SV dargestellt, die Programm-Visualisierung. Der Unterschied zwischen den beiden Visualisierungstypen soll durch eine Definition des Algorithmenbegriffes verdeutlicht werden: Algorithmen sind exakt formulierte Rechenvorschriften, die auf Computern als ausführbare Programme implementiert und auf diese Art für die Lösung von Aufgaben angewandt werden können. „Ein Algorithmus besteht somit aus einer wohldefinierten endlichen Folge von elementaren Rechenoperationen und Entscheidungen, um aus einer bestimmten Menge von Eingabegrößen [...] das gewünschte Resultat [...] zu erzeugen." [Zei96]

Programme sind hiernach nichts anderes als Implementierungsvarianten von Algorithmen. Im Vordergrund einer Programm-Visualisierung steht somit die Darstellung einer konkreten Implementierung eines Algorithmus. Das wesentliche Unterscheidungsmerkmal zwischen beiden Visualisierungstypen ist deshalb der Abstraktionsgrad, der bei einer Programm-Visualisierung in der Regel geringer ist als bei einer Algorithmen-Visualisierung. Die Untergliederung des Teilgebietes in Code- und Daten-Visualisierung weist zusätzlich auf die Techniken der Programm-Visualisierung hin. Die Einteilung erscheint sinnvoll, da zur Verwaltung und Berechnung von Daten oft charakteristische Datenstrukturen im Programm verwendet werden, deren Arbeitsweise in gesonderten Darstellungen die Vermittlung der Grundidee einer konkreten Implementierung unterstützen können. Daten-Visualisierungen sind sehr vielfältig. So kann beispielsweise eine Liste durch Rechtecke dargestellt werden, wobei die Verkettung der Listenelemente intuitiv durch Pfeile erfolgt. Werden zwei Elemente in einem Verfahrensschritt vertauscht, animiert die dynamische Daten-Visualisierung diesen Austausch durch einen graduellen Vorgang.

Was ist nun eine Code-Visualisierung? Ein einfaches Beispiel für eine statische Code-Visualisierung ist Darstellung des Programmtextes. Das kontinuierliche Hervorheben des gerade aktiven Programmteils kann als Code-Animation interpretiert werden. Auch die Darstellung eines Rekursionsbaumes bei einem

rekursiven Verfahren kann als ein Vertreter dieses Visualisierungstyps aufgefasst werden.

Abschliessend sei noch daraufhingewiesen, dass in der Praxis bei vielen existierenden Animationen keine klare Zuordnung zu einem der Teilgebiete vorgenommen werden kann, da viele Visualisierungen sowohl stark abstrahierende, als auch sehr an der Implementierung orientierte Darstellungselemente besitzen.

3 Komponenten des LEDA SV-Systems

In diesem Abschnitt werden einige der umgesetzten Konzepte anhand bereits implementierter Komponenten erläutert. Doch zuvor wird die Frage beantwortet, warum LEDA als Plattform für die Entwicklung eines SV-Systems geeignet erscheint.

Aus der Sicht eines Software-Entwicklers ist LEDA eine Plattform für die Lösung kombinatorischer und geometrischer Probleme. Durch eine einfache Transformation ist der Entwickler in der Lage, Algorithmen in effiziente C++-Programme zu überführen, da ihm hierfür eine Vielzahl von einfachen (Stack, Liste) und komplexen (Graph, Wörterbuch) Datenstrukturen zur Verfügung stehen. Diese Datenstrukturen sind als unabhängige Datentypen in Form von C++-Klassen implementiert. Dadurch hat der Entwickler die Möglichkeit über die Schnittstelle der Klasse einer Datenstruktur charakteristische Operationen auf ihr auszuführen.

Eine Erweiterung der in sich gekapselten Strukturen um neue Funktionalität ist durch das Vererbungskonzept von C++ möglich. Diese Programmiermethodik wird bei den folgenden Komponenten immer wieder angewendet, um den großen Umfang bestehender Datentypen in LEDA für Visualisierungen nutzbar zu machen.

Die einfache Programmierung der grafische Schnittstelle ist ein weiteres Merkmal von LEDA. Unter Verwendung der Fensterklasse hat der Entwickler die Möglichkeit verschiedene grafische Schnittstellensysteme (Windows 95/98/NT, Unix/Linux) zu nutzen.

Die große Menge an Datenstrukturen sowie die einfache Grafikprogrammierung sind zwei wesentliche Punkte, die den Versuch des Designs eines SV-Systems auf der Basis von LEDA sinnvoll erscheinen lassen. Für einen tieferen Einblick in das LEDA System sei der interessierte Leser auf [MN99,MNSU99] verwiesen.

Bevor im folgenden einige konkrete Konzepte erläutert werden, formuliert der nächste Abschnitt die vorläufigen Entwicklungsziele für das System.

3.1 Entwicklungsziele

Viele untersuchte Systeme wie Balsa [Bro88], Zeus [Bro91], Tango [Sta92,Sta89] oder Polka [Sta93], sowie die in [PBS93] vorgestellte Taxonomie beeinflussten die Herausarbeitung der folgenden Entwicklungsziele:

- ▷ *Wiederverwendbare und unabhängige Komponenten*
  Die implementierten Komponenten sollten für verschiedene Algorithmen einer

Algorithmenklasse wiederverwendbar sein und in einer dokumentierten Bibliothek zusammengefasst werden. Dies betrifft insbesondere die *Views* sowie ihre Steuerung und Verwaltung.

- ▷ *Einfache Erweiterung bzw. Anpassung der Visualisierungskomponenten*
  Die Konzepte sollten so umgesetzt werden, dass neue Funktionalität (z.B. eine neue Darstellung) dem System hinzugefügt werden kann, ohne dass bestehende Komponenten modifiziert werden müssen.
- ▷ *Interaktionsschnittstellen*
  Über eine Interaktionsschnittstelle sollte es dem Anwender möglich sein, die Animation zu steuern, d.h. vorerst im wesentlichen die Eingabedaten sowie die Geschwindigkeit und die Richtung zu kontrollieren.
- ▷ *Minimale Modifikationen am Verfahren*
  Ein bereits unter Verwendung von LEDA Datenstrukturen implementiertes Verfahren sollte durch minimale Veränderungen in eine einfache Animation überführbar sein.

Das letzte Entwicklungsziel lässt sich nur bis zu einem gewissen Grad erreichen, denn komplexe Animationen, die insbesondere für den Einsatz in der Lehre konzipiert werden, benötigen zusätzliche Informationen, die vom Verfahren selbst nicht erzeugt werden. Diese abundanten Attribute einer Animation müssen somit zusätzlich in den Programmtext eingefügt werden. Daher erscheint es sinnvoll, auch für diese Art von Information, leicht verwendbare Visualisierungskomponenten zur Verfügung zu stellen. Zum Abschluss der Betrachtungen wird auf diese Problematik noch einmal kurz eingegangen.

3.2 Visualisierung von Datenstrukturen

Der wesentliche Vorteil bei der Visualisierung von Datenstrukturen besteht darin, dass ein Großteil der zur Visualisierung benötigten Informationen aus dem Zustand der Struktur gewonnen werden kann. Hierbei informiert die Datenstruktur, im folgenden auch als *Client* bezeichnet, durch das Auslösen verschiedener *Events* spezielle Komponenten über eine Zustandsveränderung. Diese Komponenten werden *Observer* genannt, deren Aufgabe es ist, die mit dem jeweiligen *Event* transportierten Informationen zu verarbeiten und eventuell an geeignete Visualisierungskomponenten (*Views*) oder an andere *Observer* weiterzuleiten.

Die Kommunikation zwischen dem *Client* und dem *Observer* ist ein typisches Anwendungsbeispiel für das Beobachtungsmuster (engl. *Observer Pattern*) aus [GHJV95]. Eine allgemeine Implementierungsvariante dieses Musters war unter anderem Gegenstand der Diplomarbeit von A. LULEICH. Mit der Implementierung des Datentyps `EVENTx` stellt LEDA eine Möglichkeit des $1 : n$ Informationsaustausches zwischen beliebigen Objekten zur Verfügung. Die Abbildung 2 zeigt das *Event*-System in der OMT-Notation[4].

An dieser Stelle wird auf die Funktionsweise des *Event*-Systems nicht näher eingegangen. Der interessierte Leser sei auf [Lul98] verwiesen. Statt dessen wird

[4] Object Modeling Technique

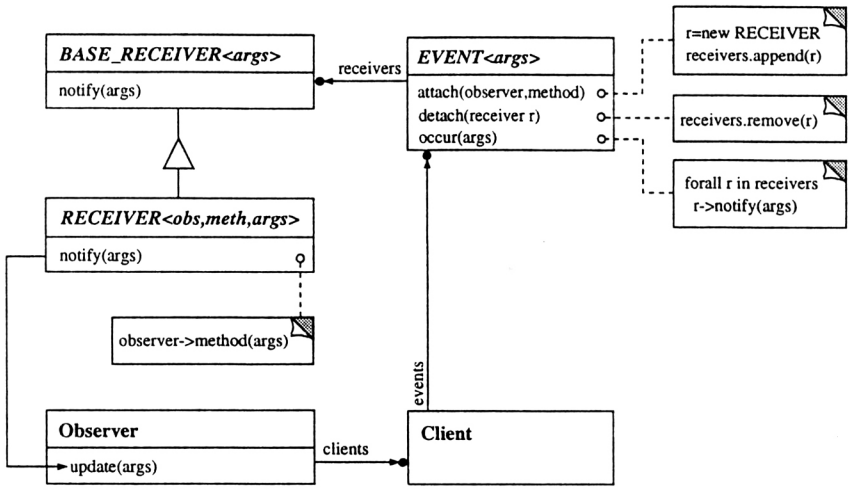


Abbildung2. Das Klassendiagramm des *Event*-Systems in LEDA [Lul98].

unter Benutzung dieses Systems und der LEDA Datenstruktur `array<E>` die Klasse `event_array<E>` betrachtet.

3.3 Event Array

Die Animation von Sortierverfahren ist ein beliebtes Einsatzgebiet eines SV-Systems, da dieser Typ von Algorithmus in der Regel auf Feldern arbeitet und sich Felder einfach visualisieren lassen. Aus diesem Grund dient der parametrierte LEDA Datentyp `array<E>` als Basis für die Template-Klasse `event_array<E>`, der ersten Grundkomponente des Visualisierungssystems. Der Template-Parameter `E` bestimmt im folgenden immer den vom Feld verwalteten Datentyp.

Durch Ableitung von der Klasse `array<E>` erbt die Klasse `event_array<E>` die Grundfunktionalität des Basistyps. Alle Member-Funktionen der Klasse `event_array<E>` (mit Argumenten) wurden zusätzlich um *Events* erweitert, so dass bei jedem Zugriff auf die Datenstruktur die zugehörigen *Events* ausgelöst werden.

Die Menge der definierten *Events* wird als *Event*-Kollektion bezeichnet, in der grundsätzlich zwei Typen zu unterscheiden sind:

- *Events zur Verwaltung*
  Diesem Typ von *Event* gehört der `construct`-, `destruct`- und der `update`-*Event* an. Sie dienen einem *Observer* zur Verwaltung seiner *Clients*. So kann sich ein *Client* an einen *Observer* an- bzw. abmelden. Das `update`-*Event* wird von der Funktion `enable_events()` ausgelöst und dient dazu, den *Observer* über eventuell komplexe Veränderungen in der Datenstruktur zu informieren, die ihm durch einen Aufruf der Funktion `disable_events()` entgangen sind.

▷ *Events in Member-Funktionen für den Zugriff auf das Feld*
Der zweite Typ von *Event* wird in Member-Funktionen ausgelöst, die entweder die Datenstruktur direkt verändern oder Informationen über einzelne Feldelemente zurückliefern. So löst zum Beispiel jede Vergleichsoperation einen `compare`-*Event* aus.

Durch einen `prev_Member_Funktions`-*Event* kann nun ein *Observer* beispielsweise die Gültigkeit der übergebenen Argumente überprüfen und bei ungültigen Argumenten (zumeist Bereichsüberschreitungen) das `_error`-Flag setzen, so dass nach der Rückkehr aus der *Event*-Behandlung die Basisfunktion nicht ausgeführt wird (vgl. Abbildung 3).

Eine zweite Funktion dieser *Events* besteht darin, einem *Observer* die Möglichkeit zu geben, den Zustand der Datenstruktur vor einer Veränderung zu speichern. Diese *Undo*-Information ist für eine Interaktionskomponente wertvoll, da hiermit Veränderungen in der Datenstruktur auch rückgängig gemacht werden können. Nachdem die Basisfunktion ausgeführt wurde, hat der *Observer* die Möglichkeit den aktuellen Zustand zum Zwecke einer *Redo*-Operation zu speichern, hierzu löst die Funktion den `post_Member_Funktions`-*Event* aus. Natürlich benötigen nicht alle Funktionen (z.B. `swap()`) einen `prev_`- und einen `post_Member_Funktions`-*Event*, um die beschriebenen Anwendungsmöglichkeiten zu gewährleisten.

Am Beispiel zweier Funktionen wird der Einsatz dieser *Events* gezeigt:

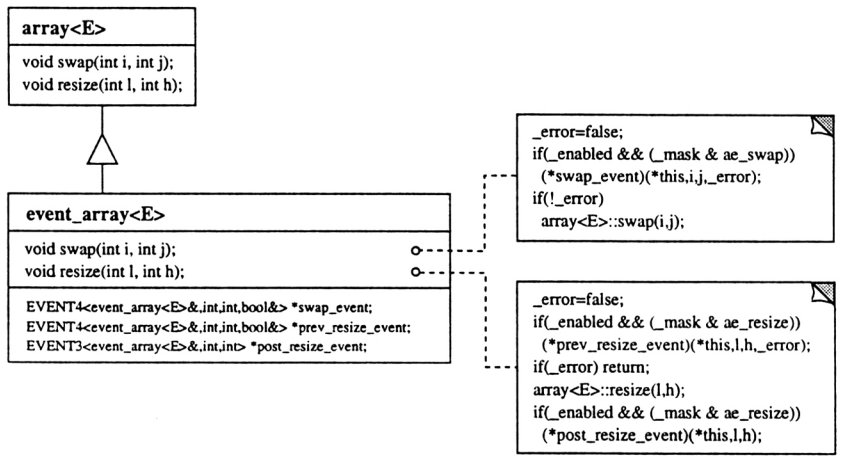


Abbildung3. Ein Klassendiagramm-Ausschnitt des Datentyps `event_array<E>`.

Nachdem nun ein *Client* betrachtet wurde, stellt der nächste Abschnitt einen passenden *Observer* vor.

3.4 Array Observer und Array View

Die Klasse `array_observer<E,View>` ist ein *Observer*, der nun auf die ausgelösten *Events* eines *Clients* reagiert und die mit dem jeweiligen *Event* transportierten Informationen an spezielle Visualisierungskomponenten weiterleitet. In Abbildung 4 sind die Beziehungen zwischen den einzelnen Klassen dargestellt:

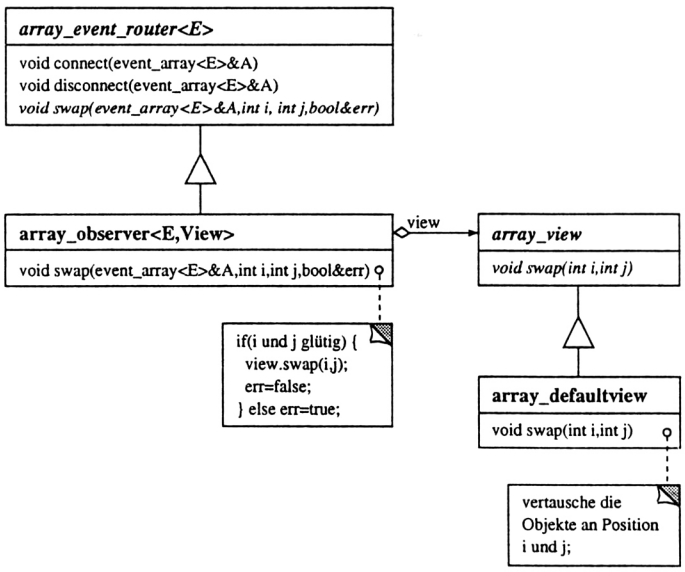


Abbildung4. Die Beziehungen zwischen den einzelnen Klassen.

Die Klasse `array_observer<E>` ist eine Unterklasse eines *Eventrouters*, dessen Funktionalität hier nicht näher betrachtet wird, da sich dessen Aufgabe im wesentlichen auf den Verbindungsaufbau `connect()` bzw. Verbindungsabbau `disconnect()` zu einzelnen *Clients* beschränkt.

Durch das Überschreiben der virtuellen *Event*-Funktionen der Basisklasse hat der *Observer* nun die Möglichkeit, gezielt Informationen über den Zugriff auf die Datenstruktur an die Visualisierungskomponente weiterzuleiten. In Abbildung 4 wird dieses Vorgehen am Beispiel der *Event*-Behandlung für den `swap`-*Event* gezeigt.

Die Klasse `array_view` stellt eine Basisklasse für verschiedene *Views* dar, so dass durch Schnittstellenvererbung die Klasse `array_defaultview` Informationen über einen *Client* empfangen und diese in einer grafischen Standard-Darstellung für kleine Felder verarbeiten kann.

Der folgende Programmausschnitt demonstriert die Anwendung der Klassen:

```
event_array<int> A(10);
array_defaultview View;
array_observer<int,array_defaultview> Observer(View);
Observer.connect(A);

// Initialisierung des Feldes und Öffnen des Views

A.swap(3,9);
```

Das vollständige Programm liefert die in Abbildung 5 dargestellte Momentaufnahme während des Austausches:

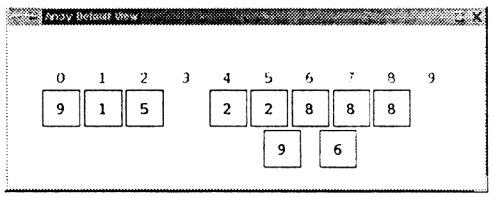


Abbildung5. Eine Momentaufnahme während eines Austausches.

Durch die Bereitstellung der Klasse `array_observer<E,View>` und `array_view` hat der Entwickler nun die Möglichkeit, spezielle Darstellungen für Felder zu entwickeln, wobei ihm dabei viele Informationen aus der Datenstruktur automatisch zur Verfügung gestellt werden, ohne dass dazu umfangreiche Anpassungen im Programmtext vorgenommen werden müssen.

Als Unterstützung für die Entwicklung eigener *Views* stehen dem Entwickler neben der Fensterklasse auch die Visualisierungswerkzeuge GraphWin und GeoWin zur Verfügung, die durch eine breite Funktionalität das Modellieren und Designen eines *Views* vereinfachen.

4 Anwendungsbeispiel: Heapsort

Bei der Animation des Sortierverfahrens Heapsort bietet sich neben der im letzten Abschnitt vorgestellten einfachen Feldsicht, auch die Sicht des Feldes aus der Baumperspektive an. Möglicherweise ist diese Sicht des Feldes auch sinnvoller, da Vertauschungen in der einfachen Feldsicht willkürlich wirken, in der Baumsicht jedoch die Idee des Verfahrens besser veranschaulicht wird.

Der folgende Programmtext erzeugt unter Verwendung eines weiteren *Views* (implementiert durch die Klasse `array_bintreeview`) und des bereits beschriebenen *Views* die dargestellte Momentaufnahme in Abbildung 6.

```
#include <LAAP/array_views.h>

event_array<int> A(1,15);
array_bintreeview TreeView("Baumsicht");
array_defaultview DefaultView("Feldsicht");
array_observer<int,array_bintreeview> Observer1(A,TreeView);
array_observer<int,array_defaultview> Observer2(A,DefaultView);
ao_client obs_client1,obs_client2;

void set_item_color(int i,color c=ivory)
{ TreeView.item_bg_color(obs_client1,i,c);
  DefaultView.item_bg_color(obs_client2,i,c);
}

void heapify(int k,int N)
{ set_item_color(k,red);      // Vater ist rot markiert
  while(2*k<=N) {
    int j=2*k;
    if(j<N && A.smaller(j,j+1)) j++;
    if(A.geq(k,j)) break;
    A.swap(k,j);
    k=j;
  }
  set_item_color(k);          // Vater wieder unmarkiert
}

int main()
{ obs_client1=Observer1.get_client(A);
  obs_client2=Observer2.get_client(A);

  // Initialisierung des Feldes und Öffnen der Views

  int N=A.high();
  for(int i=N/2;i>=A.low();i--)
    heapify(i,N);
  while(N>1) {
    A.swap(N,1);
    set_item_color(N,grey1);  // graue Feldelemente (sortiert)
    heapify(1,--N);
  }
  // ...
}
```

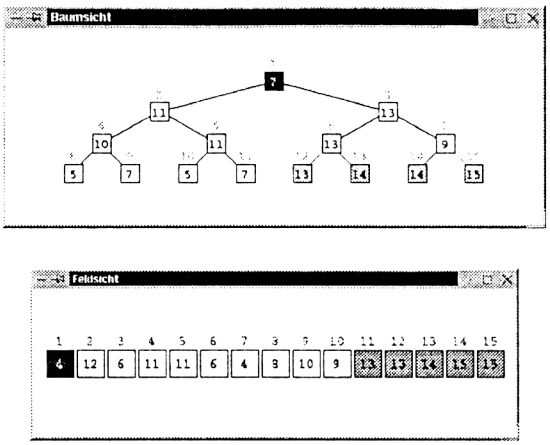


Abbildung6. Oben die Ausgabe von `TreeView` und unten die von `DefaultView`.

5 Zusammenfassung und Ausblick

Anhand des Beispiels wurde demonstriert, dass durch minimale Veränderungen im Programmtext eine einfache Algorithmen-Animation erstellt werden kann. Der Betrachter der Animation hat aber bislang noch keine Möglichkeit, mit der Animation zu interagieren. Für eine sinnvolle Bereitstellung einer solchen Interaktionsschnittstelle sind vielleicht weitere Veränderungen im Programmtext notwendig. Gegenwärtig werden für dieses Problem bereits vorgeschlagene Konzepte auf ihre Anwendbarkeit hin überprüft, so dass in naher Zukunft erste Fortschritte in diesem Bereich zu erwarten sind.

Durch das Bestreben beim Design des Systems möglichst allgemein verwendbare Konzepte zu finden, ist eine Übertragung des Vorgehens auf andere Datenstrukturen und damit andere Algorithmenklassen möglich. Ein Beispiel hierfür ist die LEDA Datenstruktur `graph`, in der ebenfalls eine *Event*-Kollektion definiert ist, so dass der zugehörige GraphWin-*Observer* die Veränderungen eines Graphen mit Hilfe des Visualisierungswerkzeuges GraphWin veranschaulichen kann.

Das allgemeine Ziel des Projektes ist die Bereitstellung eines modernen Systems zur Visualisierung von Algorithmen und Datenstrukturen, um es in der Lehre und bei Experimenten im Bereich des Algorithms Engineering einsetzen zu können.

Literatur

[Bae] M. Baesken. Interaktive Visualisierung geometrischer Algorithmen mit GeoWin. in Vorbereitung.

[Bae81] R. Baecker. Sorting Out Sorting, 1981.

[BCS96] M. Byrne, R. Catrambone, and J. Stasko. Do Algorithm Animations Aid Learning? Technical Report GIT-GVU-96-18, GVU Center, Georgia Institute of Technology, Atlanta, GA, August 1996.

[Bro88] M. Brown. *Algorithm Animation.* MIT Press, New York, 1988.

[Bro91] M. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 4–9, Kobe, Japan, October 1991.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 1995.

[KST99] C. Kehoe, J. Stasko, and A. Taylor. Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study. Technical Report GIT-GVU-99-10, GVU Center, Georgia Institute of Technology, Atlanta, März 1999.

[LBS94] A. Lawrence, A. Badre, and J. Stasko. Empirically Evaluating the Use of Animations to Teach Algorithms. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pages 48–54, St. Louis, MO, October 1994.

[Lul98] A. Luleich. Visualisierung von Graphalgorithmen. Master's thesis, Martin-Luther-Universität Halle-Wittenberg, Halle(Saale), Dezember 1998.

[MH94] A. Mester and P. Herrmann. Audiovisuelle Animation von verteilten Algorithmen und Kommunikationsprotokollen. Technical Report Projektgruppe 225, Fachbereich Informatik, Universität Dortmund, Dortmund, April 1994.

[MN99] K. Mehlhorn and S. Näher. *LEDA: A Plattform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[MNSU99] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual Version 4.0.* Saarbrücken, 1999.

[PBS93] B. Price, R. Baecker, and I. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, April 1993.

[RS99] K. Reichenberger and R. Steinmetz. Visualisierungen und ihre Rolle in Multimedia–Anwendungen. *Informatik Spektrum*, pages 88–97, April 1999.

[SDBP98] J. Stasko, J. Domingue, M. Brown, and B. Price. *Software Visualization.* MIT Press, Cambridge, London, 1998.

[Sta89] J. Stasko. *TANGO: A Framework and System for Algorithm Animations.* PhD thesis, Brown University,Providence, RI, May 1989. Available as Technical Report No. CS-89-30.

[Sta90] J. Stasko. The Path Transitions Paradigm: A Practical Methodology for Adding Animation to Program Interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, September 1990.

[Sta92] J. Stasko. Animating Algorithms with XTANGO. *SIGACT News*, 23(2):67–71, Spring 1992.

[Sta93] J. Stasko. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[Zei96] E. Zeidler. *Teubner-Taschenbuch der Mathematik.* B. G. Teubner, Stuttgart, Leipzig, 1996.

[Zlo99] O. Zlotowski. Animation der Sortierverfahren Heapsort und Quicksort unter Verwendung von LEDA. In *Informatikertage 1999, Fachwissenschaftlicher Informatik-Kongress*, pages 192–194, Bad Schussenried, März 1999.

Algorithmus-Animation in Lehr- und Lernsystemen der Computergraphik

Wolfram Luther

Fachgebiet Informatik, Fachbereich 11
Gerhard-Mercator-Universität Duisburg
Lotharstraße 65
D-47057 Duisburg

1 Einleitung

In fast allen Schulen und Hochschulen, aber auch im häuslichen Umfeld stehen heutzutage vernetzte Computer zur Verfügung, die dazu benutzt werden können, Lernenden den Umgang mit computer-basierten Medien zu vermitteln. Wir widmen uns einem speziellen Aspekt des Computereinsatzes im Unterricht, der Visualisierung von Datenstrukturen und Erklärung von Algorithmen in animierter Form. Gerade Algorithmen eignen sich durch ihren zumeist dynamischen Charakter sehr gut für die animierte Darstellung am Bildschirm, einer elektronischen Tafel oder Projektion und sind in einer klassischen Lehrveranstaltung an der Kreidetafel oder mit Hilfe von Overhead-Folien meist nur unzureichend zu vermitteln. Wir gehen dabei vor allem auch auf verschiedene Typen von Lehr- und Lernsystemen ein, definieren Anforderungen an die Animation von Algorithmen und zeigen Methoden auf, wie diese umgesetzt werden können. Gleichzeitig geben wir einen Überblick über den Stand unserer Projekte zur Erstellung von Lehr- und Lernumgebungen für Algorithmen aus dem Bereich der Computergraphik und Bildverarbeitung und ordnen sie in den Rahmen ähnlicher Aktivitäten in Deutschland und der Welt ein.

2 Eine Lernumgebung für Algorithmen der Computergraphik

Das Lehr- und Lernsystem ViACoBi (Visualisierung von Algorithmen der Computergraphik und Bildverarbeitung) [Janser 1997a, 1997b, 1998] widmet sich einem speziellen Aspekt des Computereinsatzes in der Lehre und ergänzt unmittelbar den klassischen Vorlesungsstoff zu 2D-Graphik und Grundlagen der Bildverarbeitung. Es wurde mit dem Multimedia-Autorensystem Authorware Professional für APPLE Macintosh entwickelt, auf Windows-Rechner portiert und baut auf einer Monographie zum selben Thema auf [Janser et al. 1996]. Weiterhin ist es durch eigenständige kleinere Systeme wie Grafik-Demo, HADES und MORPH, die in mehreren Seminar- und Abschlussarbeiten zu den Themen Low Level Routinen der Bildverarbeitung, fraktale Strukturen, Bintrees, Quadtrees und Octrees, Morphing sowie 3D-Rekonstruktion erstellt wurden, flankiert.

Letztere erlauben eher das freie Experimentieren mit Algorithmen und Datenstrukturen und verfügen über keinerlei Lernermodellierung oder Steuerung des Ablaufs durch das System. ViACoBi behandelt in acht Grundlagenkapiteln eine Darstellung des Arbeitsplatzes, Rasteralgorithmen für Strecke, Kreis und Ellipse und die Bearbeitung und Kompression von Grauwertbildern. Neben einer Aufbereitung des Lernstoffes in Hypertextform, der auch von einem Sprecher präsentiert wird, liegt das Schwergewicht auf einer Animation und Visualisierung der Wirkungsweise der Basis-Algorithmen. Zusätzlich enthält das System ein umfangreiches Glossar und eine Vielzahl von zum Teil automatisch generierten Aufgaben zum interaktiven Lösen. Folgende Aufgabentypen werden in ViACoBi unterstützt:

- Multiple Choice-Aufgaben,
- Aufgaben zum Erfassen visueller Zusammenhänge,
- Konstruktionsaufgaben,
- Aufgaben mit abrufbarer Musterlösung,
- Experimentelle Aufgaben.

Im Rahmen der Arbeiten am System und seiner Evaluation haben wir die traditionelle Unterrichtsorganisation modifiziert. Während statische Übersichten und beweisintensive Teile des Stoffes in der Vorlesung verbleiben und in einer Übung in Gruppen vertieft werden, sind wiederholende Zusammenfassungen, die Arbeit mit Graphikalgorithmen sowie eine Vielzahl verschiedener Aufgabentypen in eine *Praktische Übung* mit dem System ViACoBi verlagert worden.

Unsere Lehr- und Lernsysteme wurden auf verschiedenen Messen präsentiert, in Fortbildungen eingesetzt und parallel zur Entwicklung oder direkt im Anschluss an die Fertigstellung von kleineren heterogenen Lerngruppen bezüglich der folgenden Punkte in schriftlichen und mündlichen Befragungen mehrfach bewertet:

- Auswahl, Aufbereitung, inhaltliche Darstellung und Verständlichkeit des Lernstoffes
- Übersichtlichkeit und Folgerichtigkeit der Kursstruktur,
- adäquater Einsatz der Medien, Design, Benutzerführung,
- Verhältnis informierender, aktivierender und prüfender Inhalte,
- Bedienungsfreundlichkeit und Zuverlässigkeit.

3 Klassifizierung von Lehr- und Lernsystemen

Multimediale Lehr- und Lernsysteme können in mehrdimensionale Räume kognitiver Medien eingeordnet werden. Wesentliche Parameter sind

- der Grad der Lerneraktivität,
- der Anteil des Systems an Kontrolle und Steuerung,
- der Anteil von Präsentation von Lernstoffen durch das System im Vergleich zu kreativen Möglichkeiten des Lerners.

Die dreidimensionale Darstellung kognitiver Medien geht in dieser Form auf Klimsa [Klimsa 1995] zurück. Es ist allerdings zu bemerken, dass diese drei Parameter nicht unabhängig voneinander sind. Zudem sind auch weitere Parameter denkbar, wie der

Anteil oder das Gewicht verschiedener computerbasierter Medien oder die Vielfalt unterstützter Aufgabentypen.

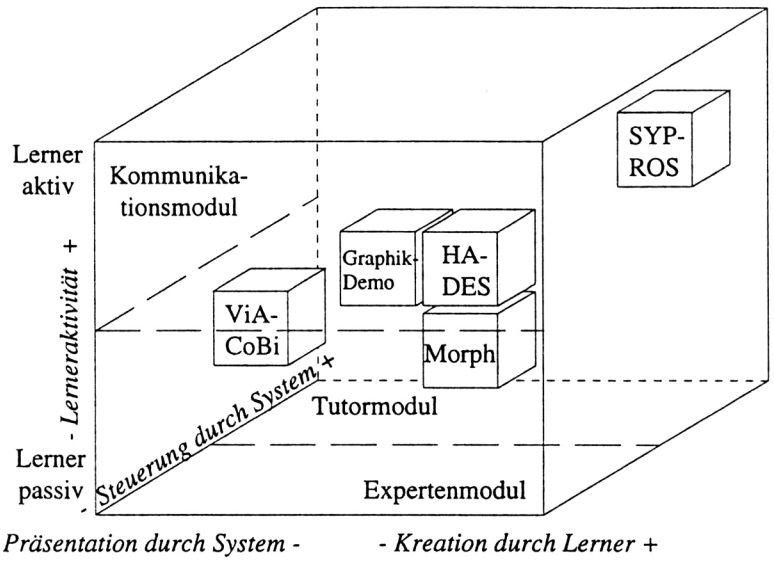


Abb. 1: Klassifizierung von Lehr- und Lernsystemen

Systeme zur Algorithmus-Animation wie ViACoBi sehen sich bezüglich der Lerneraktivität in einer mittleren Position. Innerhalb der Lektionen kann der Lerner frei navigieren und zu jeder Zeit Aufgaben lösen oder das Glossar konsultieren. Die Möglichkeiten der Interaktion werden im Rahmen eines *Kommunikationsmoduls* definiert. Es enthält Schnittstellen zur Steuerung des Lernverlaufs, zur Eingabe von Lösungen, Zugänge zu Hilfen. Glossaren und Ausgaben von Präsentationen und Animationen. Nutzer können im Rahmen von Aufgaben die Algorithmen in verschiedenen Parameterkonfigurationen austesten. Konstruktive Aufgaben sind didaktisch sinnvoll, setzen jedoch das Vorhandensein mächtiger Eingabeschnittstellen voraus, die mathematische Formeln parsen können. Auch ist dann die Anbindung von Computer-Algebrasystemen sinnvoll, die schrittweise Vorgehensweisen beim Lösen von Aufgaben unterstützen. Schließlich müssen 3D-Visualisierungswerkzeuge vorgesehen werden, da klassische Autorenprogramme im wesentlichen 2D-Graphik unterstützen.

Die Kreativität des Lerners ist somit recht eingeschränkt und kann nur in interessanten experimentellen Aufgaben mit einfachen Eingaben zur Geltung kommen. Hier entsteht jedoch ein neues Problem. Während bei Systemen, deren Lernstoffe aus großen oder verschiedenartigen Wissensdomänen stammen, Kontrolle und Steuerung eher beim Lerner verbleiben, müssen Systeme mit einer eigenen Kontrolle und Steuerung über ein *Expertenmodul* verfügen. In diesem ist das Wissen der vom System behandelten Domäne insoweit vollständig abgebildet, dass auf jede Eingabe des Lerners eine adäquate Reaktion erfolgt. Werden Aufgaben frei generiert, so muss das System jeden Schritt zu einer Lösung korrekt beurteilen und selbst eine korrekte

Lösung liefern. Daher haben wir in ViACoBi die Klasse aller korrekt abbildbaren Funktionen *KAF* der Menge der Farbbildmatrizen in sich oder auf einen *n*-stelligen Vektor von ganzen Maschinenzahlen definiert. Wir visualisieren und animieren die Klasse der korrekt abbildbaren Algorithmen *KAA*, die die Funktionen aus der Klasse *KAF* berechnen und dabei nur verifizierbare Operationen im Sinne des IEEE 754/854-Standards für die Gleitkommaoperationen ausführen.

Unter der semantischen Funktion *S(A)* eines Algorithmus *A* verstehen wir die Funktion *f*, die *A* auf seinen Eingabedaten berechnet. Dabei abstrahieren wir von der Programmsemantik eines Programms *P* und den beschriebenen Zustandstransformationen auf die Zustandsänderungen, die der Algorithmus auf den zugrunde liegenden Daten bewirkt. Wohl bemerkt machen wir keine Aussage zur Korrektheit der Algorithmen an sich und zum Problem, ob der Algorithmus die gewünschte Funktion berechnet, sondern nur zur Fragestellung, ob der Rechner auf einer realen Maschine das Ergebnis berechnet, das ein Rechner mit unendlicher Genauigkeit liefern würde. Solange sich die Rechnungen im Ganzzahlbereich bewegen und keine Bereichsüberschreitungen eintreten, können wir die Zugehörigkeit der Algorithmen zu unserer Klasse *KAA* garantieren. Im Bereich der Maschinenzahlen müssen wir kontrollierte Rundungen voraussetzen und Auslöschungen ausschließen. So sind alle in ViACoBi animierten Algorithmen aus *KAA*, und wir können selbst im Falle frei generierter Aufgaben zusichern, dass das System eine numerisch korrekte Lösung angibt.

Die Übernahme der Steuerung des Lernablaufs durch das System setzt das Vorhandensein eines *Tutormoduls* voraus, das grundlegende Strategien für das Vermitteln von Unterrichtsstoff nachbildet und in Abhängigkeit vom Kenntnisstand des Lerners das richtige Problem aussucht und adäquate Vorschläge zur Lösung macht. So zeichnet ein Tool zur Lernermodellierung Informationen darüber auf, in welchem Umfang der Lernende die Wissenselemente der Domäne bereits beherrscht, welche Arbeitsweisen er bevorzugt hat und welche Hilfestellungen er abruft.

Das intelligente Lehrsystem SYPROS, das derzeit unter der Leitung von Chr. Herzog [Rothenhöfer et al. 1993] an der TU-München entwickelt wird, ist für den Einsatz auf dem Lehrgebiet *Synchronisation paralleler Prozesse mit Semaphoren* konzipiert. Das für die Synchronisation grundlegende von Dijkstra eingeführte Konzept der Semaphore wird in SYPROS beispielhaft anhand von kooperierenden Prozessen eingeübt. Es ist ein herausragender Vertreter für ein intelligentes Lehr- und Lernsystem mit einer Lernermodellierung, einem Kommunikations-, Tutor- und Expertenmodul.

Zusammenfassend sind folgende Aspekte zu beachten:

- Interaktive multimediale Lehr- und Lernsysteme sind in der Entwicklung ausgesprochen zeitaufwändig (100 und mehr Arbeitsstunden sind für eine Kurseinheit von einer Stunde üblich).
- Systeme mit einer durchgehenden Lernermodellierung, Lernzielstrukturierung und intelligenten Lernerführung sind oft nur auf einen sehr engen Themenbereich begrenzt.
- Kurse zu umfangreichen Themenbereichen sind meist nur als freie Angebotssysteme mit eingeschränkter Lernermodellierung und Hilfestellung in Form von Anleitungen und Glossaren denkbar.

- Lehr- und Lernsoftware kann nur kostenrentabel erstellt werden, wenn sie in lernerintensiven Bereichen eventuell begleitend zu klassischer Lehre eingesetzt wird. Dabei muss der verantwortliche Dozent die Möglichkeit einer Aktualisierbarkeit und Erweiterbarkeit haben.
- Der erfolgreiche Einsatz multimedialer Lehr- und Lernsoftware ist auf eine moderne und leistungsfähige Infrastruktur an den Hochschulen angewiesen
- Die rechtlichen Probleme bei der Verwendung copyright-geschützter multimedialer Elemente müssen geklärt sein.

4 Animationsebenen

In [Janser, Luther 1994] haben wir wichtige Methoden und Anforderungen für die Algorithmus-Animation im Rahmen von Lehr- und Lernsystemen definiert, die in der Folgezeit in unseren Systemen umgesetzt wurden. Bei der visuellen Animation von Algorithmen unterscheiden wir zwei grundsätzliche Ebenen, die Animation von Daten und die Animation von Programmen.

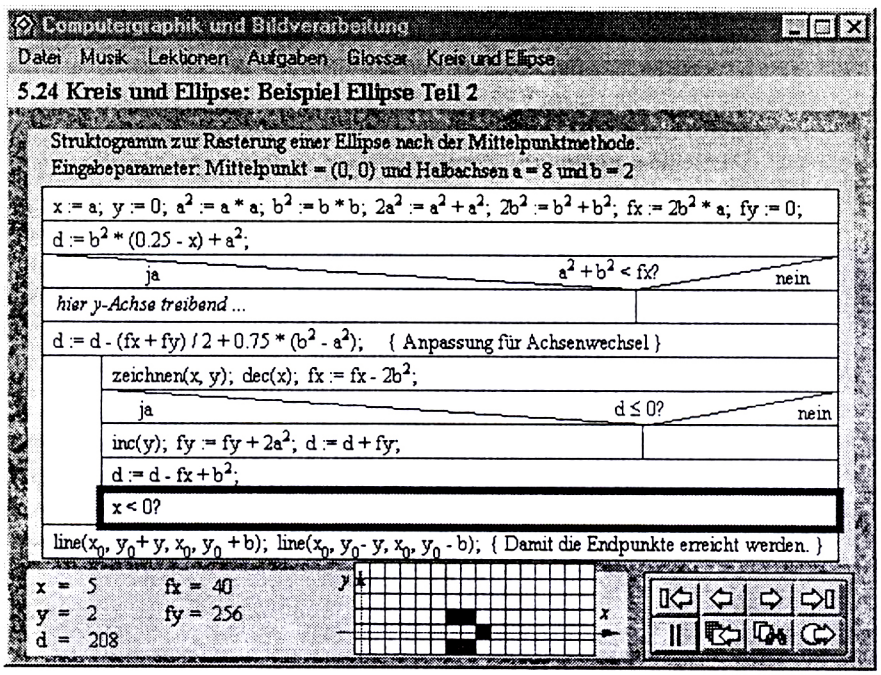


Abb. 2: Ellipsenrasterung in ViACoBi

Für die Animation eines Programms, das in einer Hochsprache geschrieben ist, bietet sich eine Darstellung in Diagrammform an, z. B. als Fluss- oder Nassi-Shneiderman-Diagramm. Ein animiertes rotes Fenster durchläuft das Struktogramm und zeigt

dem Benutzer, an welcher Stelle und in welchem Zustand sich der Algorithmus gerade befindet. Die Darstellung wichtiger Daten, Variablen, Parameter, Zähler, Flags und des Stacks bei der Abarbeitung, in unserem Fall der Punkt (x,y), die momentane Richtung (fx,fy) und die Steuervariable d, dient dem Überblick und dem Verständnis des Benutzers. Zusätzlich sollte der Algorithmus synchron kommentiert werden.

Der Vorteil dieser Animationsform liegt in ihrer Allgemeingültigkeit. Alle Algorithmen einer Hochsprache können mit derselben Technik und den gleichen Darstellungsmethoden visualisiert werden. Dafür liefert das Flussdiagramm nur ein sehr abstraktes Bild der Funktionsweise des Algorithmus. Die Wirkung der einzelnen Programmteile auf die bearbeiteten Daten bleibt dem Benutzer weitgehend verborgen, da diese Animationsform nur ein logisches Bild des Algorithmus liefert. Die Darstellung ist von ihrer Natur her noch als statisch zu bezeichnen.

Erst bei der Animation der vom Algorithmus manipulierten Datenmenge wird dem Benutzer die Möglichkeit gegeben, den Algorithmus intuitiv zu begreifen. Die genaue Implementierung durch das zugrunde liegende Programm, die im allgemeinen nicht wesentlich ist, wird dem Benutzer hierbei in der Regel nicht offen gelegt.

Für die Animation von Daten unterscheiden wir drei Ebenen, die wir als *Standbild, Analogie* und *Bewegtbild* bezeichnen.

Standbilder als unterste Ebene der Visualisierung unterteilen wir nochmals in *Abbildungen* und *logische Bilder.* Eine *Abbildung* ist ein realitätsnahes, zumeist in digitaler Form vorliegendes Bild des dargestellten Objekts, das mit einem Scanner, einer Stillvideokamera oder einem Videorecorder und einer Framegrabberkarte digitalisiert wird. *Logische Bilder* bieten eine erste Form der Abstraktion; sie dienen der vereinfachten Darstellung komplexer Zusammenhänge in Form von Graphiken oder Diagrammen.

Eine *Analogie* als abstraktere Darstellungsform liegt vor, wenn die Wirkungsweise des Algorithmus auf seine Daten dargestellt, von der Beschaffenheit der Daten jedoch abstrahiert wird. Darüber hinaus werden Analogien in Form von Modellen verwendet, um z. B. nicht direkt beobachtbare Strukturen zu modellieren und so komplexe Sachverhalte darstellen zu können.

Die aufwändigste Form der Animation stellt das *bewegte Bild* dar; es zeigt die Wirkung des Algorithmus auf die Modelldatenmenge. In unserer Abbildung 2 wird der Ellipsenbogen in einer groben Pixelebene sukzessive erzeugt.

Bewegte Bilder entstehen, indem einzelne Objekte einer Szene animiert oder fertig digitalisierte Videofilmsequenzen eingebunden werden. Allerdings sind sowohl für die Erstellung von Animationen wie auch für die Produktion von Videosequenzen zunächst die Hardwarevoraussetzungen zu schaffen.

5 Hilfs- und Stilmittel

Ausgehend von einer Algorithmus-Beschreibung in Pseudocode werden Algorithmen normalerweise möglichst effizient programmiert, um die Laufzeit und ggf. den Speicherbedarf der Programme und damit die Kosten zu minimieren. Dabei werden spezielle Möglichkeiten der eingesetzten Programmiersprache oder der Hardware verwendet, die die Ideen des Algorithmus verwässern und für sein eigentliches

Verständnis nicht von Bedeutung sind. Um dem Lerner einen Algorithmus zu erläutern, ist jedoch Effizienz in dieser Form nicht notwendig. Die Algorithmen müssen nicht laufzeit- und aufgrund der zumeist kleinen Modelldatenmenge auch nicht speicheroptimiert werden.

Das bedeutet jedoch nicht, dass über Schwächen der Algorithmen für gewisse Parameter-Konfigurationen hinweggesehen werden sollte, sie sollten im Gegenteil in der Lehrsoftware herausgearbeitet werden. Als Beispiel sei hier der *Bresenham*-Algorithmus zur Kurvenrasterung genannt, der im Falle von Kreisen im allgemeinen eine bestmögliche digitale Approximation des reellen Kreises liefert, jedoch für einige wenige Radien einen Defekt an den Winkelhalbierenden aufweist.

Im Gegensatz zu der im Programmieralltag üblichen Zielrichtung, Programme möglichst schnell und effizient zu gestalten, ist es in Lehrprogrammen oft sogar notwendig, die Programme zu verlangsamen, damit jeder einzelne Schritt am Bildschirm nachvollzogen werden kann. Gegebenenfalls muss der Benutzer die Möglichkeit haben, im Algorithmus einen oder mehrere Rückschritte zu machen, um sich den Fortgang des Algorithmus nochmals zu verdeutlichen. Die Steuerung sollte hierbei immer dem Benutzer überlassen werden, damit dieser seine Arbeitsgeschwindigkeit frei wählen und beliebig innerhalb des Systems navigieren kann.

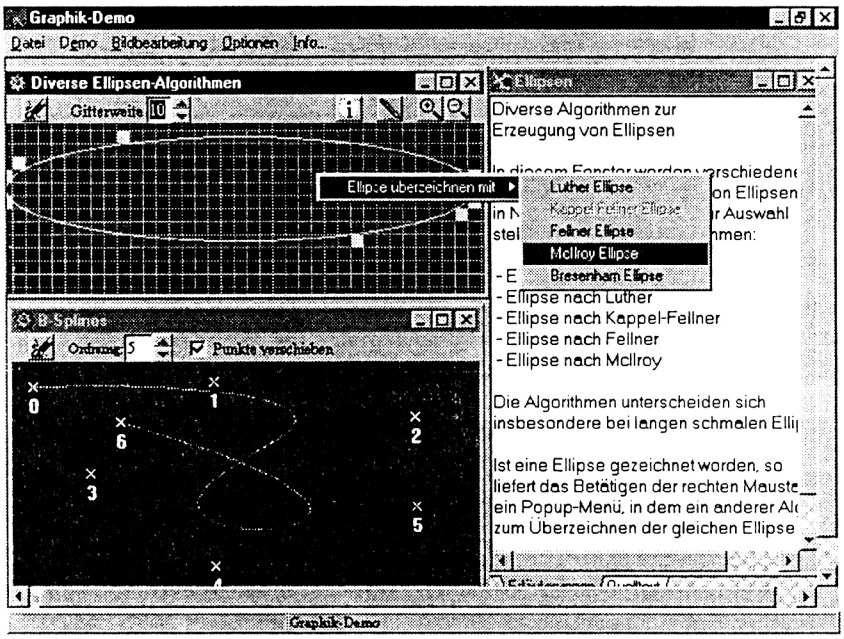


Abb. 3: Bildschirm aus dem Lerntool Graphik-Demo

Ein weiteres Stilmittel, das speziell auch bei der Animation graphischer Algorithmen eingesetzt wird, ist die Möglichkeit, Sachverhalte zu vergröbern. Bei der Animation eines Rastergraphik-basierten Algorithmus zum Zeichnen von Graphikprimitiven müssen die einzelnen Pixel stark vergrößert werden, damit der Benutzer überhaupt die

Möglichkeit hat zu sehen, wo die kritischen Punkte des Algorithmus liegen, wo der Programmierer Freiheiten nutzen kann und welche Pixel nach welchen Kriterien ausgewählt und gesetzt werden. Wichtige Details können zudem mit Mitteln der Form- und Farbgebung hervorgehoben werden.

Die einheitliche Färbung von wiederkehrenden Sachverhalten oder Objekten erleichtert dem Benutzer die Orientierung, signalisiert wichtige Zusammenhänge und löst Assoziationen aus. In dem mit Delphi entwickelten Tool Graphik-Demo (Abb. 3) ist die Rasterebene frei skalierbar. Durch verschiedene Farbgebung der Pixel können Ellipsen-Rasteralgorithmen in ihrer Funktionsweise durch Übereinanderlegung miteinander verglichen werden. Zugleich ist der Quelltext einsehbar. Eine andere Lektion betrifft B-Splines, die interaktiv durch Verschieben der Leitpunkte manipuliert werden können.

Einen ähnlichen Anspruch hat das OpenGL-basierte System MORPH [Ziegler 1998], das 2D-Morphing-Algorithmen unter einer intuitiven Benutzeroberfläche vorstellt und ihre Wirkungsweise mit den hier vorgestellten Methoden der Verlangsamung des Ablaufs, der Ausgabe und Visualisierung der Zwischenergebnisse und einer direkten Manipulierbarkeit der Parameter dem Lernenden näher bringt.

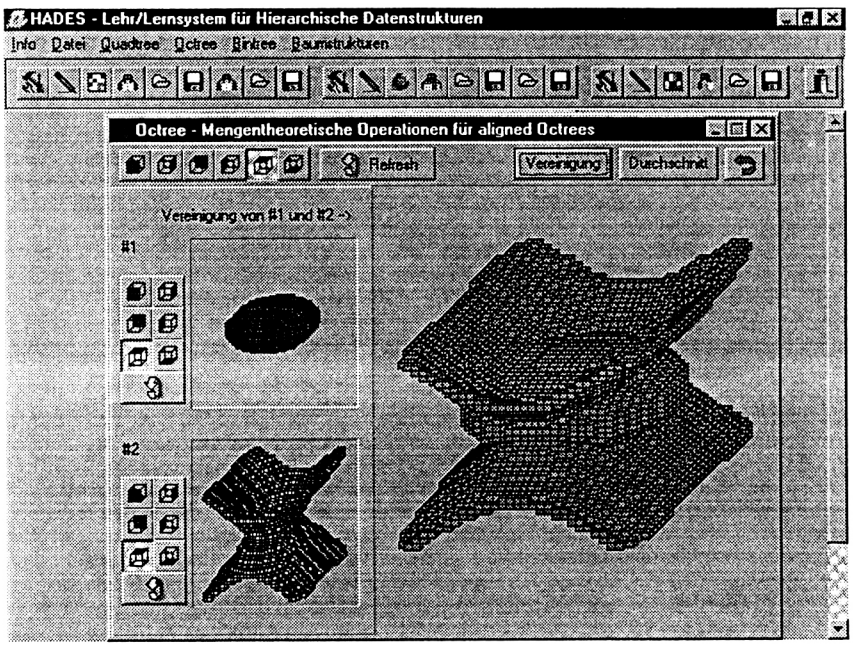


Abb. 4: Vereinigung zweier Octrees im System HADES

Bei dreidimensionalen Anwendungen stellt sich das Problem der Projektion. Hier ist eine geeignete Abbildung in die zweidimensionale Bildschirmebene zu wählen, so dass wichtige Inhalte nicht verloren gehen, die Darstellungsweise für den Betrachter

aber auch nicht zu komplex wird. Falls notwendig, kann eine Darstellung aus verschiedenen Perspektiven in mehreren Ebenen (entweder nebeneinander oder bei kleineren Bildschirmen nacheinander bzw. überlagert) erfolgen. Davon wird in unserem mit Delphi entwickelten System HADES [Klüttermann 1997] Gebrauch gemacht, das die Datenstrukturen *Bintree, Quadtree* und *Octree* visualisiert und grundlegende Konstruktionen, wie Vereinigung, Schnitt, Translation und Rotation, Darstellung von Polygonen und Flächen zweiter Ordnung durch Quadtrees bzw. Octrees animiert und visualisiert.

Textrepräsentationen am Bildschirm sind für den Betrachter ermüdend zu lesen. Eine Auflockerung des Textes durch Bilder und Graphiken schafft hier bereits Abhilfe. Noch ausdrucksstärker ist die Einbindung von gesprochenem Text und ggf. Ton- und Musiksequenzen. Multimediale Technik kann auf den Einsatz von gesprochenem, erklärendem Text nicht verzichten.

Eine Animationssoftware wie ViACoBi stellt es dem Benutzer frei, mit gesprochenem oder geschriebenem Text oder mit beidem zu arbeiten. Bei der Animation wird gesprochener Text auf wichtige Stellen beim Ablauf des Algorithmus hinweisen und ergänzende Erläuterungen vorsehen. An geeigneten Stellen können auch Ton- und Musiksequenzen eingebunden werden, um die Aufmerksamkeit des Benutzers auf bestimmte Einzelheiten zu lenken, das Wiedererkennen gleicher Situationen zu fördern oder Abläufe leitmotivisch zu strukturieren und Eindrücke zu verstärken.

Eine noch bessere Akzeptanz wird erreicht, wenn digitales Video eingebunden wird, um Praxisbeispiele einzuspielen und den Lerner stärker zu motivieren. Videosequenzen sprechen mehrere Sinne gleichzeitig an und wirken auf das menschliche Auge nicht so ermüdend wie Fließtext, durch den sich der Betrachter durcharbeiten muss. Die Steuerung des Ablaufs einer Algorithmus-Animation sollte möglichst beim Benutzer selbst liegen. Er muss die Möglichkeit haben, Abschnitte zu wiederholen (Replay), Querverbindungen zum Quelltext, zu weiteren Erläuterungen, einem Glossar oder zum Aufgabenteil zu nutzen (Hypertext) und in Interaktion mit dem Rechner sein erarbeitetes Wissen zu evaluieren.

6 Andere Systeme zur Algorithmus-Animation

Zu den Systemen, die im Bereich Algorithmus-Animation oder Computergraphik und Bildverarbeitung existieren, gehören die Systeme BALSA [Brown 1988] zur Algorithmus-Animation, LINK [Berry 1997] und ein WWW-basiertes System [Lüssem, Sünderkamp 1997] zum Arbeiten mit Baum- und Graph-Algorithmen. Eine ähnliche Thematik verfolgt das GAIGS-System [Naps 1997], das Algorithmen mittels einer Sequenz graphischer Schnappschüsse erklärt, die benutzergesteuert angezeigt werden. GRAPHICA ist ein System, das von Shabo, Guzdial und Stasko zeitlich parallel zu ViACoBi entwickelt und evaluiert worden ist. Es ist ebenfalls Web-basiert und unterstützt kollaboratives Lernen an ausgewählten Beispielen.

An der Universität Freiburg wird unter der Leitung von Th. Ottmann seit 1995 an Systemen zum Thema *Authoring on the Fly* gearbeitet [Bacher et al. 1997]. In diesem Rahmen sind auch CD-basierte Kurse zu Algorithmen und Datenstrukturen entstanden, die Sortier- und Baumalgorithmen animieren. Ziel weiterer Forschungen ist

111

es, die Animation nutzergesteuert ablaufen zu lassen und in jeder Phase persistente Annotationen zu erlauben.

Am Lehrstuhl von R.Wilhelm an der Universität Saarbrücken werden in Abschlussarbeiten, Dissertationen und Forschungsprojekten seit mehreren Jahren Lehr- und Lernsysteme mit Algorithmus-Visualisierung und -Animation zu den Themen Compilerbau und Programmiersprachen entwickelt und eingesetzt [Diehl et al. 1997, Kerren 1999]. Wesentliche Bemühungen gehen dahin, die Erstellung dieser Systeme zu automatisieren. Dazu wurde eine semi-automatische, Shape-basierte Programmanalyse eingeführt, die eine neue kondensierte Darstellung des Algorithmus in geclusterten Datenstrukturen eines Shape-Graphen mit aktiven und passiven Knoten ermöglicht. Dieser Ansatz erleichtert die Animation und Visualisierung klassischer Baum-basierter Algorithmen erheblich.

Unter der Leitung von J. Stasko sind die Systeme XTANGO, TANGO, POLKA und das Frontend SAMBA, das jetzt auch in einer Java-Version vorliegt, zur Algorithmus-Animation entwickelt worden. Das Ausgangssystem XTANGO erlaubt, aus dem Hochsprachenprogramm des Algorithmus wichtige Ereignisse zu isolieren und den Zustand der vom Algorithmus manipulierten Datenmenge in abstrakter Form zu visualisieren. Durch Animation der Einzelbilder kann die Wirkungsweise des Algorithmus illustriert und zu Lernzwecken eingesetzt werden. Eine umfangreiche Bibliographie befindet sich unter http://www.cc.gatech.edu/gvu/softviz/algoanim/.

MAPLE, Mathematica und MuPAD [Oevel et al. 1998] sind bekannte Computer-Algebrasysteme, die die Ausgabe von Hypermedia-Dokumenten mit Text, Graphiken, Formeln und Animationen ermöglichen. In den jüngsten Versionen können Funktionen und 3D-Objekte animiert werden, um Parameterabhängigkeiten zu visualisieren. Die verschiedenen Algorithmen sind unter Effizienzaspekten implementiert und besitzen Black-Box Charakter. Inkrementelle Vorgehensweisen beim Lösen von Aufgaben, Verlangsamung des Ablaufs und Skalierungen der Datenmengen sind nicht vorgesehen. Diese Programme sind daher nicht geeignet, die Wirkungsweise von Algorithmen zu vermitteln.

7 Ausblick auf Weiterentwicklungen und Web-basierte Lösungen

Wir haben Lehr- und Lernsysteme zur Computergraphik vorgestellt und zu ihrer Erstellung moderne Programmiersprachen, aber auch Autorensysteme wie Authorware oder Toolbook als Werkzeuge eingesetzt, die eine Aktualisierung und Erweiterung auch durch den Dozenten erlauben, der das Material verwendet. Allerdings bietet keines dieser Autorensysteme eine zufrieden stellende Integration leistungsfähiger Editoren, mächtiger Programmiersprachen oder formelmanipulierender Computer-Algebrasysteme. Daher müssen Hersteller solcher multimedialen Präsentationen auf die teure und ineffiziente Eigenentwicklung einzelner Komponenten zurückgreifen. So ist für eine Weiterentwicklung von ViACoBi die Integration einer leistungsfähigen Eingabeschnittstelle und des Computer-Algebrasystems MuPAD vorgesehen.

In [Hoppe et al. 1999] zeigen wir Möglichkeiten auf, wie Dozenten ihre Manuskripte unter Einsatz von Standard-Tools ohne großen technischen Aufwand in WWW-basierte Hypermediadokumente umsetzen können. Die Bereitstellung eines

Freihandtools auf einer elektronischen Tafel mit Stifteingabe ermöglicht im Gegensatz zu mit einer Präsentationssoftware erstellten elektronischen Folien einen schnellen Wechsel zwischen Präsentation und handschriftlicher Darstellung. Spontane Entwicklungen in der Vorlesung können automatisch in die vorbereiteten Dokumente eingebracht und später weiterverarbeitet werden. Ziel ist eine dynamische Darstellung der vom Dozenten entwickelten Herleitungen oder Algorithmen.

Dabei grenzen wir uns deutlich von CBT-Ansätzen, aufwändigen Multimedia- oder Authoring on the Fly-Produktionen mit der Betonung der Videokomponente auf Unix/MBone-Plattformen wie auch von Fernlernmaterialien ab. Den Lehrenden eröffnet sich ein Zugang mit niedriger Einstiegsschwelle zu moderner Informationstechnologie. Lernende erhalten aktuelle, mit multimedialen, interaktiven und dynamischen Anteilen angereicherte Hypermediadokumente zur Bearbeitung in der Präsenzphase und Weiterverwendung in einer individuellen Softwareumgebung.

Zur Anzeige von dynamischen Darstellungen bevorzugen wir Anzeigemodule (Viewer, Plug-Ins), z. B. für die dynamische XML-Sprache SMIL oder Java-Applets und Servlets, die Freihandeingaben direkt in das Skriptum integrieren.

So haben wir in Duisburg die Grundvorlesungen, Übungen und Praktika im Hauptstudiengang Lehramt Informatik und den Nebenfachstudiengängen zu den Informatischen Grundlagen Neuer Medien und Kommunikationstechniken, zur Informatik I mit Rechneraufbau, Rechnerstrukturen, imperativen, logischen und funktionalen Programmierparadigmen, zu den begleitenden Programmierkursen, zur Informatik II mit Algorithmen und Datenstrukturen, zur Theoretischen Informatik sowie einige weitere Vorlesungen aus dem Hauptstudium auf die neue Form umgestellt.

Beispielsweise werden die wichtigsten Sortieralgorithmen mit einem Applet visualisiert. Dabei sind sowohl die Anzahl der Elemente als auch die Geschwindigkeit parametrisiert, und es kann zwischen einem Fluss- und Schrittmodus hin- und hergeschaltet werden.

Dieser Ansatz wird zur Zeit im Projekt *InterActive Multimedia-Supported authoring for an Electronic Lecture hall* (AMSEL) weiterentwickelt.

8 Literatur

Bacher, Chr., Müller, R., Ottmann, Th., Will, M.: Authoring on the fly. In Hoppe, H. U. und Luther, W. (Hrsg.) 1997, 14-18.

Berry, J.: Improving Discrete Mathematics and Algorithms Curricula with LINK. ITiCSE'97, Uppsala, Sweden. SIGCSE/SIGCUE Proc. New York: ACM Press (1997), 14-20.

Brown, M. H.: Algorithm Animation. An ACM Distinguished Dissertation 1987. Massachusetts: The MIT Press (1988).

Diehl, S., Kunze, T., Placzek, A.: GANIMAM. Generierung interaktiver Animationen von abstrakten Maschinen. In Proc. Smalltalk und Java in Industrie und Ausbildung STJA'97, Erfurt 1997, 185-190.

GraphicaWebSide:
http://www.cc.gatech.edu/gvu/multimedia/nsfmmedia/graphics/edulib/CSA.html

Hoppe, H.U., Luther, W. (Hrsg.): Informatik und Lernen in der Informationsgesellschaft. 7. GI-Fachtagung Informatik und Schule; Duisburg, September 1997. Reihe Informatik aktuell. Berlin: Springer (1997).

Hoppe, H.U., Luther, W., Mühlenbrock, M., Otten, W., Tewissen, F.: Interactive Presentation Support for an Electronic Lecture Hall - a practice report. ICCE'99, Chiba (Japan), November 1999. Advanced Research in Computers and Communications in Education. G. Cumming et al. (Eds.), IOS Press (1999), 923-930.

Janser, A., Luther, W.: Algorithmus-Animation als Hilfsmittel der Informatikausbildung. In Hoppe, H.U. und Luther, W. (Hrsg.): Neue Lehr-/Lernsysteme aus informatischer und fachdidaktischer Sicht. Duisburg: SM-DU-266 (1994), 20 S.

Janser, A., Luther, W., Otten, W.: Computergraphik und Bildverarbeitung. Braunschweig/Wiesbaden: Vieweg (1996), 360 S. mit Hybrid-CD.

Janser, A.: An interactive learning system visualizing computer graphics algorithms. ITICSE'97, Uppsala, Schweden 1997. SIGCSE/SIGCUE Proceedings New York: ACM Press (1997), 21-23.

Janser, A.: Ein Lehr-/Lernsystem zur Computergraphik und Bildverarbeitung mit unterschiedlichen Aufgabentypen. In Hoppe, H.U., Luther, W. (Hrsg.) 1997, 77-88.

Janser, A.: Entwurf, Implementierung und Evaluierung des interaktiven Lehr- und Lernsystems ViACoBi für die Visualisierung von Algorithmen der Computergraphik und Bildverarbeitung. Dissertation, Gerhard-Mercator-Universität Duisburg, Gesamthochschule. Berlin: Logos (1998), ISBM 3-89722-065-2.

Kerren, A.: Animation der semantischen Analyse. In A. Schwill (Hrsg.) Informatik und Schule. Fachspezifische und fachübergreifende didaktische Konzepte. 8. GI-Fachtagung Informatik und Schule; Potsdam, September 1999. Reihe Informatik aktuell. Berlin: Springer (1999), 108-119.

Klimsa, P.: Multimedia aus psychologischer und didaktischer Sicht. In Issing, L. und Klimsa, P. (Hrsg.): Information und Lernen mit Multimedia. Weinheim: Beltz Psychologie-Verlags-Union (1995), 7-24.

Klüttermann, K.: Grundlegende Algorithmen von Quadtrees und Octrees implementiert unter Delphi in Form eines Lehr-/Lernsystems. Duisburg: Diplomarbeit (1997).

Lüssem, J., Sünderkamp, St.: Einsatz von multimedialen Lernsystemen im Informatikunterricht. In Hoppe, H.U., Luther, W. (Hrsg.) 1997, 188-196.

Naps, T.L.: Algorithm Visualization on The World Wide Web - The Difference Java Makes! ITiCSE'97, Uppsala, Sweden. SIGCSE/SIGCUE Proc. New York: ACM Press (1997), 59-61.

Oevel, W., Postel, F., Rüscher, G., Wehmeier, S.: Das MuPAD Tutorium. Paderborn (Germany): SciFace Software (October 1998), ISBN 3-933764-00-9.

Rothenhöfer, D., Herzog, Chr.: Sypros - an intelligent tutoring system for parallel programming. In Chan, Tak-Wai (ed.). Proc. 1993 Int. Conf. on Computers in Education, Taiwan, Dec. 1993, 300-305.

XTANGO, TANGO, POLKA, SAMBA:
http://www.cc.gatech.edu/gvu/softviz/algoanim/.

Ziegler, K.: Ein Demonstrationssystem für verschiedene Morphing Algorithmen unter OpenGL. Duisburg: Diplomarbeit (1998).

Algorithmen lernen mit interaktiven Visualisierungen durch Betrachten, Nachgehen und Experimentieren

Nils Faltin

C. v. Ossietzky Universität Oldenburg
Abt. Computer Graphics & Software-Ergonomie
D – 26111 Oldenburg

Faltin@Informatik.Uni-Oldenburg.DE

Dieser Artikel definiert drei Arten von interaktiven Visualisierungen für das Lernen von Algorithmen: Betrachten (Animation), Nachgehen (Nachmachen der Schritte) und Experimentieren (mit einer Simulation des Algorithmus). Für sie werden aus didaktischer und software-ergonomischer Sicht Gestaltungshinweise gegeben und Beispiele genannt. Schließlich wird eine SW-Architektur skizziert, die es erleichtern soll, die drei Arten von Visualisierung zu entwickeln.

Einführung

Dieser Artikel versucht die folgenden Fragen zu klären:

- Wie kann man mit interaktiven Visualisierungen Algorithmen lernen?
- Welche Rolle kann das Lernmedium Computer dabei spielen?
- Welche neuen Möglichkeiten bieten dabei insbesondere grafische Benutzungsoberflächen?

Als Einstieg soll ein Beispiel aus der Grundschule dienen: Der erste (formale) Algorithmus den ein Kind typischerweise lernt ist das schriftliche Addieren von Dezimalzahlen. Die Lehrkraft zeigt den Schülern schrittweise den Ablauf an einem Beispiel und erklärt Regeln, wie z.B. der Übertrag zu handhaben ist. Die Ziffern (Datenobjekte) werden an der Tafel in einem bestimmten, tabellenartigen Muster angeordnet. Das Tafelbild wird schrittweise weiterentwickelt, der Algorithmus wird also in Einzelschritten dynamisch visualisiert. Die Schüler *betrachten* diese dynamische Visualisierung und lernen so den Algorithmus kennen.

In einer zweiten Phase sollen sie den Algorithmus selbst ausführen, indem sie eine Beispielaufgabe auf Papier rechnen. Dabei ist es hilfreich, wenn das tabellenartige Muster schon in einem Arbeitsblatt vorgegeben ist und dort die zu addierenden Zahlen bereits eingetragen sind. Das Abarbeiten des Algorithmus durch das *Nachgehen* der zuvor gesehenen Schritte hat mehrere didaktische Funktionen. Zunächst dient es der Selbstkontrolle, ob der Algorithmus verstanden wurde. Weiterhin prägt sich der Ablauf ins Gedächtnis ein und kann nach mehreren Durchläufen schließlich automatisch (rou-

tiniert) ausgeführt werden. Eine Aufgabe soll auch zum Lernen motivieren. Sie fordert die intellektuelle Fähigkeit der Schüler heraus und bringt sie von einer aufnehmenden (rezeptiven) Haltung zu einer aktiven Arbeit. Wird die Aufgabe richtig bearbeitet, so kann dies als Erfolg zum weiteren Lernen motivieren. Ein falsches Ergebnis dient als Hinweis, dass der Algorithmus noch nicht verstanden wurde oder noch nicht sicher beherrscht wird.

In diesem Beispiel sieht man zwei Formen der Visualisierung von Algorithmen für das Lernen. Zunächst wird die Veränderung der Datenstruktur visualisiert und der Lerner *betrachtet* dies rezeptiv. In der zweiten Phase führt der Lerner die Schritte des Algorithmus auf einer Visualisierung der Datenstruktur selbst aus. Da er sozusagen den Fußstapfen des zuvor gesehenen Standardalgorithmus nachgeht, soll dies als *Nachgehen* bezeichnet werden.

Die genannten didaktischen Prinzipien lassen sich auch auf das Lernen von Algorithmen in späteren Stufen der Ausbildung, z.B. an der Hochschule anwenden. Dort geht es dann um Algorithmen, die von Computern ausgeführt werden sollen. Hat der Lerner gelernt, welche Schritte der Algorithmus durchführt so dürfte es ihm leichter fallen, den Algorithmus mathematisch zu analysieren und zu implementieren.

Soll der Lerner aber befähigt werden einen Algorithmus um neue Funktionen zu erweitern, so wird eine Lernform benötigt, die die promblemlösenden Fähigkeiten fördert. Die Aufgabe kann dann lauten, den Pseudocode für eine Funktion des Algorithmus selbst zu entwickeln. Eine Funktion besteht aus Aufrufen von anderen, bereits vorhandenen Funktionen. Die Effekte einer selbst gewählten Aufrufkette kann der Lerner durch *Experimentieren* mit einer interaktiven visuellen Simulation des Algorithmus erforschen.

Die Lernformen *Betrachten*, *Nachgehen* und *Experimentieren* dienen dem Erlernen von Algorithmen mit interaktiven Visualisierungen. In den folgenden Abschnitten werden die Lernformen, insbesondere unter didaktischem und software-ergonomischem Blickpunkt, beleuchtet. Schließlich wird eine Softwarearchitektur vorgestellt, die es erleichtern soll, interaktive Visualisierungen für diese Lernformen zu entwickeln.

Lernform „Betrachten“

Bei der Lernform „Betrachten“ ist der Ablauf des Algorithmus fest vorgegeben. Vom Lerner wird im wesentlichen erwartet, dass er die Schritte des Algorithmus in einer Visualisierung (Algorithmen-Animation) betrachtet. Es wird nicht erwartet, dass er Schritte voraussagt oder selbst Teile des Algorithmus entwickelt. Von den drei Lernformen ist „Betrachten“ die mit Abstand verbreitetste. Einen guten Überblick über Forschungsarbeiten zu Algorithmen-Animation gibt das Werk von Stasko et al [1].

Gestaltungshinweise

Aus didaktischer und software-ergonomischer Sicht möchte ich nun einige Gestaltungshinweise für Animationen geben, die für die Lernform „Betrachten" entwickelt werden. Peter Gloor hat eine ähnliche Liste mit Schwerpunkt auf software-ergonomischen Kriterien entwickelt [2]. Die Animation sollte möglichst viele der Kriterien erfüllen.

Funktionale Struktur. Der Algorithmus wird in Funktionen aufgeteilt. Jede nichttriviale Funktion wird in einer eigenen Animation erklärt. Bereits gelernte Funktionen können in einer Animation als Bausteine (Funktionsaufrufe) verwendet werden und werden nur noch als ein Schritt angezeigt.

Lehrtext. Mit der Animation ist ein Lehrtext verbunden, der in der Art eines Lehrbuchs mit Text und Bildern den Algorithmus erklärt. Alternativ kann dies auch durch einen vorherigen Vortrag vermittelt werden.

Anzeige des Pseudocode. Zusätzlich zu den Datenstrukturen wird der Pseudocode des Algorithmus angezeigt. Die aktuelle Befehlszeile wird hervorgehoben. So kann der Lerner den Pseudocode und die Aktionen auf den Datenstrukturen miteinander in Beziehung setzen.

Erklärung der Schritte. Die Schritte des Algorithmus werden textuell oder per Sprachausgabe kommentiert, damit der Lerner versteht, was der Algorithmus gerade tut.

Interessante Eingabedaten. Viele Algorithmen zeigen bei bestimmten Eingabedaten ein besonderes Verhalten. Der Sortieralgorithmus Quicksort zeigt z. B. bei vorsortierten Eingabedaten ein anderes Laufzeitverhalten als bei zufällig gemischten Eingabedaten. Dem Lerner werden die verschiedenen Sätze von Eingabedaten zur Auswahl gestellt werden.

Kleine Datenmenge. Die Animation arbeitet auf einer kleinen Zahl von Datenobjekten. Die Anzahl ist gerade so gewählt, dass das wesentliche am Algorithmus noch zu erkennen ist. Große Datenmengen erschweren die Wahrnehmung und das Verständnis. Dabei bringen sie im Normalfall keine zusätzlichen Erkenntnisse. Kognitive Psychologen haben in Tests ermittelt, dass Menschen typischerweise 7-9 Einheiten („chunks") im Arbeitsgedächtnis halten können. Deshalb sollten nicht mehr als 7 Objekte an einem Algorithmusschritt aktiv beteiligt sein.

Wenige Schritte. Analog werden nur soviele Wiederholungen von Schleifendurchläufen gezeigt, wie für Wahrnehmung und Verständnis nötig sind. Ansonsten wird der Lerner schnell gelangweilt und seine Aufmerksamkeitskapazität unnötig verbraucht.

Flexible Ablaufsteuerung. Die Animation des Algorithmus kann ähnlich wie bei einem Videorecorder gesteuert werden: Einzelschritte vor und zurück, Abspielen, Pause, Stopp, Sprung an Anfang und Ende. Aus didaktischer

Sicht ist es besonders wichtig in Einzelschritten vor und zurück schalten zu können. Gerade Anfänger, die den Algorithmus neu lernen sind mit einem durchlaufenden Film meist überfordert. Da normale Programme nicht „rückwärts" laufen können stellt dies besondere Anforderungen an die Software-Architektur der interaktiven Visualisierung. Auf eine mögliche Lösung wird im letzten Abschnitt eingegangen.

Interessante Ereignisse. Ein Schritt umfasst dabei ein „interessantes Ereignis" des Algorithmus. Das kann z. B. eine Aktion des Algorithmus oder das Erreichen wichtiger Zwischenergebnisse sein. Im Gegensatz zu diesem Prinzip machen manche SW-Visualisierungs-Systeme grafische Operationen zu Schritten. Das Konzept der „interesting events" wurde von Marc Brown entwickelt [8].

Fokussierung der Aufmerksamkeit. Der Blick des Lerners wird auf die Teile der Darstellung gelenkt, die sich als nächstes verändern werden. Dazu kann der Autor diese Objekte z.B. blinken oder ihre Größe ändern lassen. So wird verhindert, dass der Betrachter wichtige Schritte nur als diffuse Bewegung wahrnimmt.

Weiche Übergänge. Veränderungen auf den Datenstrukturen werden möglichst mit weichen, fließenden Bewegungen gezeigt. Werden z.B. zwei Elemente eines Array vertauscht, so können sie sich jeweils auf einem Halbkreisbogen bewegen und so „Plätze tauschen". Dies erleichtert die Wahrnehmung des Vorgangs. Damit beim Lerner nicht der Eindruck entsteht, in einem Computer würden sich die Daten „fließend bewegen" sollte im Begleittext darauf hingewiesen werden, wie die Veränderung der Daten implementiert wird. Vertauschung wird z. B. über „plötzliche" Kopiervorgänge im Dreieckstausch realisiert. Es war eines der wesentlichen Ziele bei der Entwicklung des SW-Visualisierungssystems Tango, Animationen mit weichen Übergängen zu ermöglichen [7].

Beispiel

Mir ist keine Algorithmen-Animation bekannt, die alle genannten Prinzipien umsetzt. Die Autoren der folgenden Animation des Bubblesort-Algorithmus (Abb. 1) wollten eine ästhetische Animation entwickeln, die die Metapher „Luftblasen" (Bubbles) aufgreift [3]. Sie haben dabei die folgenden Prinzipien umgesetzt:

- Weiche Übergänge
- Flexible Ablaufsteuerung
- Anzeige des Pseudocodes
- Kleine Datenmenge
- Lehrtext

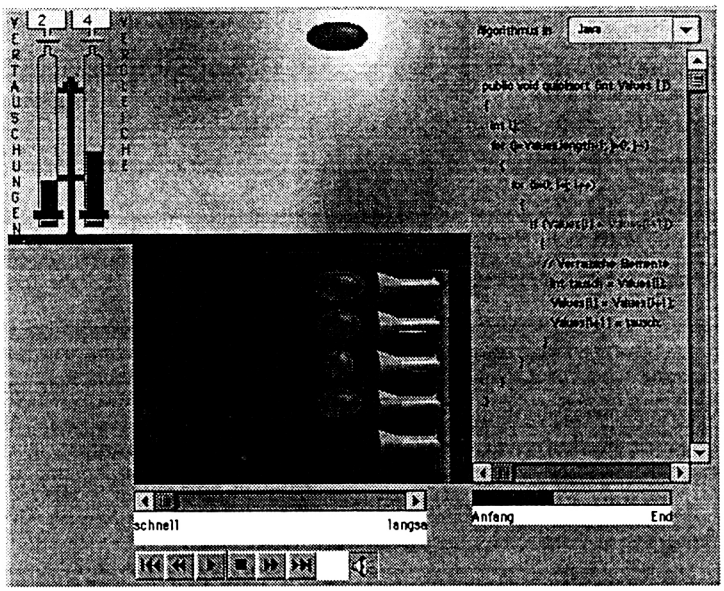


Abb. 1 - Lernen durch Betrachten: Bubblesort-Animation

Lernform „Nachgehen“

Nachdem ein Algorithmus durch „Betrachten“ gelernt worden ist, sollte das Wissen durch Übung überprüft und vertieft werden. Bei der Lernform „Nachgehen“ soll der Lerner mehrfach den nächsten Schritt des Algorithmus voraussagen. Es gibt dabei nur einen „richtigen“ nächsten Schritt, und zwar den, den der Standardalgorithmus als nächstes ausführen würde. Dazu interagiert der Lerner mit einer Visualisierung der Datenstruktur über eine grafische Benutzungsoberfläche. Es scheint nur wenige Projekte zu geben die versucht haben, interaktive Visualisierungen zu entwickeln, die diese Lernform unterstützen [5], [4].

Gestaltungshinweise

Die Gestaltungshinweise für Animationen, die ich für das Lernen durch „Betrachten" gegeben habe, gelten bis auf den Punkt „Anzeige des Pseudocode" auch für das Lernen durch „Nachgehen". Zusätzlich sollten noch die folgenden Gestaltungshinweise beachtet werden.

Kein Pseudocode. Da der Lerner den nächsten Schritt eigenständig voraussagen soll, wird kein Pseudocode angezeigt. Insbesonder der Charakter der Überprüfung des Wissens würde sonst unterlaufen.

GUI nutzen. Die Möglichkeiten einer grafischen Benutzungsoberfläche (GUI) werden soweit möglich und sinnvoll für die Interaktion genutzt. Insbesondere kann der Lerner Datenobjekte in der Grafik der Visualisierung direkt anwählen (z.B. mit der Maus anklicken). Ein Schritt eines Algorithmus besteht oft im Aufruf einer Funktion. Die Funktionen können z.B. durch Schaltflächen (Buttons) dargestellt werden. Der Lerner ruft eine Funktion auf, indem er erst die Objekte (aktuelle Parameter) in der Grafik anklickt und dann die Schaltfläche der Funktion.

Bedienung vereinfachen. Die Bedienung der Aufgabe ist möglichst einfach gestaltet. Es ist einfach erkennbar, welche Funktionen angeboten sind und welche Objekte anwählbar sind. Der Lerner muss also Objekte und Funktionen nur auswählen. Bedienungsfehler sollten als solche gemeldet werden. Die Bedienung sollte beschrieben sein (z.B. als Online-Hilfe oder im Begleittext).

Lösung erschweren. Es werden z. B. mehr Funktionen angeboten, als tatsächlich benötigt werden. Auch sind möglichst alle Objekte der Visualisierung anwählbar. Dadurch wird es erschwert, den nächsten Schritt zu erraten.

Fehlermeldungen und gestufte Hilfe. Wählt der Lerner eine falsche Funktion oder falsche Parameter, so kann es sinnvoll sein zu begründen, warum dieser Funktionsaufruf jetzt keinen Sinn macht. Meist wird es aber reichen zu melden, dass der Algorithmus einen anderen Schritt ausführen würde. Es sollte aber eine gestufte Hilfe abrufbar sein, die einen Hinweis gibt, was zu tun ist und als nächste Stufe den Schritt direkt angibt.

Beispiel

Ein Beispiel für die Lernform „Nachgehen" ist das folgende Applet aus der Lernsoftware zum Binomial Heap Algorithmus (Abb. 2) [4]. In ihm sollen zwei Listen von Binomialbäumen zu einer Ergebnisliste verschmolzen werden. Dies wird ähnlich dem schriftlichen Addieren durchgeführt. Der Lerner muss genau die Schritte des Standardalgorithmus einhalten, kann aber über die Schaltfläche „Tipp" eine einstufige Hilfe abrufen. Die oben genannten Kriterien sind im wesentlichen erfüllt.

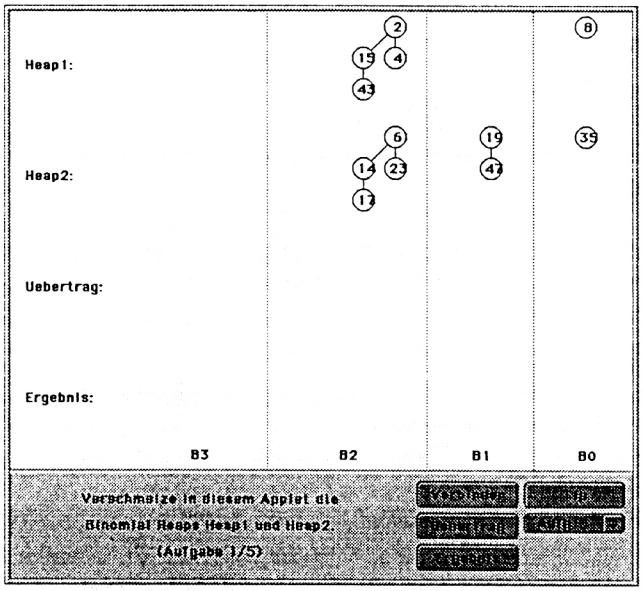


Abb. 2 – Lernen durch Nachgehen: Verschmelzen-Funktion

Lernform „Experiment“

Die *Lernform „Experiment“* soll es dem Lerner ermöglichen den Algorithmus „nachzuentdecken“. Der Algorithmus wird dem Lerner nicht als ein fertiges Werk vermittelt, sondern dieser soll einen Teil der Arbeit der Entwicklung und Entdeckung wiederholen. Er wird dadurch in die Rolle eines Algorithmenentwicklers versetzt.

Die Lernform „Experiment“ muss, wie andere Lernformen auch, in einen größeren didaktischen Kontext eingebettet werden. Für die Gestaltung von Lernsoftware zu Algorithmen habe ich hierzu das didaktisches Konzept „Strukturierte aktive Lernen von Algorithmen (SALA)“ entwickelt [6].

Nach meinem Verständnis besteht ein Algorithmus aus den folgenden Strukturen:

1. Statische und dynamische *Struktur der Daten* (z.B. Listen, Verzeigerung, Records/ Structs)
2. Mathematische und logische Bedingungen zwischen Datenobjekten, den sog. *Invarianten*
3. Funktionale Struktur
4. *Pseudocode* der einzelnen Funktionen

Die *funktionale Struktur* eines Algorithmus gibt an, in welche Funktionen der Algorithmus aufgeteilt ist und welche Funktionen einander aufrufen. Für jede Funktion muss angegeben sein:

- *Zweck* der Funktion
- *formale Parameter* (Typ und Rolle)
- *Start-Invarianten*, die bei Aufruf der Funktion gelten müssen, damit die Funktion arbeiten kann.
- *Arbeits-Invarianten*, die während der Arbeit der Funktion beachtet werden müssen. Teils drückt sich darin die „Strategie" aus, nach der die Funktion arbeitet.
- *Ziel-Invarianten*, die beim Ende der Funktion gelten müssen. Diese leiten sich aus dem Zweck der Funktion ab und aus den Voraussetzungen (zeitlich) nachfolgender Funktionen.

Man kann nicht erwarten, dass ein Lerner den Algorithmus selbst entwickelt (nachentdeckt), wenn er nicht zuvor wesentliche Teile vermittelt bekommt. Ich schlage vor, die Punkte 1 bis 3 (Struktur der Daten, Invarianten und funktionale Struktur) zu vermitteln und den Lerner den Pseudocode selbst entwickeln zu lassen.

Der Pseudocode bestimmt, welche Schritte der Algorithmus auf einem konkreten Satz Daten ausführt. Umgekehrt kann sich der Lerner durch Ausprobieren von Schritten auf Daten zum allgemein funktionierenden Pseudocode vortasten. Dabei ist zu vermuten, dass ihm ein zielloses Probieren wenig weiterhilft, sondern dass er sich eine Strategie überlegen muss, die er durch Experimentieren erproben und verfeinern kann.

Verglichen mit dem Lernen durch „Nachgehen" arbeitet der Lerner beim „Experimentieren" mit einer echten Simulation, bei der aus einer Situation heraus in der Regel alternative Schritte möglich sind. Es kann durchaus mehrere Wege zum Ziel geben. Der Fortschritt der Bearbeitung sollte sich in der Visualisierung zeigen, so dass der Lerner sieht wo „die Arbeit erledigt ist" und wie nah er dem Ziel gekommen ist.

Die Simulation ermöglicht dem Lerner zu experimentieren. Dazu stellt sie mehrere Funktionen zur Verfügung, die auf den Datenobjekten angewendet werden können. Sie führt aber einen Funktionsaufruf nur dann aus, wenn die Arbeits- und Start-Invarianten dies erlauben.

Gestaltungshinweise

Die Benutzungsoberfläche ähnelt der Lernform „Nachgehen". Zusätzlich zu den bei „Nachgehen" genannten Prinzipien sollten die folgenden Gestaltungshinweise beachtet werden:

Inhaltsbezogene Fehlermeldungen. Versucht der Lerner eine Funktion aufzurufen, deren Start-Invarianten nicht erfüllt sind oder würde dadurch eine Arbeits-Invariante verletzt, so muss die interaktive Visualisierung dies mit einer Fehlermeldung abweisen. Die Fehlermeldung sollte den Grund in Bezug auf die Invarianten angeben.

Fortschrittsanzeiger. Der Lerner kann erkennen, wie nah er dem Ziel gekommen ist. Die Visualisierung kann dazu z.B. anzeigen, welcher Anteil der Daten die Ziel-Invariante bereits erfüllt.

Undo. Es kann durchaus sein, dass der Lerner Schritte tätigt, die zwar die Invarianten nicht verletzen, aber nicht zum Ziel führen. Über einen Undo-Mechanismus kann der Lerner einzelne Schritte zurücknehmen.

Beispiel

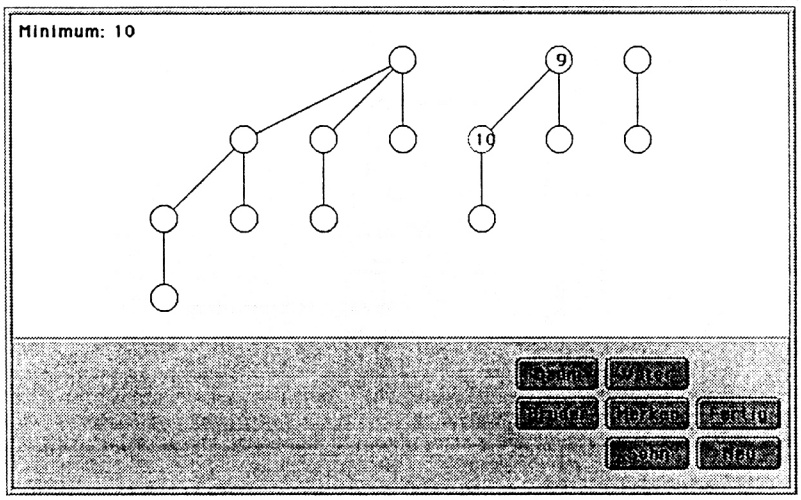


Abb. 3 Lernen durch Experimentieren: „Minimum Suchen"-Funktion

Abbildung 3 zeigt einen Screenshot einer Simulation der „Minimum suchen"-Funktion aus dem Binomial-Heap Algorithmus [4]. Der Lerner kann frei von Baum zu Baum und von Knoten zu Knoten navigieren um das Minimum zu suchen. Sobald ein Knoten besucht wird, wird sein Wert sichtbar. Aufgrund der Invarianten reicht es die Wurzeln abzusuchen und sich den kleinsten Wert zu merken. Diese Simulation verzichtet weitgehend auf Fehlermeldungen, Fortschrittsanzeiger und Undo. Stattdessen wird am Ende gesagt, ob das Minimum gefunden wurde und ob dies der effizienteste Weg war.

SW-Architektur für interaktive Visualisierungen

In unserer Arbeitsgruppe werden schon seit einigen Jahren Algorithmen-Visualisierungen für Lernzwecke entwickelt. Während die Visualisierungen zunächst das Lernen durch „Betrachten" unterstützten ([9],[3]), haben wir im Projekt „Medienunterstütztes

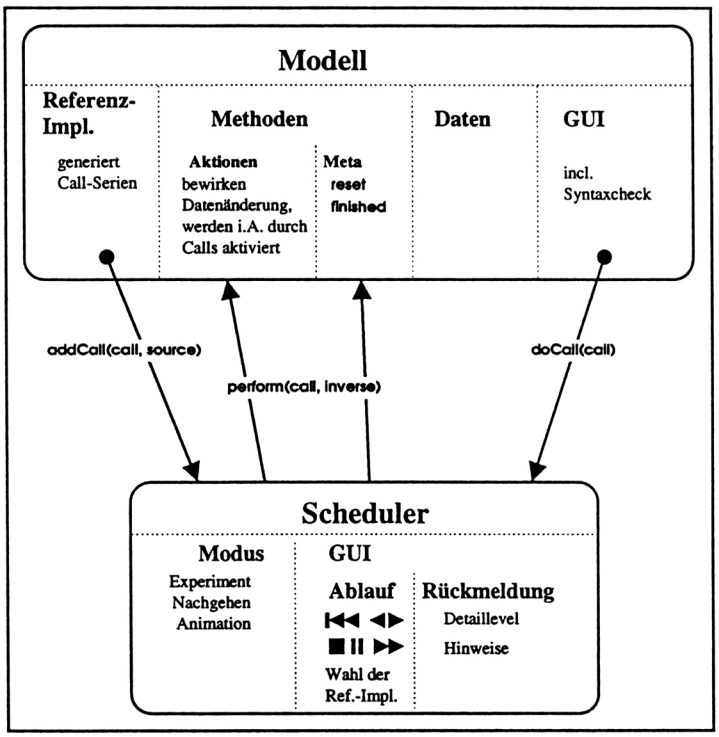


Abb. 4 – SW-Architektur

Studium der Informatik" ([10]) das Lernen durch „Experimentieren" untersucht und Algorithmen-Simulatoren dafür entwickelt. Jede der Lernformen „Betrachten", „Nachgehen" und „Experimentieren" hat einen didaktisch sinnvollen Einsatzbereich. Es kann daher sein, dass für einen Algorithmus alle drei Arten der interaktiven Visualisierung entwickelt werden sollen.

Daher arbeiten wir zur Zeit an einer SW-Architektur, die dem Autor erleichtern soll die drei Arten der Visualisierung zu erstellen. Die Teile, die unabhängig vom Algorithmus sind, sollen dem Autor als Modul bereitgestellt werden. Außerdem soll eine programmtechnische Beschreibung des Algorithmus und einer „allgemeinen" Visualisierung die automatische Erzeugung der drei Arten der Visualisierung erlauben.

Ein erster Entwurf soll hier skizziert werden. Er wurde in Java prototypisch implementiert und mit einer einfachen Testaufgabe erprobt.

Die Architektur realisiert die interaktive Visualisierung für eine Funktion des Algorithmus (Abb. 4).

Die zentrale Idee ist die Trennung der Visualisierung in ein *Modell* und einen *Scheduler*. Funktionen des Algorithmus werden nicht direkt aufgerufen, sondern die *Aufrufe* (engl. *call*) laufen über die Zwischeninstanz des Schedulers.

Im *Modus Animation* (Lernen durch „Nachgehen") arbeitet die Visualisierung in der folgenden Weise. Bei der Initialisierung generiert die sog. *Referenzimplementierung* der Funktion eine Serie von Aufrufen und leitet diese an den Scheduler, der sie in einer Liste speichert. Der Lerner kann über die Ablaufsteuerung (GUI des Schedulers) die Aufrufe ausführen lassen (*perform*). Der Scheduler reicht die Aufrufe als Objekte an die Aktions-Methoden des Modells. Diese berechnen selbst die nötigen Daten, mit denen ihr eigener Aufruf rückgängig gemacht werden kann, inklusive des inversen Aufruf-Objekts. Das Inverse Aufruf-Objekt wird als Rückgabewert an den Scheduler gereicht, der dies in einer Inverse-Liste speichert. Führt der Lerner einen Schritt rückwärts aus, so wird das entsprechende Inverse Aufruf-Objekt an das Modell geschickt.

Eine Aktionsmethode arbeitet in folgenden Phasen:

1. **Regelprüfung**. Die Start-Invariante wird geprüft. Ist sie verletzt, gibt die Methode eine Fehlermeldung aus und endet.
2. **Invers-Daten bestimmen**. Die Methode bestimmt die Daten, die nötig sind, um den Effekt der Methode umzukehren. Dazu wird auch die dafür nötige Aktionsmethode bestimmt.
3. **Änderungen animieren**. Die Veränderung der Datenstruktur wird, wenn möglich, (animiert) in der Visualisierung dargestellt.
4. **Daten ändern**. Die in der Visualisierung sichtbaren Daten werden auch intern im Modell verändert.
5. **Zustand darstellen**. Wenn nötig wird nun der neue Zustand der Modell-Daten in der Visualisierung dargestellt.

Die Daten werden durch die GUI des Modells visualisiert. Sie nimmt auch die Interaktionen des Benutzers entgegen, mit denen er Funktionen aufruft und prüft auf einfache syntaktische Fehler (z.B. Anzahl und Typ der Parameter).

Im *Modus „Nachgehen"* vergleicht der Scheduler die Aufrufe des Benutzers mit denen der Referenzimplementierung und lässt nur die gleichen Schritte wie die Referenzimplementierung zu.

Im *Modus „Experimentieren"* ignoriert der Scheduler die Aufrufe der Referenzimplementierung und leitet alle Aufrufe des Lerners an die Aktionsmethoden durch. Da auch hier intern inverse Call-Objekte erzeugt werden, kann der Lerner über den Scheduler Schritte zurücknehmen (Undo).

Zusammenfassung und Ausblick

Die Interaktive Visualisierung von Algorithmen sollte kein Selbstzweck sein, sondern im Blick auf den Einsatzzweck gestaltet werden. Sollen interaktive Visualisierungen für das Lernen von Algorithmen verwendet werden, so ist die didaktische Einbettung von großer Bedeutung. Als Teil der didaktischen Einbettung wurden in diesem Artikel die Lernformen „Betrachten", „Nachgehen" und „Experimentieren" vorgestellt. Wäh-

rend es viele Veröffentlichungen zu technischen Fragen rund um die Animation von Algorithmen gibt, werden didaktische Fragestellungen nur selten behandelt. Hier ist sicher noch Forschungsbedarf gegeben.

Neue Lernformen wie „Nachgehen" und „Experimentieren" müssen sich in der Praxis bewähren. Dazu braucht es zunächst noch mehr Lernmaterial (z.B. Lernsoftware), die nach diesen Methoden gestaltet ist. Anschließend ist der Lernerfolg dieser Vermittlungsmethoden zu evaluieren.

Abschließend möchte ich die Leser ermutigen, sich anhand der genannten Lernsoftware ([4], [10])ein eigenes Bild davon zu machen, was die neuen Lernformen leisten können. Die Lernsoftware kann mit normalen Webbrowsern aus dem Internet geladen und ausgeführt werden.

Quellen

[1] John T. Stasko, John B. Domingue, Marc H. Brown und Blaine A. Price (Hrsg.): *"Software Visualization"*. 1998. MIT Press, Cambridge, Massachusets.

[2] Peter Gloor: „User Interface Issues for Algorithm Animation" in: [1], S. 145-152.

[3] A. Barbu, M. Dromowicz, X. Gao, M. Köster und C. Wolf: *Lernsoftware „Sortieren mit Bubblesort und Quicksort"*. 1998. http://www-cg-hci.informatik.uni-oldenburg.de/~musik/lp_musik.html#label19 (letzer Zugriff: 4.3.2000).

[4] Karsten Block: *"Lernsoftware zum Binomial Heap Algorithmus"*. 1999. http://www-cg-hci.informatik.uni-oldenburg.de/~musik/lp_musik.html#label2 (Zuletzt abgerufen 4.3.2000).

[5] Linda Stern, Harald Sondergaard and Lee Naish: *"A Strategy for Managing Content Complexity in Algorithm Animation"*. PP 127-130 in: Bill Manaris (Ed.), Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education - ITiCSE '99. June 1999. ACM Press, New York.

[6] Nils Faltin: *"Gestaltung von Lernprogrammen zu Algorithmen für aktives Lernen mit virtuellen Brettspielen"*. 1999. Tagungsband der Tagung "Informatiktage 1999" der Gesellschaft für Informatik, 12. - 13. November 1999 in Bad Schussenried. Konradin Verlag. Preprint-Version: http://www-cg-hci.informatik.uni-oldenburg.de/~musik/Gest_LP/

[7] John Stasko: *„Smooth Continous Animation for Portraying Algorithms and Processes"*. S. 103-118 in [1].

[8] Marc Brown: *"Algorithm Animation"*. 1987. MIT Press, Cambridge.

[9] Karsten Block, Tammo Freese, Palle Klante et al.: *Lernsoftware "Computergrafik Interaktiv - Grafiti"*. http://www-cg-hci.informatik.uni-oldenburg.de/~musik/lp_musik.html#label1 . 1999. (Zuletzt abgerufen 4.3.2000).

[10] Peter Gorny und Nils Faltin: *"Das Projekt Medienunterstütztes Studium der Informatik (MuSIK)"*. 2000. http://www-cg-hci.informatik.uni-oldenburg.de/~musik/ (Zuletzt abgerufen 4.3.2000).