Informatik Forsch. Entw. (2002) 17: 12–20

Informatik
Forschung und Entwicklung

Generierung interaktiver Animationen von Berechnungsmodellen

Stephan Diehl, Andreas Kerren

Universität des Saarlandes, FR 6.2 Informatik, Postfach 15 11 50, 66041 Saarbrücken (e-mail: {diehl,kerren}@cs.uni-sb.de)

Eingegangen am 18. Juli 2000 / Angenommen am 29. Juni 2001

Zusammenfassung. In diesem Artikel werden zwei generative Ansätze für animierte Berechnungsmodelle vorgestellt und im Kontext von Lernsoftware für den Compilerbau angewandt. Zuerst wird eine Implementierung für den ersten Ansatz beschrieben. Basierend auf der Erfahrung mit diesem Prototypen wurde der zweite Ansatz entwickelt und das GANIMAL Rahmenwerk 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 Programmier- und Animationsbeschreibungssprache GANILA bietet.

Schlüsselwörter: Algorithmenanimation, Softwarevisualisierung, Compiler, Lernsoftware

Abstract. In this article we introduce two new generative approaches of animated computational models. These approaches are applied in context of educational software systems for compiler design. First we describe the implementation of the first approach. Based on the experiences with this prototype implementation we have developed the second approach and the GANIMAL framework was designed. This framework consists of a generic algorithm animation system, which offers a unique set of possibilities because of its graphical base package, its concurrent runtime system with graphical user interface and its programming and animation specification language GANILA.

Keywords: Algorithm animation, Software visualization, Compilers, Educational software

CR Subject Classification: D.2.6, D.3.2, D.3.4, F.1.1, H.5.1, H.5.2, H.5.4, K.3.2

1 Einleitung

Im Software-Engineering gewinnen generative und generische Techniken insbesondere unter dem Aspekt der Wiederverwendbarkeit 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 Bestandteil einer web-basierten Lernsoftware für den Compilerbau werden.

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 der Unentscheidbarkeit 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 welcher der Computer als Korrektor auftritt, entmutigend wirkt und daher wenig erfolgreich ist. Die existierenden Untersuchungen hierzu sind jedoch teilweise widersprüchlich, siehe [24]. Unser Ansatz ermöglicht exploratives, selbstgesteuertes 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 [11, 12].

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.

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

2 Relevante Arbeiten

Einen Überblick zum Thema “Softwarevisualisierung” und insbesondere zu Systemen zur Programm- und Algorithmenanimation geben zwei Sammelbände aus den Jahren 1996 und 1998 [16, 29]. Letzterer enthält auch einige überarbeitete Versionen von klassischen Papieren. Neuere Arbeiten gerade im deutschsprachigen Raum werden im Tagungsband des GI Workshops SV2000 vorgestellt [8], aktuelle Informationen bietet zudem die Webpage `www.softwarevisualisierung.de`. Wir geben hier nur einen knappen Abriß der historischen Entwicklung, wobei wir insbesondere Systeme erwähnen, die neue Konzepte eingeführt haben.

Knowltons Film über die Listenverarbeitung mit der Programmiersprache L6 gehört zu den ersten Versuchen, Programmverhalten mit Hilfe von Animationstechniken zu visualisieren [22]. Der Einsatz in der Lehre stand schon früh im Mittelpunkt [19, 1], und eine Reihe weiterer Filme wurde produziert, insbesondere der Klassiker “Sorting Out Sorting” [33], der 9 verschiedene Sortierverfahren erläutert. Die Erfahrungen mit von Hand programmierten Algorithmenanimationen und die weite Verfügbarkeit graphikfähiger Rechner führten in der Folge zur Entwicklung von Algorithmenanimationssystemen. BALSA [6, 7] führte das Konzept der Interesting Events (IEs) ein und damit die Verwendung mehrere Sichten auf den gleichen Zustand. Das Programm wird an wichtigen Punkten mit IEs annotiert. Wann immer ein IE bei der Ausführung erreicht wird, sendet es Informationen über den aktuellen Programmzustand an die Sichten. In BALSA II [3] kamen Schritt- und Haltepunkte hinzu. In CAT [4] und später JCAT [5] wurden die Sichten auf mehrere Rechner verteilt. TANGO [25] setzte erstmals das Pfad-Übergangs-Paradigma [28] um und erlaubte damit gleichmäßige und parallele Animationen von Zustandsübergängen. Dies wurde in POLKA [27] noch verbessert und mit SAMBA [26] ein Frontend geschaffen, das es erlaubte, Programme in beliebigen Programmiersprachen zu schreiben, welche die IEs als Text ausgeben. Dieser wird dann post mortem von SAMBA eingelesen und die Animation ausgeführt. Post-mortem-Visualisierungen anhand von Protokollen aus Programmläufen (Traces) wurden aber bereits bei PVM [30] für parallele Programme verwendet.

Anstatt Algorithmen zu annotieren, wurde in GASP [31] der Weg beschritten, eine Bibliothek mit geometrischen Datentypen bereitzustellen, wobei die Operationen des Datentyps mit Animationen versehen sind. Damit erhält man dann sozusagen die Algorithmenanimation als Nebeneffekt eines Programms, welches die Bibliothek verwendet.

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 den folgenden Abschnitten zwei Methoden vor. Zunächst betrachten wir eine Implementierung, die auf der ersten Methode beruht, analysieren diese und entwerfen ein Rahmenwerk (Framework), das als Grundlage für die zweite Methode verwendet werden kann.

3 Lernsoftware für den Compilerbau

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 [24] (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 [2].

Tabelle 1. Compilerphasen

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

In der Informatik und insbesondere im Compilerbau sind Theorie und Algorithmen sehr abstrakt und meist auch komplex. Daher ist ihre visuelle Darstellung ein wichtige Hilfe im Informatikunterricht [20, 21]. 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.

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, z.B. durch 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.

Im Compilerbau werden Techniken entwickelt, mit denen Programme höherer Programmiersprachen in effiziente und korrekte Programme einer realen oder abstrakten Maschine übersetzt werden können [32]. 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 Tabelle 1 werden Compilerphasen, ihre Spezifikationssprachen und zugrundeliegende Berechnungsmodelle aufgeführt.

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 desselben Sprachenparadigmas (imperativ, funktional, logisch, objektorientiert) maßgeschneidert. In Abb. 1 ist eine interaktive Animation einer abstrakten Maschine für eine funktionale Sprache zu sehen. An Programmspeicheradresse 22 auf der linken Seite der Abbildung wird die Instruktion

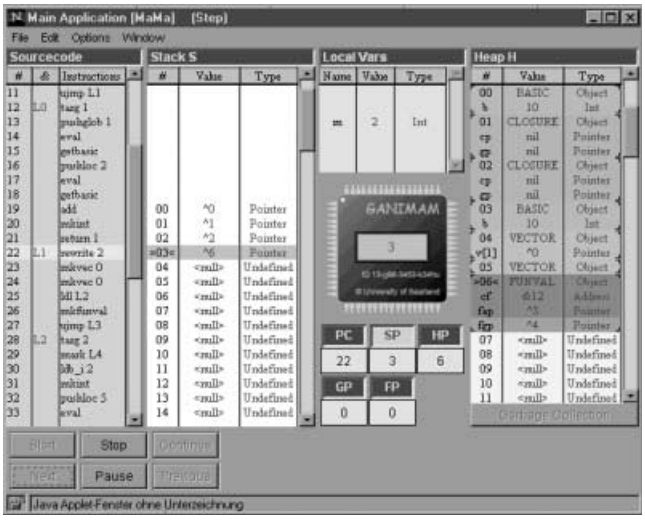


Abb. 1. Momentaufnahme einer animierten abstrakten Maschine

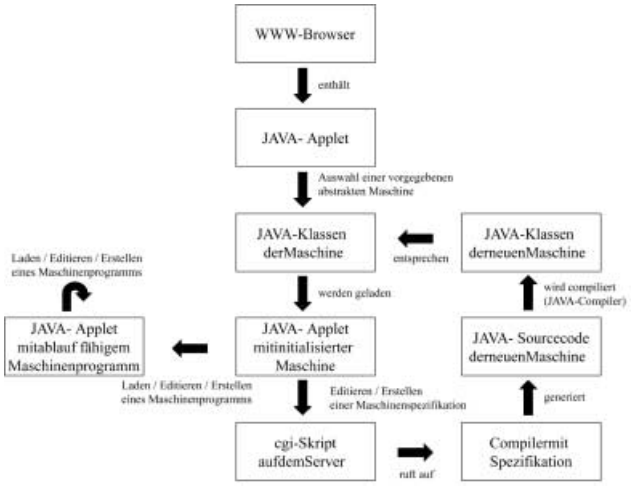


Abb. 2. Interaktion der Systemkomponenten

`rewrite 2` abgearbeitet. In Abhängigkeit von deren Definition, die in einem zusätzlichen Fenster angezeigt wird, kann der Benutzer den jeweils aktuellen Zustand der Maschine und den Prozeß der Instruktionsabarbeitung genau verfolgen. Beispielsweise liegt auf der obersten Kellerzelle an Kelleradresse 3 ein Zeiger mit der Haldenadresse 6, welcher auf das komplexe Haldenobjekt `FUNVAL` mit 3 Komponenten verweist. Eine zusätzliche Beschreibung der Visualisierung von abstrakten Maschinen ist in Abschnitt 4.1 gegeben.

4 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. Der nächste Abschnitt 4.1 stellt eine Implementierung vor, die auf dieser Methode beruht.

4.1 Ein erster Prototyp: GANIMAM

Als Beispiel der ersten Methode betrachten wir die Entwicklung von GANIMAM, unseres web-basierten Generators für interaktive Animationen abstrakter Maschinen [13, 14]. Abbildung 1 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.

Technischer Überblick. GANIMAM kann über die Webseite des GANIMAL-Projekts [18] aufgerufen werden. Nachdem die Klassen des Basispakets des Applets vollständig geladen sind, stehen dem Benutzer mehrere bereits vorgegebene abstrakte Maschinen zur Auswahl. Die auf dem Server gespeicherten Java-Klassen der ausgewählten Maschine werden vom Applet geladen, und der Benutzer erhält eine vorinitialisierte abstrakte Maschinenvisualisierung, dabei sind z.B. die Register auf Defaultwerte eingestellt. Einerseits hat man die Möglichkeit, bereits erstellte Maschinenprogramme für diese abstrakte Maschine zu laden und sich den Programmablauf animiert darstellen zu lassen. Weiterhin lassen sich selbsterstellte Maschinenprogramme eingeben, die vordefinierten Maschinenprogramme verändern, das Layout der verschiedenen Teile der visualisierten abstrakten Maschine ändern und die animierte Ausführung eines Maschinenprogramms steuern. Andererseits hat der Benutzer nun die Option, die bereits geladene Maschine zu ändern, indem er die entsprechende Spezifikation modifiziert oder ganz neu eingibt. Das Applet schickt diese neue Spezifikiation zum Server. Ein CGI-Skript auf dem Server erzeugt Java-Quellcode, der durch einen Java-Compiler in Klassendateien der neuen Maschine übersetzt wird. Nachdem das Applet diese Klassen per Java Reflection geladen hat, kann der Benutzer für diese neue Maschine wieder neue Maschinenprogramme eingeben und deren Ablauf visualisieren oder erneut die abstrakte Maschinenspezifikation ändern und sich auf dem Server eine neue abstrakte Maschine erzeugen lassen usw., siehe Abb. 2.

Abstrakte Maschinen können eine unterschiedliche Anzahl von Registern, Kellern und 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 Abb. 1). 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 Mausklick auf eine Instruktion im Programmcode kann einerseits die Definition der Instruktion in das Akkumulatorfenster geladen werden, andererseits der Wert des Programmzählers auf die Adresse der Instruktion gesetzt werden, 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 dem der Benutzer den Wert und den Typ, bei Registern nur den Wert ändern kann.

Vorteile interaktiver Animationen. GANIMAM bietet verschiedene Arten der Interaktion. Erstens kann der Benutzer Spezifikationen einer abstrakten Maschine eingeben oder ändern [15]. Beim Visualisieren einer abstrakten Maschine ist es wichtig, daß man nicht nur Daten und Vorgänge auf dem geringen Niveau und der Granularität der Spezifikationssprache darstellt, sondern auch Abstraktionen auf höherem Niveau. Um dies zu ermöglichen, wurden Visualisierungsannotationen zur Spezifikationssprache hinzugefügt. Diese Annotationen ähneln den Interesting Events einschlägiger Algorithmenanimationssysteme, siehe Abschnitt 2.

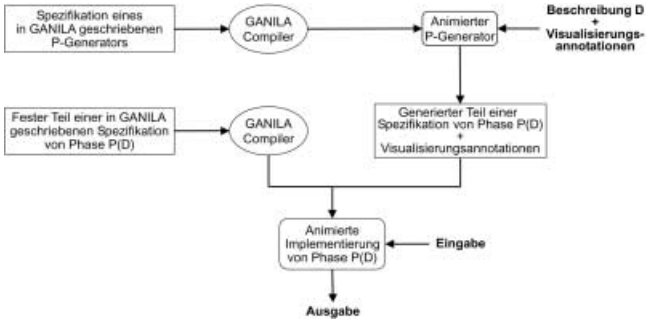
Nach der Erzeugung der Implementierung der abstrakten Maschine kann der Benutzer Programme in der Sprache der abstrakten Maschine eingeben, diese Schritt für Schritt ausführen und den Inhalt jedes Registers und jeder Speicherzelle inspizieren. Während der Ausführung einer Instruktion zeigt eine Animation 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 [32].
- 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 Methode II: Implementiere Generatoren in einer Programmier- und Animationsbeschreibungssprache

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



Als Beispiel betrachten wir einen lexikalischen Analysatorgenerator. P ist die lexikalische Analyse und D ein 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.

Abb. 3. Generierung animierter Generatoren und Compilerphasen

mit einem festen Treiber die Implementierung einer Compilerphase ergeben. Zuerst wurde eine Programmier- und Animationsbeschreibungssprache GANILA (siehe Abschnitt 5.3) 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 Abb. 3. Zusätzlich zur Annotierung der Spezifikation der Compilerphase, wie in der ersten Methode beschrieben, annotieren wir die Spezifikation des Generators und die des Treibers durch Markieren von Programmpunkten mit 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 verschiedene Events reagiert. Auf dieser Methode beruhen die interaktiven, animierten Visualisierungen im elektronischen Textbuch zur Theorie der Generierung endlicher Automaten [17].

5.1 Das GANIMAL-Rahmenwerk

Basierend auf den mit GANIMAM und verwandten Arbeiten zur Softwarevisualisierung [8, 26] gesammelten Erfahrungen haben wir das GANIMAL-Rahmenwerk für die Erstellung generativer Lernsoftware entwickelt, das in Abb. 4 schematisch dargestellt ist. Es besteht aus dem GANILA-Compiler und einem Laufzeitsystem, für das der Compiler Java-Code generiert. In diesem Rahmenwerk wird aus einer in GANILA geschriebenen Spezifikation ein Algorithmenmodul automatisch erzeugt. Während der Ausführung des Algorithmus sendet dieses Modul ein Interesting Event `IE`, den Programmpunkt `pp`, an dem das Event aufgetreten ist, sowie den aktuellen Animationsmodus (`RECORD`, `PLAY`, `REPLAY`) zu einem Kontrollobjekt, welches in seinen Einstellungen überprüft, ob für diesen speziellen Programmpunkt das Interesting Event zu allen Sichten weitergeleitet werden muß. In diesem Fall wird ein Broadcast des Events `IE` zu allen Sichten initiiert. Jede Sicht 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

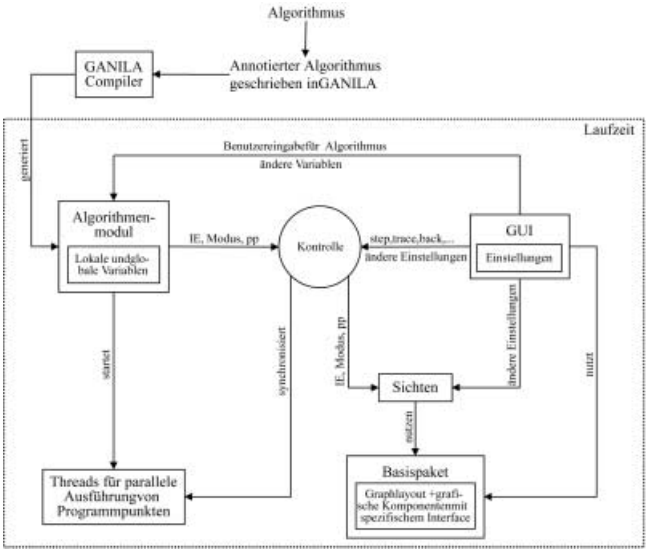


Abb. 4. Das GANIMAL-Rahmenwerk

Abb. 5. Algorithmenanimation von Heapsort

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 Sichten sollten das objektorientierte Basispaket verwenden, das graphische 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.

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-Rahmenwerk, auch klassische Algorithmenanimationen in einfacher Weise zu implementieren, etwa die Animation des Heapsortalgorithmus, siehe Abb. 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.

5.2 Vordefinierte Standardsichten

GANILA und das Basispaket stellen eine Menge von vordefinierten Sichten (Views) zur Verfügung, die durch das GANILA-Sprachkonstrukt `view <Name>(Parameter)` einfach in das Programm eingebunden werden:

```
// Eine View ohne Parameter
view CodeView();
// Eine View mit Parameter
view Aura("http://www.cs.uni-sb.de/sounds/");
```

Der Entwickler der Animation (Animator genannt) kann diese Sichten unverändert übernehmen oder durch Vererbung deren Implementierung erweitern bzw. verändern. Dabei sind die Methoden der Sichten die Eventhandler der Interesting 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 in das Rahmenwerk eingebetteten HTML-Browser (IceBrowser-JavaBean) angezeigt werden. Andererseits können Informationen, die zur Laufzeit an diesen Programmpunkten zur Verfügung stehen, durch Parameterübergabe an CGI-Skripten zu einem Server transferiert werden. In “Literate Programming” [23] wird ein verbundenes statisches Dokument durch die Dokumentationen der Programmpunkte erzeugt. Im Gegensatz dazu kann in GANILA die Dokumentation eines Programmpunktes angezeigt werden, wann immer dieser Programmpunkt während der Animation erreicht wird.

GraphView. Die GraphView bietet mehrere Graphlayoutalgorithmen, siehe auch Abschnitt 4.1. Knoten und Kanten können beliebig hinzugefügt bzw. gelöscht werden. Hierbei können die Knoten (fast) beliebige Java-Swing-Komponenten sein.

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

Aura. Diese Sicht verbindet in analoger Weise zur HTML-View Programmpunkte mit Sounddateien. Sie erhält über ein Interesting Event die URL einer Sounddatei und spielt diese ab. Start, Stop, Schleifen, Lautstärkeregelung etc. werden ebenfalls über das Senden von Interesting Events gesteuert.

5.3 Design der Programmier- und Animationsbeschreibungssprache 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 Sichten 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 Sicht einzeln definiert werden, falls keine Parallelausführung eingestellt ist.

5.3.1 Interesting Events. Das nachfolgende Programmfragment zeigt den GANILA-Code für eine einfache Operation, die oft exemplarisch in Animationen der bekannten Algorithmenanimationssysteme enhalten ist: das Vertauschen der Inhalte zweier Feldelemente, hier `a[i]` und `a[j]`.

```
*IE_MoveToTemporary(i);
help = a[i];
*IE_MoveElement(i,j);
a[i] = a[j];
*IE_MoveFromTemporary(j);
a[j] = help;
```

Interesting Events haben das Präfix `*IE_` und übertragen lokale Informationen über ihre Argumente zu den einzelnen Sichten, die diese Events bearbeiten. Es ist leicht zu sehen, daß eine Implementierung einer Sicht in unserem Falle den Wert in `a[i]` zu einer Darstellung der Hilfsvariablen bewegen könnte. Daraufhin würde sich der entsprechende Wert von `a[j]` nach `a[i]` und schließlich der Wert der Hilfsvariablen nach `a[j]` bewegen. Im Vergleich zu anderen eventbasierten Algorithmenanimationssystemen wie Zeus oder Polka entsprechen GANILA-Events (GEvents) bisher genau der Interesting Events. Der Unterschied ist, daß ein GEvent zuerst zur zentralen Kontrolle des Rahmenwerks gesendet wird und dabei einem gewöhnlichen Programmpunkt entspricht. Daher hat der Benutzer nun zur Laufzeit der Animation die Wahl, ob er den Effekt eines Events in einer Sicht beobachten möchte oder nicht, indem er das Event in der GUI deaktiviert. Der Eventhandler einer Sicht muß nun so implementiert sein, daß er deaktivierte Events trotzdem erhält, um so eventuelle Änderungen am internen Zustand der Sicht vorzunehmen, daß Inkonsistenzen verhindert werden. Jede Sicht registriert sich bei der Kontrolle und diese leitet alle Interesting Events zu den registrierten Sichten weiter. Ein Eventhandler einer Sicht kann nun wiederum neue Sichten kreieren und muß diese jeweils bei der Kontrolle registrieren.

5.3.2 Alternative Interesting Events. In der Sprache GANILA ist es möglich, Programmpunkte durch die Klammerung `*{ ... *}` zu gruppieren. Das Statement `*Alt {<Eventliste>}` löst ein oder mehrere alternative GEvents für einen Programmpunkt bzw. eine Programmpunktgruppe aus. Das folgende Programmbeispiel zeigt die Anwendung dieses Statements:

```
*{ *IE_MoveToTemporary(i);
   help = a[i];
   *IE_MoveElement(i,j);
   a[i] = a[j];
   *IE_MoveFromTemporary(j);
   a[j] = help; *}
*Alt{*IE_Swap(i,j)}
```

In der alternativen Darstellung könnten sich die beiden spezifizierten Werte des Feldes beispielsweise gleichzeitig zu ihren neuen Positionen innerhalb des Feldes bewegen. Zu beachten ist aber, daß es sich hierbei um eine nebenläufige Implementierung des entsprechendas Eventhandlers der Sicht handelt und nicht um eine Anwendung des im nächsten Abschnitt 4.1 beschriebenen Paralleloperators. Der Benutzer kann hier ebenfalls zur Laufzeit entscheiden, welche Realisierung er beobachten möchte.

5.3.3 Parallele Ausführung. Programmpunkte bzw. Gruppen von Programmpunkten können durch den Operator `*||` parallel ausgeführt werden.

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

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

Im oben dargestellten Programm 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 verwendet hätten, Datenabhängigkeiten eine parallele Ausführung nicht erlauben würden. Das heißt in obigen Fall, daß der Algorithmus hier leicht umgeschrieben werden muß, damit die gewünschte Animation möglich ist.

5.3.4 Test von Invarianten. Nicht selten ist man an Eigenschaften von Algorithmen interessiert, die für alle Programmzustände oder eine Teilmenge davon erfüllt sein sollen. Der Entwickler einer Animation hat die Möglichkeit, eine Hypothese vorzugeben und diese an einer Menge von Programmpunkte bzgl. ihrer Gültigkeit zu testen. Im nachfolgenden Programmbeispiel wird die sogenannte Heapeigenschaft an einem Teilcode des Heapsortalgorithmus überprüft, welcher die Werte eines Feldes `a[]` sortiert:

```
*IV(a[i]>=a[2*i+1] && a[i]>=a[2*i+2])
*{
   // Teil des Codes eines
   // Heapsortalgorithmus
*}
```

Handelt es sich bei der Hypothese um eine Invariante, so erkennt der Benutzer des Systems zur Laufzeit des Programms, an welchen Programmpunkten die Invariante verletzt und ab welchem Punkt wiederhergestellt wird. Andererseits kann der Benutzer zur Laufzeit selbst Hypothesen

formulieren und austesten. Dabei ist es manchmal notwendig, komplexere Funktionen aufzurufen, die vom Animationsentwickler im zugrundeliegenden GANILA-Programm spezifiziert werden müssen. Da es nicht sinnvoll ist, jede Methode des Programms von außen aufrufbar zu halten, muß der Entwickler derartige Methoden mit dem Modifikator `interactive` versehen:

```
interactive boolean heapProperty(a,i) {
   // Testet Heapeigenschaft und
   // liefert "true" bzw. "false"
}
```

5.3.5 Visualisierungskontrolle für die Steuerung von Schleifen und Rekursionen. Oftmals finden interessante Ereignisse innerhalb von Schleifen oder rekursiven Funktionsaufrufen statt. Man betrachte z.B. das Durchlaufen einer Liste bzw. rekursive Sortierverfahren wie Quicksort usw. Ist nun ein solcher Schleifendurchlauf ein Bestandteil eines größeren Algorithmus, dann ist es gegebenenfalls nicht wünschenswert, daß alle Schleifendurchläufe auch visualisiert werden. Für den Benutzer könnte es schnell langweilig werden, beispielsweise 100 Durchläufe zu sehen, wenn es für das Verständnis ausreichend ist, nur die letzten fünf Schleifendurchläufe zu verfolgen. Um dieses Ziel zu erreichen, ist es in GANILA möglich, Schleifenkonstrukte (`do`, `while`, `for`) selbst mit Visualisierungsbedingungen zu annotieren. Diese werden in eckigen Klammern hinter die originäre Schleifenbedingung eingefügt:

```
for(int i=0;i<100;i++) [$i>=$n-5] {
   foo(i);
}
```

Das Programmbeispiel hätte zur Folge, daß lediglich die letzten fünf Aufrufe der Funktion `foo(i)` animiert würden. Dabei repräsentieren die Schlüsselwörter `$i` den aktuellen Durchlauf und `$n` die maximale Anzahl der Durchläufe. Beide Werte stehen normalerweise erst zur Laufzeit des Programms fest und müssen jeweils neu berechnet werden. Beachtenswert sind hierbei mögliche Vorkommen von Schleifenaustritten mittels des bekannten Sprachkonstrukts `break` im Rumpf der Schleife, so daß in diesem Fall eventuell kein Durchlauf der Schleife visualisiert wird.

Analog ist dieses Vorgehen auch für rekursive Funktionsaufrufe vorgesehen, wobei hier `$i` die aktuelle Rekursionstiefe und `$n` die maximale Tiefe beschreiben.

5.3.6 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 [9, 10].

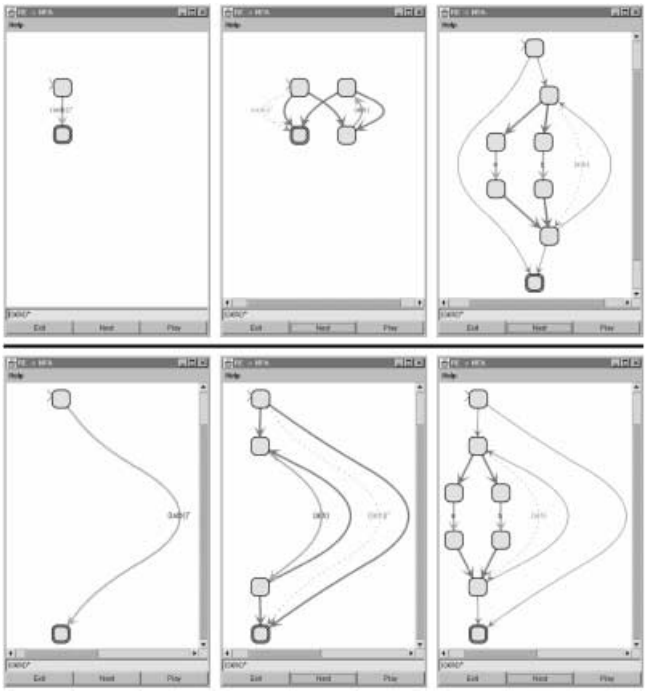


Abb. 6. Ad-hoc und vorausschauendes Graphenlayout für endliche Automaten

Abbildung 6 zeigt, wie dieser Mechanismus verwendet werden kann, um die Umwandlung eines regulären Ausdrucks in einen nichtdeterministischen Automaten (RA→NEA) zu animieren.

```
.
.
.
*RECORD;
  // Code fuer die Umwandlung eines
  // regulaeren Ausdrucks in einen NEA
*REPLAY;
*PLAY;
.
.
.
```

Hier 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 Events werden 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.

Unabhängig von der Anwendung im Graphlayout ermöglicht der RECORD/REPLAY-Mechanismus die Vermischung von post mortem und life/online Algorithmenanimation. Diese Eigenschaft weisen die in Abschnitt 2 beschriebenen Animationssysteme nicht auf.

5.3.7 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 die Rückwärtsausführung eines Algorithmus oder die 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.

6 Abschlußbemerkungen

Eine prototypische Implementierung des Compilers sowie Beispielvisualisierungen, die durch den Compiler erzeugt werden (z.B. Heapsort und Berechnungsmodell der endlichen Automaten), sind verfügbar. Mehr Informationen über das GANIMAL-Projekt und weitere Beispiele sind unter [18] zu finden.

GANIMAL vereinigt Konzepte aus verschiedenen Systemen: Interesting Events und Views (BALSA), schrittweise Ausführung und Breakpoints (BALSA-II), parallele Ausführung (TANGO). Darüber hinaus bietet es aber auch neue Möglichkeiten, wie etwa die alternativen Interesting Events, vorausschauendes Layout bei RECORD/REPLAY, Vermischung von post mortem und life/online Algorithmenanimation, Visualisierungssteuerung der Besuchsfolge von Schleifen und rekursiven Funktionen und Testen von Invarianten während der Ausführung von Blöcken.

Das GANIMAL-Rahmenwerk 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, weitere Inhalte für die Lernsoftware als solche zu kreieren. 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 die Modellierung ökonomischer Prozesse eingesetzt werden.

Letztes Jahr haben wir in Experimenten mit über 100 Studenten jeweils eine Lernsoftware über endliche Automaten ohne und mit generativem Teil evaluiert. Die Ergebnisse zeigen, daß beide Lernsysteme im Vergleich zu einer konventionellen Vorlesung bzw. einem Lehrbuch mit diesen gleich gut abschneiden. Darüber hinaus empfanden die Versuchspersonen eine wesentliche Motivationssteigerung durch die generativen Möglichkeiten.

Literatur

1. R. Baecker. Towards Animating Computer Programs: A First Progress Report. In: *Proceedings of the Third NRC Man-Computer Communications Conference*, 1973
2. B. Braune, S. Diehl, A. Kerren, R. Wilhelm. Animation of the Generation and Computation of Finite Automata for Learning Software. In: *Proceedings of Workshop on Implementing Automata*, Potsdam, 1999
3. M. Brown. Exploring Algorithms with Balsa-II. *Computer*, 21(5), 14–36, 1988
4. M. Brown, M. Najork. Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom. In: *Proceedings of the 1996 IEEE International Symposium on Visual Languages*, Boulder, CO, 1996
5. M. Brown, R. Raisamo. JCAT: Collaborative Active Textbooks Using Java. In: *Proceedings of CompuGraphics'96*, Paris, France, 1996
6. M. Brown, R. Sedgewick. A system for Algorithm Animation. In: *Proceedings of ACM SIGGRAPH'84*, Minneapolis, MN, 1984
7. M.H. Brown. *Algorithm Animation*. MIT Press, 1987
8. S. Diehl, A. Kerren (Eds.) Proceedings of the GI-Workshop "Software Visualization" SV2000. Technical Report A/01/2000, FR 6.2 - Informatik, University of Saarland, May 2000. http://www.cs.uni-sb.de/tr/FB14
9. S. Diehl, C. Görg, A. Kerren. Foresighted Graphlayout. Technical Report A/02/2000, FR 6.2 - Informatik, University of Saarland, December 2000. http://www.cs.uni-sb.de/tr/FB14
10. S. Diehl, C. Görg, A. Kerren. Preserving the Mental Map using Foresighted Layout. In: *Proceedings of Joint Eurographics – IEEE TCVG Symposium on Visualization VisSym'01*, 2001
11. S. Diehl, A. Kerren. Increasing Explorativity by Generation. In: *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, EDMEDIA-2000*. AACE, 2000
12. S. Diehl, A. Kerren. Levels of Exploration. In: *Proceedings of the 32nd Technical Symposium on Computer Science Education, SIGCSE 2001*. ACM, 2001
13. 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
14. S. Diehl, T. Kunze. Visualizing Principles of Abstract Machines by Generating Interactive Animations. *Future Generation Computer Systems*, 16(7), 831–839, 2000
15. 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
16. P. Eades, K. Zhang (Eds.) *Software Visualization*. World Scientific Pub., Singapore, 1996
17. GaniFA. Download and Documentation. http://www.cs.uni-sb.de/GANIMAL/GANIFA
18. GANIMAL. Project Homepage. http://www.cs.uni- sb.de/GANIMAL
19. F. Hopgood. Computer Animation Used as a Tool in Teaching Computer Science. In: *Proceedings IFIP Congress*, 1974
20. A. Kerren. Animation of the Semantical Analysis (in German). Master's thesis, University of Saarland, Saarbrücken, Germany, 1997
21. A. Kerren. Animation of the Semantical Analysis. In: *Proceedings of 8. GI-Fachtagung Informatik und Schule INFOS99 (in German)*, Informatik aktuell. Berlin Heidelberg New York: Springer 1999
22. K. Knowlton. L6: Bell Telephone Laboratories Low-Level Linked List Language. 16-minute black-and-white film, 1966
23. D.E. Knuth. *Literate Programming*. Center of the Study of Language and Information – Lecture Notes, No. 27. CSLI Publications, Stanford, California, 1992
24. R. Schulmeister. *Foundations of Hypermedial Learning Systems (in German)*. Bonn: Addison Wesley 1996
25. J. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23(9):27–39, 1990
26. J. Stasko. Using Student-Built Algorithm Animations as Learning Aids. In: *Proceedings of the 1998 ACM SIGCSE Conference*, San Jose, CA, 1997
27. J. Stasko, E. Kraemer. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18(2), 258–264, 1993
28. J.T. Stasko. The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990
29. J.T Stasko, J. Domingue, M.H. Brown, B.A. Price. *Software Visualization*. MIT Press, 1998
30. V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4), 315–339, 1990
31. A.Y. Tal, D.P. Dobkin. Visualization of Geometric Algorithms. *IEEE*

Transactions on Visualization and Computer Graphics, 1(2), 194–204, 1995
32. R. Wilhelm, D. Maurer. *Compiler Design: Theory, Construction, Generation*. Reading, MA: Addison-Wesley, 2nd edn., 1996
33. R. Baecker (with assistance of D. Sherman). Sorting out Sorting. 30 minute color film (distributed by Morgan Kaufmann Pub.), 1981

Stephan Diehl studierte Informatik und Computerlinguistik in Saarbrükken und Computer Science als Fulbright-Stipendiat am Worcester Polytechnic Institute, Massachusetts. Er promovierte 1993 als Stipendiat der DFG an der Universität des Saarlandes. Anschließend arbeitete er als wissenschaftlicher Assistent in der Forschungsgruppe von Prof. Reinhard Wilhelm und als Lehrstuhlvertreter an der Universität Duisburg. Stephan Diehl ist Autor mehrerer Bücher und hat in den letzten vier Jahren über 30 wissenschaftliche Papiere zu Themen aus den Bereichen der Programmiersprachentheorie, Internettechnologie, Visualisierung sowie zu Java und VRML veröffentlicht. Er ist aktives Mitglied von GI, ACM und des Web3D-Konsortiums. Stephan Diehl war u.a. Program Chair der vierten internationalen ACM SIGGRAPH Konferenz VRML99 und General Chair der gleichen Konferenz im Jahr 2001 sowie Mitorganisator des GI-Workshops *Softwarevisualisierung* und des internationalen Dagstuhl-Seminars *Software Visualization*. Seit Sommer 1998 ist er Projektleiter des von der DFG geförderten Projekts GANIMAL.

Andreas Kerren studierte Informatik und Betriebswirtschaftslehre an der Universität des Saarlandes in Saarbrücken. Nach seinem Diplom 1997 arbeitete er als wissenschaftlicher Mitarbeiter am IT Transfer Office (ITO) der TH Darmstadt. Seit Sommer 1998 ist er wissenschaftlicher Mitarbeiter am Lehrstuhl für Programmiersprachen und Compilerbau von Prof. Dr. Reinhard Wilhelm an der Universität des Saarlandes. Herr Kerren arbeitet im Team des von der DFG geförderten Projekts GANIMAL, ist in das BMBF Projekt ULI involviert und war Mitorganisator des GI-Workshops *Softwarevisualisierung 2000*. Seine aktuellen Arbeitsgebiete liegen in den Bereichen Programmiersprachen, Java, Lernsysteme und Softwarevisualisierung.