Prüfer:	Prof. Dr. D. Roller
Betreuer:	Dipl.-Inform. Monika Bihler

begonnen am:	1. Oktober 1994
beendet am:	31. März 1995

Diplomarbeit Nr. 1220

Graphischer Editor zur Festlegung multimedialer Präsentationen mittels visueller Programmierung

Jürgen Weber

Institut für Informatik
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Motivation

Ein neues und faszinierendes Gebiet der Informatik umfaßt der Begriff Multimedia. Darunter versteht man den gleichzeitigen Einsatz mehrer unterschiedlicher Medien wie Bild, Video oder Ton zur Informationsdarstellung auf dem Rechner. Aus einzelnen Medienbausteinen können Multimediapräsentationen zusammengestellt werden, die Information weitaus effizienter und ansprechender darstellen als konventionelle Informationsübermittlung vom Rechner zum Menschen mit textueller Darstellung. Multimediapräsentationen bietet sich eine weites Einsatzfeld, das vom Einsatz in Informationskiosken über rechnerunterstütztes Lernen bis zum Einsatz im betrieblichen Informationsmanagement reicht.

Die Erstellung multimedialer Präsentationen ist bei Einsatz konventioneller Programmiersprachen jedoch sehr mühsam und aufwendig und erfordert professionelle Programmierkenntnisse. Wünschenswert wäre aber, daß auch Anwender wie z.B. Sachbearbeiter in Firmen eine Präsentation erstellen könnten, ähnlich wie sie mit den vorhandenen Textverarbeitungssystemen nach kurzer Einarbeitungszeit einen Text erstellen können. Daher müssen spezielle, leicht und intuitiv bedienbare, computerbasierte Werkzeuge für das `multimedia authoring` entwickelt werden. Multimediapräsentationen lassen sich durch einen Graphen modellieren, in dem die Darstellung von Medien durch Knoten und zeitliche Beziehungen zwischen den Darstellungen einzelner Medien durch Kanten repräsentiert werden. Ein Ansatz zur Erstellung von Multimediapräsentationen ist daher, mit einem interaktiven graphischen Editor eine graph–artige Visualisierung der Präsentation einzugeben.

In dieser Arbeit soll ein Konzept für einen solchen graphischen Editor zur visuellen Erstellung multimedialer Präsentationen in einem betrieblichen Umfeld entwickelt werden. Der Ablauf der Präsentation soll unter Einsatz von Drag und Drop direkt–manipulativ festgelegt werden und in eine Scriptsprache übersetzt werden können.

Das erstellte Konzept soll prototypisch unter X Windows / OSF–Motif in C++ implementiert werden.

Kapitel 2

Einleitung

Die Aufgabe der vorliegenden Arbeit war, einen graphischen Editor zur Festlegung multimedialer Präsentationen mittels visueller Programmierung zu entwickeln. Diese Arbeit fließt in ein Forschungsprojekt ein, in dem Anforderungen an eine integrierte, umfassende Produktdatenmodellierung und Konzept für universelle Informationssysteme im CAD/CAM–Bereich entwickelt werden.

2.1 Das Projekt POWER

Der industrielle Entwicklungs- und Fertigungsbereich ist von verkürzten Produktentwicklungszeiten und höherer kunden- und marktgerechter Flexibilität gekennzeichnet. Daraus resultieren aber auch höhere Anforderungen an Verarbeitung und Verfügbarkeit von Informationen. Rechnergestützte Dokumentation ist im industriellen Umfeld momentan noch durch Insellösungen gekennzeichnet (vergl. [Roller et al. 1995]). Dadurch wird ein integriertes, bereichsübergreifendes Informationsmanagement verhindert. Im Rahmen des Projektes POWER (Product modelling in object–oriented Databases with efficient Mechanisms for Retrieval) wird nach Lösungen für diese Problematik gesucht. Mit einem einheitlichen Datenmodell sollen Produktdaten umfassend modelliert und für integrierte Datenhaltung in allen Unternehmensbereichen verwendet werden können. Hierzu müssen Daten geeignet aufbereitet und präsentiert werden. Für die Erfüllung der letzteren Anforderung bietet sich ein multimediales Informationssystem an. In einer früheren Arbeit ([Bröckel 1994]) wurde ein solches System entwickelt.

2.2 Das Multimedia Informations System

Das Multimedia Informations System wurde für die Verwaltung und Präsentation der während eines Produktentwicklungszyklusses anfallenden Dokumente entwickelt. Dieses System ist darauf ausgerichtet, multimediale Daten beliebigen Formates durch externe Viewer darstellen zu können. Als Viewer werden freie

Programme verwendet, was in einer inhomogenen Bedienung des Gesamtsystems resultiert. Das Informationssystem stellt dabei unterschiedliche Medien automatisch mit dem richtigen Viewer dar. In einer Autorensprache können Multimediapräsentationen geschrieben werden, die vom Informationssystem dargestellt werden. Das Informationssystem enthält einen Dateimanager (Navigator), mit dem multimediale Daten ausgewählt und dargestellt werden können. Auch ist es möglich, diese Auswahl aufzuzeichnen und ein Programm in der Autorensprache zu erzeugen. Dieses Programm wird dann in ein Shell Script übersetzt und ausgeführt. Allerdings können so nur rein sequentielle Präsentationen erstellt werden. Da die Aufzeichnung wie ein aus Textverarbeitungen bekannter Makrorecorder arbeitet und im Hintergrund die Auswahl notiert, hat der Benutzer auch keine Visualisierung der Präsentationsstruktur.

Diese Einschränkungen gaben den Anstoß für die vorliegende Arbeit.

2.3 Der graphische Editor VisEd

Um Präsentationen auf visuellem Weg erstellen zu können, wurde ein graphischer Editor (VisEd) zur Spezifikation von Multimediapräsentationen entwickelt. Der Benutzer erstellt mit VisEd eine graphische Repräsentation des Präsentationsaufbaus. Die entwickelte Präsentation kann als Quellcode der im Multimediainformationssystem verwendeten Autorensprache gespeichert werden. Auch ist es möglich, existierenden Quellcode einzulesen und zu bearbeiten. In Abbildung 2.1 ist die Einbindung des VisEd Editors in das Multimediainformationssystem dargestellt.

2.4 Aufbau der vorliegenden Arbeit

Bei der Entwicklung des graphischen Editors VisEd wurde auf die Erkenntnisse verschiedener Teildisziplinen der Informatik aufgebaut.

Kapitel 3 gibt eine kurze Darstellung der Multimedia und Hypermedia Technologie sowie von Möglichkeiten der Erstellung von Multimediapräsentationen.

VisEd kann direkt manipulativ unter einer graphischen Benutzeroberfläche bedient werden. Hierfür wurden Kriterien für benutzergerechte Schnittstellen verwendet, die von Softwar–Ergonomie entwickelt wurden. In Kapitel 4 werden einige wichtige Erkenntnisse der Software–Ergonomie vorgestellt.

Der Editor ermöglicht, mittels visueller Programmierung multimediale Präsentationen zu erstellen. In Kapitel 5 wird das Gebiet der visuellen Programmierung und damit zusammenhängende Aufgaben, wie die Entwicklung geeigneter graphischer Darstellungen für Programme, beschrieben. Da VisEd Icons zur Repräsentationen von Statements der erstellten visuellen Programmiersprache verwendet, wird auch auf den Entwurf von Icons eingegangen. Auch werden existierende

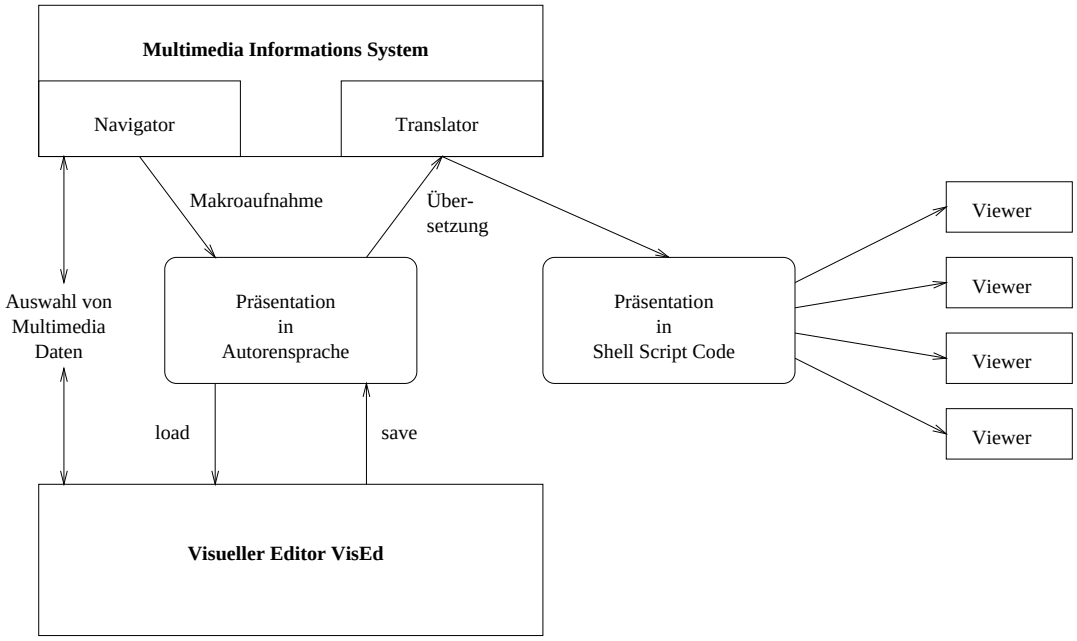

Abbildung 2.1: Einbindung des Editors in das Gesamtsystem

Systeme zur visuellen Programmierung beschrieben.

Im folgenden Kapitel werden einige existierende Multimedia– bzw. Hypermediasysteme beschrieben. XMAD und MAEstro entstanden bei Forschungsprojekten an Universitäten, HyperCard und Authorware Professional sind kommerzielle Produkte.

Kapitel 7 beschreibt das X Window System und die Toolkits, welche die Systemumgebung für den graphischen Editor bilden. Auch wird auf die bei der Entwicklung des Editors eingesetzte Programmiersprache C++ und die verwendeten Softwarewerkzeuge eingegangen.

Der VisEd Editor ist in der Lage, bestehende Quelltexte der Autorensprache einzulesen. Hierfür mußten mit Compilergeneratoren Teile eines Compilers entwickelt werden. Kapitel 8 beschreibt Aspekte des Compilerbaus und Compilergeneratoren.

Es folgen Kapitel über das verwendete Lösungskonzept, die Autorensprache und die bei der Entwicklung des Editors verwendeten Techniken.

Kapitel 3

Hypermedia und Multimedia

Ein Multimediasystem ist eine Software, die Verarbeitung und Darstellung heterogener Medien, wie Text, Graphik, Video oder Audio, mit Rechnern ermöglicht (vergl. [Götze 1994]). Durch Kombination dieser Einzelmedien (Monomedien) ergibt sich ein neues zusammengesetztes Medium. Man erhofft sich durch die Verwendung von Multimedia die Effizienz der Kommunikation zwischen Rechner und Mensch zu erhöhen, da dadurch mehr Sinnesorgane des Menschen angesprochen werden können als bei Verwendung von Monomedien. Die dadurch erreichte Verbesserung der Mensch–Maschine Schnittstellen trägt zu einer besseren Akzeptanz und zu breiten Einsatzmöglichkeiten von Rechnern bei (vergl. [Steinmetz 1993]).

Medien sind Mittel zur Verbreitung und Darstellung von Information. Steinmetz teilt Medien in zeitunabhängige (zeitinvariante, diskrete) Medien wie Text oder Einzelbilder, deren Darstellungsdauer nicht festgelegt ist, und in zeitabhängige (kontinuierliche) Medien wie Video und Audio ein, bei denen die Information auch durch die Abfolgereihenfolge der Medienbestandteile (Mediensegmente) codiert ist und bei deren Darstellung zeitliche Restriktionen eingehalten werden müssen. Die Darstellung kontinuierlicher Medien erfordert daher einen hohen technischen Aufwand, zumal dabei große Datenmengen bearbeitet werden müssen. Erst wenn in einem System gleichzeitig zeitinvariante und kontinuierliche Medien verwendet werden können, spricht man von Multimedia. Eine weitere Forderung ist die Unabhängigkeit der einzelnen Medien. Die Ansteuerung eines Videorecorders durch einen Rechner ist in diesem Sinne keine Multimediaanwendung, da Ton und Bild auf dem Videoband nicht getrennt abgespielt werden können. Steinmetz ([Steinmetz 1993]) gibt daher folgende Definition des Begriffs Multimediasystem: “Ein Multimedia–System ist durch die rechnergesteuerte, integrierte Erzeugung, Manipulation, Darstellung, Speicherung und Kommunikation von unabhängigen Informationen gekennzeichnet, die in mindestens einem zeitabhängigen und einem zeitunabhängigen Medium kodiert sind.” Nach dieser Definition ist also ein mit Bildern unterlegter Text kein Multimediasystem.

Ein wichtiger Aufgabenbereich bei der Verwendung von Multimedia in Applikationen ist die Integration der neuen Medien in die Benutzerschnittstelle. Da

Multimediasysteme verschiedene diskrete und kontinuierliche Medien unterstützen müssen, müssen für Multimedia Benutzerschnittstellen neue Techniken und Softwarewerkzeuge entwickelt werden (vergl. [Götze 1994]). Aufgrund der beschriebenen Eigenschaften von Multimediasystemen sind multimediale Schnittstellen durch Zeitinvarianz, Nebenläufigkeit und Interaktivität gekennzeichnet.

3.1 Präsentationen

Visuelle Monomedien, die in einem Multimediasystem verwendet werden, müssen auf dem Bildschirm dargestellt werden, akustische Medien müssen über Lautsprecher ausgegeben werden. Werden in der Zukunft noch weitere Medien verwendet, die z.B. den Tastsinn oder Geruchsinn des Menschen ansprechen, müssen hierfür geeignete Ausgabegeräte gefunden werden. Meistens werden Monomedien durch seperate spezialisierte Softwareprogramme bearbeitet und abgespielt, die auf geeignete Weise miteinander kommunizieren. Drapeau ([Drapeau 1993]) nennt solche Programme Medieneditoren. Es ist auch möglich, daß die Darstellung der einzelnen Medien durch ein einziges Programm erfolgt, ein Beispiel hierfür ist Authorware Professional (vergl. Abschnitt 6.4).

Die kombinierte Darstellung der einzelnen Monomedien bezeichnet man als Multimedia Präsentation. Die Gesamtheit der für die Ablaufsteuerung der Präsentation und zur Darstellung der Medien notwendigen Daten heißt Multimediadokument. Als idealer Datenträger für ein Multimediadokument bieten sich CD-Roms an.

Neben Multimediasystemen existieren auch als Hypertext oder Hypermedia bezeichnete Systeme. Sie entwickelten sich aus normalen Textdokumenten, in die aktive Stellen oder Buttons eingefügt wurden, die zu anderen Stellen im Dokument führen. Man erhält dann Hypertextsysteme. Sind in den Text noch andere Medien eingebunden, spricht man von Hypermedia.

Multimedia– und Hypermediadokumente können durch einen Graphen modelliert werden, in dem die Knoten elementare oder komplexe Informationen repräsentieren. Semantische Beziehungen zwischen diesen Informationselementen finden ihre Entsprechung in den Kanten zwischen den Knoten (vergl. [Götze 1994]). In Hypermediadokumenten muß der Benutzer nun aktiv zwischen diesen Knoten navigieren und damit in der Präsentation springen. Dies kann durch Anklicken von unterlegten Textstellen oder von Fortschaltbuttons geschehen. Von dieser Möglichkeit des Springens zwischen Informationseinheiten leitet Fisher den Ursprung der Bezeichnung Hypertext ab ([Fisher 1994]), da Science Fiction Autoren schon früh den Begriff Hypersprung für verzögerungsfreie Bewegung im Weltraum unter Ausnutzung eines Hyperraumes verwendet hätten.

Das Charakteristische an multimedialen Präsentationsanwendungen ist dagegen, daß der Benutzer vom System durch die Präsentation geführt wird und oft nur geringe Einflußmöglichkeiten auf die Präsentation hat (vergl. [Götze 1994]).

Es fehlen also die Hypermedialinks.

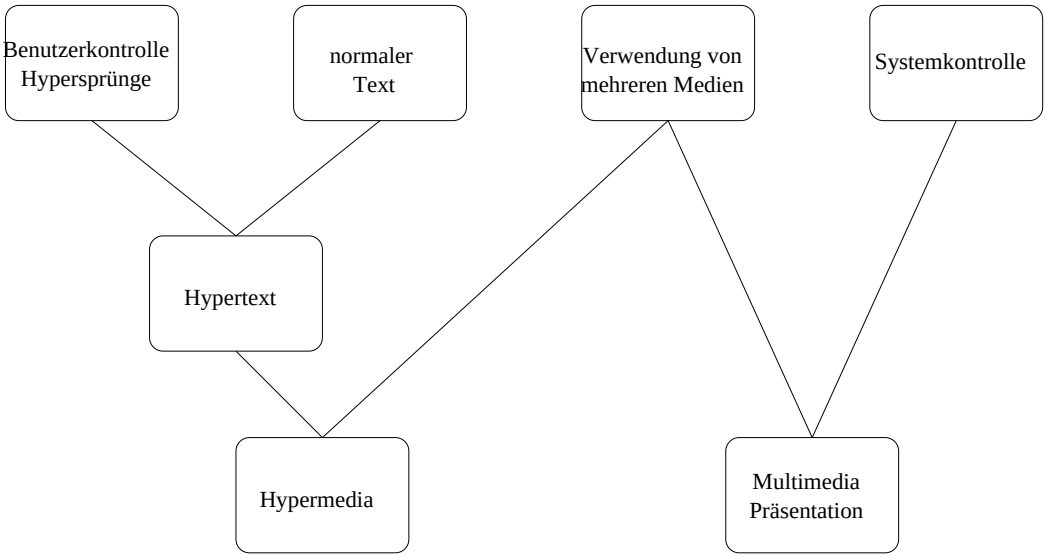

Abbildung 3.1: Begriffe als Klassenhierarchie

Demnach lassen sich die Möglichkeiten der Informationsdarstellung durch den Rechner in einer Art Klassenhierarchie wie in Abb. 3.1 darstellen. Allerdings hält Götze eine eindeutige Unterscheidung zwischen Hypermedia-Systemen und multimedialen Präsentationsanwendungen häufig für nicht möglich, da die Übergänge zwischen beiden Arten fließend sind.

Schreyjak ([Schreyjak 1994]) dagegen führt Hypermedia als spezielles Erstellungsmodell für Multimediadokumente an (vergl. Abschnitt 3.3).

3.2 Einsatzmöglichkeiten

Hypermedia– und Multimediapräsentationssysteme bietet sich ein weites Einsatzfeld. Hypertextsysteme werden häufig für Hilfesysteme unter graphischen Benutzeroberflächen eingesetzt. Hier wäre eine Ergänzung mit weiteren Medien nützlich, z.B. könnten Filme die Arbeit mit Objekten der Benutzeroberfläche besser erklären als Text. Generell könnten z.B. Reparatur– und Betriebsanleitungen mit Hyper– oder Multimediasystemen besser erklärt werden als mit Text. Fisher ([Fisher 1994]) entwickelt als ein Beispiel dafür ein System, das einen Mechanikerkurs zur Reperatur eines bestimmten Sportwagens bildet. Die richtige Einstellung des Motors kann dabei am besten über die akustische Wiedergabe des Motorgeräusches gezeigt werden.

In Fabriken könnten multimediale Informationskioske aufgestellt werden, an denen sich Arbeiter informieren könnten, wie ein bestimmter Arbeitsschritt zu erledigen ist. Interessante Einsatzgebiete finden sich auch in der Werbung, die Pro-

dukte multimedial darstellen könnte. An Hochschulen können Forschungsergebnisse mit Multimedia besser präsentiert werden. Sachbearbeiter können anderen betrieblichen Abteilungen Arbeitsergebnisse als Multimediadokumente präsentieren.

3.3 Erstellungsmodelle für Multimediapräsentationen

Das Erstellen von guten Multimediapräsentationen ist eine kreative Tätigkeit, bei der dem Autor möglichst große Freiheiten gegeben werden sollten. ein gutes Autorensystem sollte daher dem Autor individuelle Vorgehensweisen ermöglichen. Beim Entwurf kann top–down oder bottom–up vorgegangen werden. Beim top–down Entwurf wird zuerst ein grobes Modell der Präsentation erstellt, das durch Hinzunahme von immer mehr Details verfeinert wird. Dieser Entwurfsansatz sollte durch hierarchische Abstrahierungsmöglichkeiten unterstützt werden. Beim bottom–up Entwurf dagegen baut der Autor meist Teilpräsentationen, die schließlich zu einem Gesamtsystem zusammengesetzt werden.

Schreyjak ([Schreyjak 1994]) unterscheidet folgende Modelle, gemäß denen beim Schreiben von Multimediapräsentationen vorgegangen werden kann. Die Modelle sind aus Autoren– und Benutzersicht dargestellt und nicht unter implementierungstechnischen Aspekten, auf die weiter unten eingegangen wird.

- **Multimediales Buch:** Hierbei handelt es sich um ein weitgehend textbasiertes Dokument, das in einzelne Seiten strukturiert ist. Die Seiten können neben Text mit Bildern und zeitabhängigen Medien ergänzt werden. So könnte auf einer Buchseite ein mit Ton unterlegtes Video ablaufen, während daneben textuelle Erläuterungen stehen. Solche Präsentationen sind vor allem dann sinnvoll, wenn der größte Teil der Informationen im Text steht und weitere Medien nur zur Ergänzung dienen. HyperCard ist ein Beispiel für ein System, mit dem soche Multimediabücher erstellt werden können (vergl. Abschnitt 6.3). Die Seiten des Buches müssen weitergeblättert werden. Dies kann zeitlich gesteuert erfolgen oder durch Dialogobjekte wie Buttons durch den Benutzer bestimmt werden. Zusätzlich zu den zeitlichen Restriktionen bei Multimediapräsentationen kommen bei der Buchform auch geometrische Restriktionen zum Tragen, um die Multimediadarstellungen in einem ansprechenden Layout auf den Bildschirm zu bringen.

  Ähnlich wie Bücher können auch Diashows oder Folienpräsentationen, wie sie in der Geschäftswelt oder in der Lehre eingesetzt werden, multimedial aufbereitet und auf den Rechner umgesetzt werden. Statt aus Buchseiten besteht die Präsentation dabei aus Folien. Merkmale dieser Präsentationsart sind eine lineare Ablaufstruktur und die Konzentration auf visuelle Eindrücke.

- **Zeitachsen:** Nach dem Zeitachsenmodell entwickelte Präsentationen orientieren sich nicht so sehr an Analogien zu bekannten Medien wie Büchern. Vielmehr wird mehr Wert auf die zeitlichen Beziehungen zwischen einzelnen Medien gelegt. Die Monomedien werden auf einer Zeitachse plaziert. Die Reihenfolge der Darstellung ist streng sequentiell und wird vom Autor der Präsentation festgelegt.

- **Hypermedia:** Nach dem Hypermediamodell erstellte Präsentationen arbeiten ähnlich wie Multimediabücher. Der Benutzer muß die Präsentation jedoch nicht sequentiell durchgehen, sondern kann eine beliebige Durchlaufreihenfolge durch ein Netz von Informationen wählen, er hat einen wahlfreien Zugriff auf Informationen. Der Benutzer kann dabei aktiv Verweisen folgen, indem er z.B. mit Mausklicks farbig unterlegte Stellen im Dokument auswählt, die auf andere Informationseinheiten verweisen.

Das im Rahmen des Projekt POWER entwickelte Multimediainformationssystem kann in keine dieser Klassen eindeutig eingeordnet werden. Am ehesten entspricht die Präsentation betrieblicher Daten dem Diashowmodell, hat aber auch Bezüge zu den anderen vorgestellten Konzepten.

3.4 Spezifikation von Präsentationen

Bei der Erstellung von Multimediasystemen muß der Autor eine Idee und Vorstellung über Inhalt und Aussehen der Präsentation haben. Mit Hilfe von Softwarewerkzeugen muß diese Idee in eine computerbasierte Präsentation umgesetzt werden.

An den Autor stellen sich nach [Inwood et al. 1994] dabei folgende Anforderungen:

- Der Autor muß Kenntnisse über den Informationsinhalt der Präsentation haben. Eine Präsentation soll Information weitergeben.

- Ebenfalls sind Kenntnisse über den Benutzer und dessen Bedürfnisse notwendig. Multimedia erlaubt dem Autor, Informationen speziell auf den Benutzer auszurichten.

- Der Autor braucht eine Präsentationsstrategie. Er muß wissen, wie die Informationen präsentiert werden sollen, z.B. ob der Benutzer selbst auswählen, kann was er sehen will, oder ob die Präsentation systemgesteuert erfolgt.

- Der Autor muß wissen, in welcher Benutzeroberflächenumgebung das Multimediasystem laufen wird, um zu wissen, welche Dialogmittel zur Verfügung stehen.

- Der Autor sollte über ästhetische und kreative Fähigkeiten verfügen, um eine ansprechende Präsentation erstellen zu können.
- Schließlich sollte der Autor über Mittel und Werkzeuge verfügen, um tatsächlich eine Präsentation erstellen zu können.

Es fällt auf, daß keine Programmierkenntnisse verlangt werden. Der Autor sollte sich auf den Informationsinhalt der Präsentation konzentrieren können und für die technische Erstellung der Präsentation ein mächtiges Werkzeug verwenden können. Ein solches Werkzeug ist der in dieser Arbeit erstellte graphische Editor VisEd.

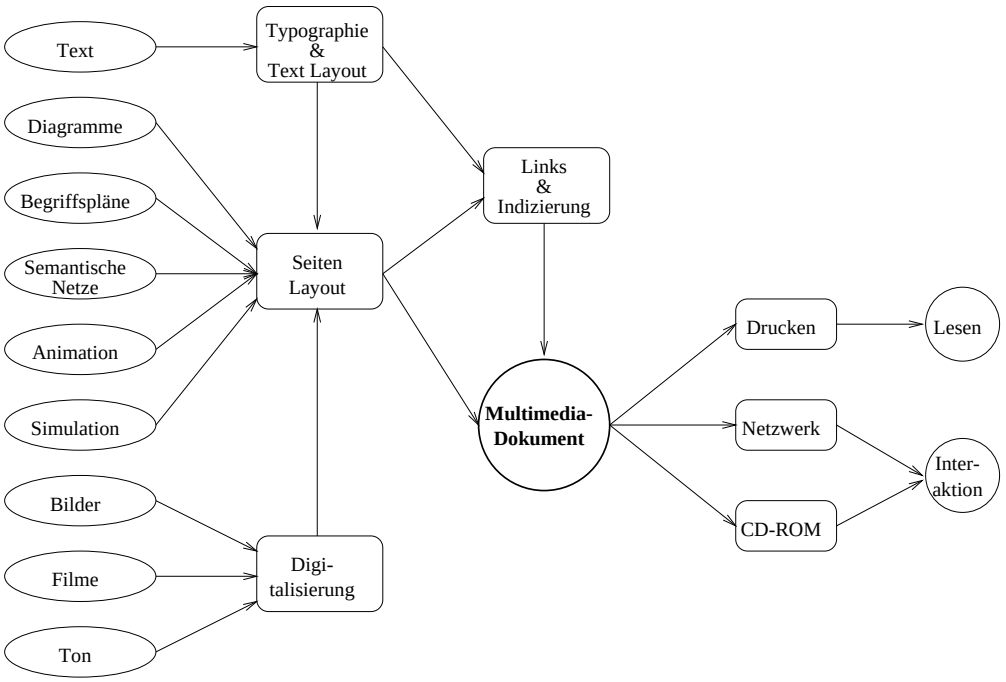

Abbildung 3.2: Produktion von Multimedia Dokumenten (aus [Gaines, Shaw 1993])

In Abbildung 3.2 sind die Schritte dargestellt, wie aus Monomedien ein Multimediadokument produziert wird. Dabei müssen sowohl räumliche als auch zeitliche Beziehungen zwischen Mediensegmenten spezifiziert werden. Die räumliche Restriktion beim Bildschirmlayout der Präsentation legt der Autor meist interaktiv durch Verscheiben der Darstellungsfenster fest (vergl. [Schreyjak 1994]). Spezifikationen der zeitlichen Beziehungen werden mit Hilfe von Autorensystemen entwickelt. Hier können schwierige Synchronisationsprobleme auftreten. Soll z.B. eine Ansprache einer Person mit getrennten Bild– und Tonspuren wiedergegeben werden, fallen schon geringe Verschiebungen zwischen Ton und Lippenbewegungen auf. Die sogenannte Lippensynchronität erfordert hohen technischen Auf-

wand. Bei Präsentationen für Produktdokumentationen im betrieblichen Umfeld ist jedoch keine solche Synchronisation nötig.

Schreyjak schlägt vor, den Ablauf einer Präsentation immer zuerst ohne Computerhilfe auf Papier zu skizzieren. Auch Fisher ([Fisher 1994]) befürwortet ein solches Vorgehen. In der Filmbranche wird eine spezielle Art von Skizzen, das Storyboard eingesetzt. Es stammt aus der Zeichentrickfilmproduktion. In den Studios von Disney war eine Wand mit einem großen Korkbrett bedeckt, auf dem Skizzen befestigt waren, welche die Szenen des Filmes repräsentierten. Die Abfolge der Skizzen auf der Wand entsprach der Abfolge der Szenen des Filmes. Das Storyboard war das zentrale Planungsinstrument für den Film. Heute wird das Storyboard bei fast allen Film- und Videoproduktionen verwendet. Fisher nennt als Hauptvorteile des Storyboards:

- Das Storyboard zwingt, in visuellen Termen zu denken und Ideen in informativen Skizzen darzustellen.
- Das Storyboard ist flexibel, Ideen, Bilder und Sequenzen von Ereignissen können leicht an eine andere Stelle des Filmes gehängt werden.
- Schließlich kann ein Storyboard Bilder oder Text enthalten, so daß es sich als natürliches Modell für die Erstellung multimedialer Präsentationen auf dem Rechner anbietet.

Ein Storyboard ähnelt damit einem Flußdiagramm.

3.5 Spezifizierungs–Modelle

Das erstellte Storyboard kann mit unterschiedlichen Spezifizierungsverfahren in eine rechnergesteuerte Präsentation umgesetzt werden. Schreyjak beschreibt folgende Verfahren ([Schreyjak 1994]):

3.5.1 Courseware

Hierbei wird eine Multimediapräsentation in einer herkömmlichen Programmiersprache erstellt. Dieses Verfahren erbt alle Vor- und Nachteile von konventioneller Programmierung. Es ist ein hoher Erstellungsaufwand erforderlich, jedoch kann die volle Flexibilität von Programmiersprachen eingesetzt werden. Oft ist die Funktionalität zur Manipulation von Multimediadaten in spezielle Bibliotheken ausgelagert.

So kann z.B. mit der Multimedia Extention Bibliothek für Microsoft Windows eine Sounddatei durch folgenden Aufruf einer C Bibliotheksfunktion abgespielt werden:

```
sndPlaySound("C:\\SOUNDS\\BELLS.WAV",SND_SYNC);
```

(Beispiel aus [Microsoft Press 1991]).

3.5.2 Formale Sprachen

Die Präsentation wird in einer formalen Sprache beschrieben. Auch die Autorensprache, auf die der in der vorliegenden Arbeit entwickelte Editor VisEd aufbaut, ist eine solche formale Sprache. Um die Erstellung von Präsentationen auf einfache Art zu ermöglichen, wird eine formale Sprache oft mit Mitteln der visuellen Programmierung dargestellt.

3.5.3 Zeitachsenmodell

Bei diesem Modell existiert für jedes Monomedium, das bei einer Multimediapräsentation verwendet werden soll, eine Zeitachse. Darauf werden die einzelnen Segmente (Teilbereiche) der Monomedien plaziert. Damit wird die zeitliche Abfolge der Präsentationen definiert. Wenn Mediensegmente übereinander beginnen, ist damit ein gemeinsamer Startpunkt impliziert (vergl. Abb. 3.3). Wesentliche

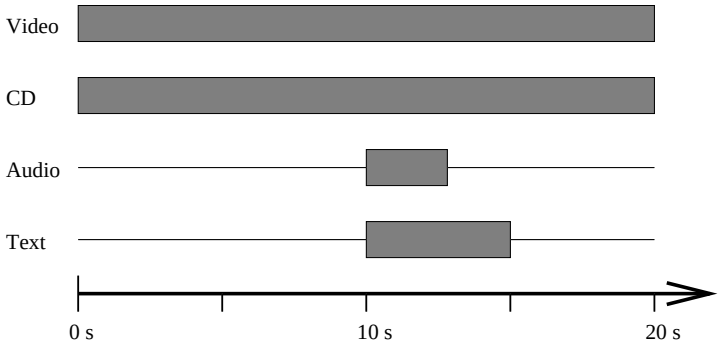

Abbildung 3.3: Spezifikation nach dem Zeitachsenmodell (aus [Schreyjak 1994])

Nachteile dieses Modells sind, daß die Länge der Medien exakt bekannt sein muß und keine Benutzerinteraktion möglich ist, da der Zeitbedarf für die Benutzerinteraktion im voraus nicht bekannt ist. Der graphische Editor VisEd ist aber zur Erstellung von Präsentationen mit Benutzerinteraktion vorgesehen und damit kann das Zeitachsenmodell nicht verwendet werden.

3.5.4 Petrinetze

Dieses Modell stammt aus der Erstellung paraller Systeme, also aus dem Gebiet der Betriebssysteme. Es kann jedoch auf parallele Präsentationen übertragen werden.

Multimediapräsentationen, die im Rahmen des Projektes POWER entwickelt werden, arbeiten mit Darstellung von Medien durch externe Viewer. Eine Darstellung ist dabei eine unteilbare Aktion. Es existieren keine Referenzzeitpunkte innerhalb von Darstellungen. Mögliche Beziehungen zwischen Darstellungen sind dabei gemeinsamer Start und parallele Ausführung. Solche Beziehungen können

durch das Petrinetzmodell gut ausgedrückt werden. In Abbildung 3.4 ist eine Spezifikation einer Präsentation durch ein Petrinetz abgebildet.

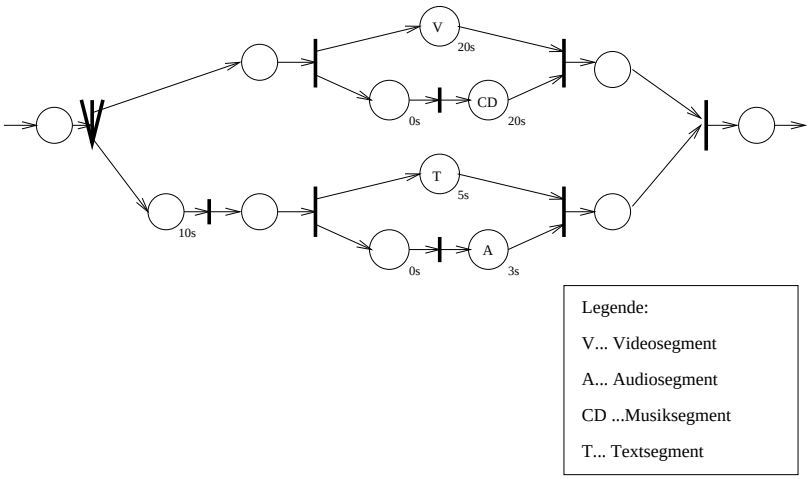

Abbildung 3.4: Beispielpräsentation im Petrinetzmodell (aus [Schreyjak 1994])

Das Petrinetzmodell kann auch so erweitert werden, daß eine Ersetzung von Subnetzen durch eine einzelne Stelle möglich ist (vergl. Abb. 3.5). Damit ist eine Möglichkeit der Abstraktion und Hierarchisierung gegeben.

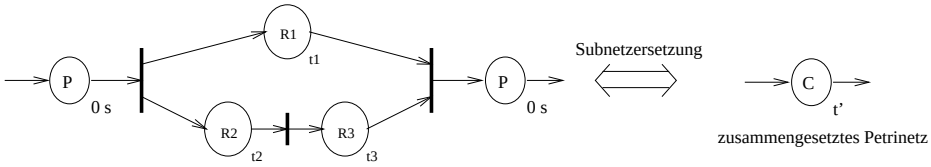

Abbildung 3.5: Subnetzersetzung (aus [Schreyjak 1994])

Beim graphischen Editor VisEd wurde eine leicht abgewandelte Version des Petrinetzmodells zur Spezifikation von Präsentationen verwendet (vergl. Kapitel 9). Eine der Subnetzersetzung entsprechende Art der Hierarchisierung ist mit dem Konzept der Compound–Statements vorhanden. Teilpräsentationen können dadurch gruppiert werden.

Kapitel 4

Konzepte der Software–Ergonomie

Im vorigen Kapitel wurden der Einsatzbereich des graphischen Editors VisEd sowie Modelle für die Erstellung von Präsentationen vorgestellt. Um ein computerbasierte Werkzeuge wie ein Editor zur Erstellung von Multimediapräsentationen möglichst effizient einsetzen zu können, müssen beim Entwurf der Benutzeroberfläche ergonomische Grundsätze und Richtlinien eingehalten werden, die von der Software–Ergonomie erforscht werden. Im Folgenden wird daher diese Wissenschaft kurz vorgestellt. Es werden moderne Dialogformen wie direkte Manipulation beschrieben, die für einen graphischen Editor besonders geeignet sind. Schließlich wird auf das MVC–Paradigma eingegangen, ein Modell zur Modularisierung von interaktiven Computeranwendungen. Im letzten Abschnitt wird diskutiert, wie die vorgestellten Erkenntnisse der Software–Ergonomie für die Entwicklung des graphischen Editors verwendet werden können.

Eine breite Übersicht der Software–Ergonomie findet sich bei Herczeg, dessen Darstellung die nächsten vier Abschnitte folgen ([Herczeg 1994]).

4.1 Arbeitsbereich der Software–Ergonomie

Die Software–Ergonomie ist das Teilgebiet der Informatik, welches sich mit der Bereitstellung von Leitlinien und Rahmenbedingungen zur Gestaltung benutzergerechter computerbasierter Werkzeuge beschäftigt (vergl. [Herczeg 1994]). Sie stützt sich auf Erkenntnisse über die menschliche Informationsaufnahme und Verarbeitung, die aus Medizin, Psychologie und Arbeitswissenschaften stammen.

Das Ziel der Software–Ergonomie ist die Entwicklung von menschengerechten und nützlichen computerbasierten Werkzeugen, die bei den Benutzern keine unnötige körperliche oder geistige Belastung hervorrufen. Davon erhofft man sich gesteigerte Leistungsfähigkeit und Motivation und eine bessere Akzeptanz des Rechners bei Benutzern. Geistige Belastungen sind nach Herczeg:

- Belastung des Gedächtnisses
- Belastung der Konzentrationsfähigkeit
- Demotivation durch Mißerfolge oder negative Kommentare
- unverständliche oder sich ständig ändernde Informations- und Aktionsstrukturen, so daß sich kein einfaches Systemmodell beim Benutzer bilden kann
- Angst vor nicht reversiblen Aktionen mit hoher Tragweite
- Angst vor Beobachtung und Leistungsbeurteilung

Das System soll an den Benutzer angepaßt sein und nicht umgekehrt. Traditionelle Werkzeuge wurden in einem sehr langen evolutionären Prozeß an die menschliche Anatomie und die menschlichen Fähigkeiten angepaßt. Da es Computer erst seit vergleichsweise kurzer Zeit gibt, fehlen bei der Gestaltung computergestützter Werkzeuge Erfahrungen aus einem ähnlich langen evolutionären Prozeß. Hinzu kommt, daß lange Zeit bis zum Aufkommen hochauflösender Bildschirme mangelnde Hardwarefähigkeiten benutzergerechten Programmoberflächen entgegenstanden.

Aufbauend auf diese Kenntnisse muß die Software–Ergonomie für die Mensch–Computer Schnittstellen von rechnerbasierten Applikationen Gestaltungsgrundsätze liefern. Herczeg nennt als nutzbare Ergebnisse der Software–Ergonomie die folgenden Hilfsmittel des Softwareentwicklers:

- Normen
- Empfehlungen (Vorschläge)
- Designregeln (Richtlinien)
- Tools (Software–Bausteine und Entwicklungswerkzeuge)

Maßgeblich für Nützlichkeit und Nutzbarkeit von Computersystemen ist die Qualität der Kommunikation zwischen Mensch und Rechner. Die Regeln, nach denen der Dialog zwischen Benutzer und System abläuft, werden durch die Benutzungsschnittstelle festgelegt.

4.2 Interaktionstechniken

Das System teilt sich dem Benutzer in einer Ausgabesprache durch visuelle oder akkustische Ausgaben mit. Diese Sprache ist durch das Ausgabealphabet, z.B.

Text oder graphische Symbole, und die Ausgabesyntax festgelegt. Diese legt Reihenfolge und Anordnung der Elemente des Ausgabealphabets fest. Der Benuzter teilt dem System Anforderungen und Reaktionen auf Systemausgaben mit. Hierfür verwendet er eine Eingabesprache und muß dazu eine Eingabesyntax einhalten. Herczeg unterteilt die Möglichkeiten der Interaktion mit dem System in

1. Deskriptive Interaktionsformen auf der Grundlage sprachlicher Beschreibungen und

2. Deiktische Interaktionsformen auf der Grundlage von Selektionen mittels Zeigehandlungen

Des weiteren sind Mischformen möglich.

Bei den deskriptiven Interaktionsformen muß der Benutzer seine Eingaben aktiv in einer textuellen Sprache formulieren. Dabei ist Gedächtnisleistung gefordert, um die Sprache in Erinnerung zu rufen. Dies ist der Hauptnachteil von deskriptiven Interaktionsformen. Sie erfordern immer einen Lernaufwand, um die Sprache zu lernen. Arten von deskriptiven Interaktionsformen sind Symbole, wie Icons oder Funktionstasten, formale Sprachen, insbesondere Kommandosprachen und natürliche Sprachen.

Bei deiktischen, d.h. selektionsorientierten, Interaktionsformen kann der Benutzer aus einem dargebotenen Angebot von möglichen Systemaktionen auswählen. Der Benutzer muß sich dazu nicht mehr aktiv einen Befehl in Erinnerung rufen, sondern es genügt, die Bedeutung des Angebots wiederzuerkennen. Herczeg ([Herczeg 1994]) führt als Beispiel für deiktische Interaktionsformen die wohlbekannten Menüs, beschriftete Funktionstasten und metaphorische Dialoge an. Vor allem letztere setzten sich in den vergangenen Jahren immer mehr durch. Sie werden im folgenden Abschnitt genauer beschrieben.

In der Praxis treten deskriptive und deiktische Interaktionsformen meist als Mischformen auf, die dann die Vorzüge beider Grundtypen ausnützen können. Bei *Formularen* wird ein Feld aus einer Menge ausgewählt und dann deskriptiv ausgefüllt. Ein weiteres Beispiel sind die für geübte Benutzer vorgesehenen Shortcuts für Menüeinträge. Anstatt mit der Maus einen Menüpunkt auszuwählen, drückt er eine Tastenkombination.

4.3 Metaphorische Dialoge

Bei dieser Dialogform wird beim Benutzer vorhandenes Wissen über eine Arbeitsumgebung ausgenutzt, indem sie auf den Bildschirm in Form von Pictogrammen (Icons, vergl. Abschnitt 5.4) abgebildet wird. Diese Icons sind die Repräsentation (View, vergl. Abschnitt 5.3) der Objekte der Anwendung. Mit der Maus können die dargestellten Objekte direkt manipulativ (vergl. Abschnitt 4.4) bewegt oder

für die Bearbeitung ausgewählt werden. Kennzeichen von metaphorischen Dialogen ist jedoch die Visualisierung der Anwendungsobjekte auf dem Bildschirm und nicht die Manipulationsmöglichkeit, dies ist nur bei Systemen mit direkter Manipulation der Fall. Erfolgt eine Manipulation der Bildschirmobjekte, bewirkt das auch eine Änderung der durch sie repräsentierten Objekte der Anwendung. Wird ein Symbol für ein Dokument kopiert, muß auch die zugehörige interne Datenstruktur File kopiert werden. Anwendungen, die als metaphorische Dialoge repräsentiert werden können, sind neben Schreibtischoberflächen vor allem Editoren für verschiedenen Einsatzzwecke. Anwendungen reichen vom Schaltungsentwurf (vergl. Abb. 4.1) über die Komposition von Musikstücken bis natürlich zu vielen Arten von Editoren.

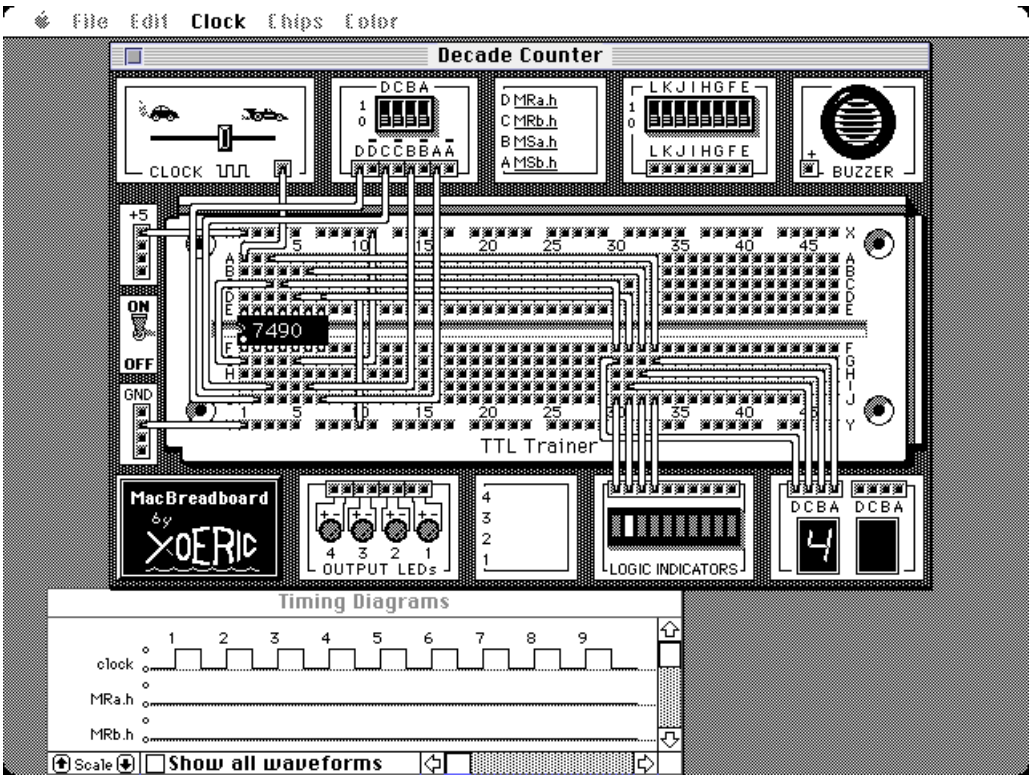

Abbildung 4.1: Die Benutzeroberfläche eines TTL Simulators. MacBreadboard, YOERIC Software 1993. An diesem Editor wird besonders deutlich sichtbar, wie ein Bereich der Anwendungswelt modelliert wird, indem die Objekte der Anwendung auf dem Bildschirm repräsentiert werden. Der Benutzer kann eine TTL Simultion direkt manipulativ erstellen und austesten. Aus realen Elektroniklabors stammendes Wissen kann direkt zur Bedienung der Computeranwendung benutzt werden.

Kennzeichen dieser Anwendungen ist eine stark objektorientierte Arbeitsweise. Objekte sind elektronische Bauteile, Musiknoten oder Textbestandteile wie

Buchstaben oder Abschnitte. Die Objekte werden bewegt, verändert, gekoppelt, kopiert oder gelöscht. Nachteile metaphorischer Dialoge sind nach Herczeg, daß für iterative Prozesse, Suche oder ähnliches die Metaphorik nicht mehr ausreicht. Auch haben Benutzer zu hohe Erwartungen an die Abbildungsgenauigkeit socher Metaphern. Daher muß dem Benutzer ein klares Modell des Systems vermittelt werden, das dessen Grenzen erkennen läßt. Herczeg faßt Vor– und Nachteile metaphorischer Systeme wie folgt zusammen:

Vorteile metaphorischer Dialoge sind:

- kurze Einlernzeit
- Transfer von Wissen aus anderen Gebieten
- hohe Akzeptanz bei Benutzern

Nachteile sind dagegen:

- die Bearbeitung großer Objektmengen ist aufwendig
- Beschreibung von Kontrollstrukturen wie Iteration oder Suche kaum möglich
- die Benutzer überschätzen die Fähigkeiten der Systeme

Bei einem graphischen Editor zur Erstellung von Multimediapräsentationen hält sich der Zahl der erstellten Objekte in Grenzen, auch treten die anderen Probleme nicht auf. Es spricht also alles dafür, für den Editor einen metaphorischen Dialog zu verwenden. Das Problem ist dabei aber, eine passende Metaphorik zu finden. Multimediapräsentationen wurden erst durch moderne Computertechnik möglich und daher existieren im gewohnten beruflichen Umfeld des Menschen keine Beispiele, die für einen Präsentationseditor auf dem Rechner metaphorisch dargestellt werden können. Eine wichtige Aufgabe bei der Entwicklung von Autorensystemen ist daher, geeignete Metaphern zu erschaffen. Eine Möglichkeit ist dabei, die Vorgehensweise bei der Erstellung von anderen Medien zu übertragen. Hier bieten sich z.B. die aus der Produktion von Kinofilmen bekannten Storyboards (vergl. Abschnitt 3.4) an.

Fast immer verwenden metaphorische Systeme auch direkte Manipulation, die im folgenden Abschnitt beschrieben wird.

4.4 Direkte Manipulation

Computeranwendungen wie Text– und Graphikeditoren, Tabellenkalkulationen, Computerspiele und Desktopsysteme visualisieren Objekte der Anwendung auf natürliche, erwartungskonforme Art und bieten einfach anzuwendende Möglichkeiten der direkten Manipulation dieser Objekte an. Shneiderman bemerkt, daß

Benutzer solche Systeme gern benutzen, weil sie das System gut verstehen und im Griff haben, es leicht lernen können und zuversichtlich sind, die Bedienung des Systemes nicht zu vergessen ([Shneiderman 1983]). Diese Qualität von Systemen mit direkter Manipulation beruht auf drei wesentlichen Eigenschaften ([Shneiderman 1992]):

- Objekte und Werkzeuge der Anwendung werden ständig visualisiert.
- Die Bedienung des Systems erfolgt nicht über eine komplexe Kommandosprache, sondern über physische Aktionen wie Bewegungen der Maus oder des Joysticks oder über beschriftete Funktionstasten.
- Es gibt schnelle, inkrementielle und reversible Aktionen, deren Auswirkung auf die bearbeiteten Objekte sofort sichtbar ist.

Bei Systemen mit Kommandosprachen sind sich Benutzer häufig unsicher, ob ihre Aktion den gewünschten Erfolg hatte. So ist oft zu beobachten, daß sich Benutzer nach Kopieraktionen den Inhalt des Zielverzeichnisses anzeigen lassen, um zu prüfen, ob die Daten tatsächlich kopiert wurden. Bei direkt manipulativen Systemen dagegen hat der Benutzer das Gefühl, die Kopieroperation durch die Mausbewegung aktiv durchzuführen und sieht, wie das Dateisymbol zum Ziel geführt wird (vergl. Abb. 4.2).

Herczeg ([Herczeg 1994]) führt folgende Vorteile von direkt–manipulativen Systemen auf:

- Sie sind schnell erlernbar, oft genügt eine Vorführung durch erfahrene Benutzer.
- Gelegenheitsnutzer können sich die grundlegenden Bedienungselemente gut merken.
- Fehlermeldungen werden nur selten benötigt.
- Es ist sofort ersichtlich, ob die gerade ausgeführte Benutzeraktion ein Schritt zum gewünschten Ziel war, falls nicht, kann die Aktion leicht rückgängig gemacht werden und ein anderer Weg zum Ziel eingeschlagen werden.
- Dadurch, daß Aktionen leicht rückgängig gemacht werden können, arbeiten die Benutzer mit weniger Angst vor Fehlern und sind eher bereit zu experimentieren.
- Benutzer lösen Aktionen aus, die sie steuern und deren Resultate sie voraussehen können.

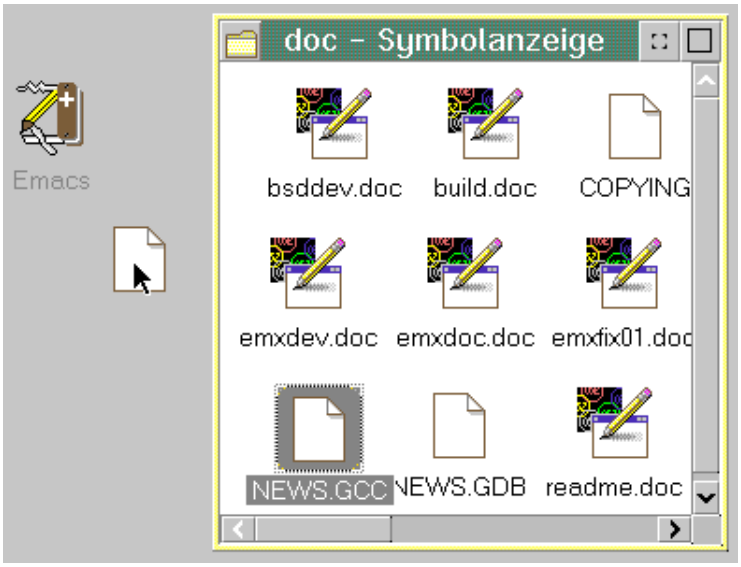

Abbildung 4.2: Ein Beispiel für Drag and Drop. Durch direkte Manipulation mit Drag and Drop hat der Benutzer bessere Kontrolle über das System und es wird einfacher bedienbar. Hier wird das der Datei NEWS.GCC zugeordnete Icon mit der Maus auf das Taschenmesser–Icon des Editors Emacs gezogen, der dann gestartet wird und automatisch diese Datei lädt. Das Dateiicon bewegt sich synchron zur Maus mit. (Workplace Shell des Betriebssystems OS/2 Warp, IBM 1994)

Wichtig bei der Erstellung von Präsentationen im betrieblichen Umfeld ist, daß keine Programmierkenntnisse erforderlich sind und Sachbearbeiter Präsentationen ohne lange Einarbeitungszeit so leicht wie einen Text erstellen können. Wünschenswert ist auch, daß Autoren beim Eingeben der Präsentation experimentieren können, um Wirkungen von Änderungen leicht ausprobieren zu können.

Da es unterschiedliche Ausprägungen von direkt manipulativen Systemen gibt und daher keine charakteristische, genau definierbare Interaktionsform angegeben werden kann, gibt Herczeg zwei psychologische Aspekte für die Systemqualität an, nämlich die Direktheit der Interaktion und die Einbezogenheit des Benutzers in die visualisierte Anwendungswelt. Die **Direktheit** der Interaktion ist umso größer, je geringer die Distanz zwischen dem mentalen Modell des Benutzers über die Anwendungswelt und der im Rechner repräsentierten Welt ist. Diese Distanz muß durch einen mentalen Transformationsaufwand überwunden werden.

Einbezogenheit ist ein Maß dafür, wie unmittelbar der Benutzer auf im Weltmodell visualisierte Objekte ohne eine trennende Zwischensprache einwirken kann. Das Weltmodell liegt direkt manipulativen Systemen zugrunde. Im **Weltmodell** wird die Anwendungswelt visualisiert und Operatoren zur Manipulation der Objekte der Modellwelt bereitgestellt. Bei diesem Modell wird dem Benutzer ein Gefühl der Einbezogenheit in die Anwendungswelt gegeben. Das **Konversa-**

tionsmodell dagegen betrachtet System und Benutzer als Konversationspartner, die mittels einer Sprache kommunizieren. Trägermedium des Dialogs ist eine textbasierte Sprache wie z.B. die Eingabesprache von textorientierten Benutzershells. Der Benutzer kann damit nicht direkt auf die Anwendungsobjekte einwirken, sondern muß seine Wünsche als Befehl an das System formulieren. Eine Einordnung verschiedener Systeme nach den Kriterien Direktheit und Einbezogenheit findet sich in Abbildung 4.3.

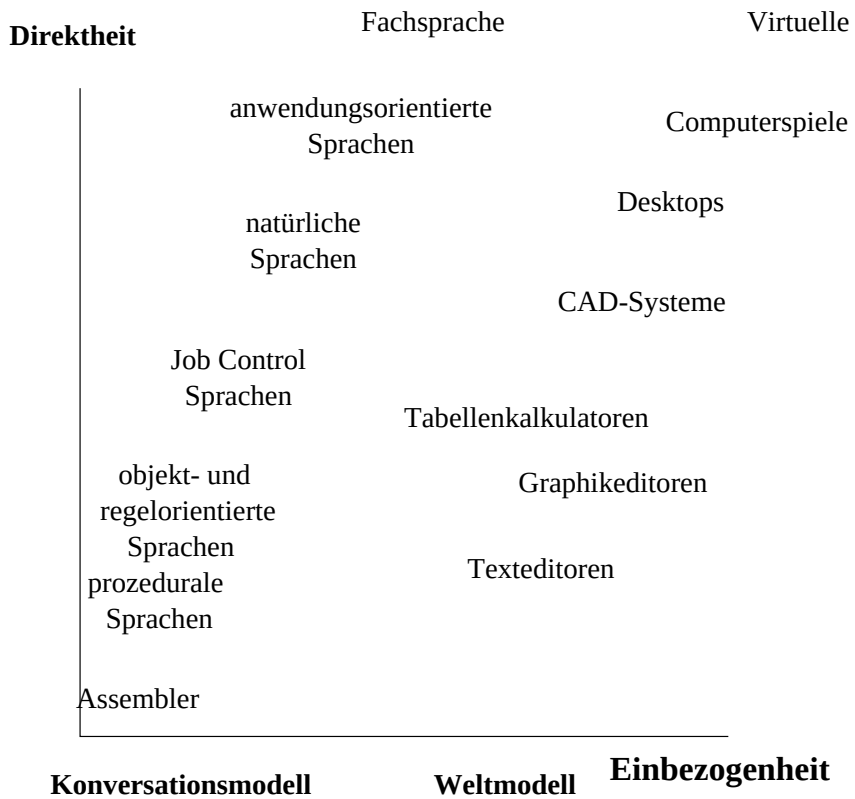

Abbildung 4.3: Einordnung nach den Dimensionen Distanz und Einbezogenheit (aus [Herczeg 94]). Der Präsentationseditor würde wie ein Graphikeditor eingeordnet werden.

Direkt manipulative Systeme sind jedoch nicht immer vorteilhaft. Bei größeren Objektmengen wird die direkt–manipulative Bearbeitung von Objekten viel zu umständlich. In Textshells können tausend Backupdateien mit einem Befehl `rm *.bak` gelöscht werden. Tausend Dateiicons zu markieren und mit der Maus auf den Papierkorb zu ziehen, erfordert dagegen einen gewissen Arbeitsaufwand. Ein nicht zu vernachlässigendes Kriterium beim Design von Benutzerschnittstellen ist daher auch die Effizienz der Interaktion. Herczeg ([Herczeg 1994]) fordert daher eine sorgfältige Aufgabenanalyse, um zu entscheiden, ob direkte Manipulation oder eine Kommandosprache eingesetzt werden soll. Für den in der vorliegenden Arbeit erstellten Editor VisEd ist direkte Manipulation gut geeignet, da beim

Erstellen von Präsentationen keine großen Objektmengen oder sich immer wiederholende Tätigkeiten auftreten.

Ein Problem beim Entwurf von Software mit Dialogschnittstellen ist auch, wie das Zusammenspiel zwischen den Programmteilen, die den eigentlichen Aufgabenbereich der Software erledigen, und der Benutzeroberfläche modelliert wird. Das im nächsten Abschnitt vorgestellte MVC–Paradigma bietet eine Lösung.

4.5 Das MVC–Paradigma

Frühe Computeranwendungen enthielten Programmteile, die Hardware und externe Speichermedien direkt programmierten. Natürlich war die Erstellung dieser Teile aufwendig und fehleranfällig. Jedes Anwenderprogramm mußte diese Programmteile enthalten. Ein wichtiger Fortschritt in der Informatik war daher die Entwicklung von Betriebssystemen, die dafür sorgten, daß Anwendungen nicht mehr direkt die Hardware ansprechen mußten. Dadurch wurde eine Abstrahierung von der zugrundeliegenden Hardware erreicht.

Eine ähnliche Situation lag wieder mit dem Aufkommen von graphischen Benutzeroberflächen vor. Anwendersoftware mußte direkt die Graphikhardware programmieren und Code für die Generierung von Benutzerschnittstellen enthalten. Dies brachte erhebliche Portabilitätsprobleme mit sich, die noch durch die schnellen technischen Fortschritte im Bereich der Graphikhardware verschärft wurden. Eine Abstrahierung von der Graphikhardware wurde durch die Einführung von Graphik– und Fenstersystemen erreicht. Fenstersysteme wie das X Window System ([Scheifler et al. 1988]) führen eine Zwischenschicht zwischen Anwendung und Graphikhardware ein, indem sie Primitive zur Erzeugung von Graphikobjekten, Fonts und rechteckigen Bildschirmteilbereichen (Fenster) zur Verfügung stellen. An neue Graphikhardware muß jetzt nur noch das Fenstersystem angepaßt werden, jedoch nicht mehr die Gesamtheit aller Anwendungsprogramme, welche dieses Fenstersystem benutzen.

Für den Aufbau von Benutzeroberflächen in Objectworks \\ Smalltalk wurde zur Trennung von Applikationslogik und Benutzeroberfläche das Model–View–Controller Konzept (MVC–Paradigma) entwickelt ([Bücker et al. 1993]). Die Anwendung besteht dabei aus drei logischen Teilen:

- Das **Model** repräsentiert Objekte der Realwelt, d.h. Objekte, die durch die Anwendung repräsentiert werden.

- Die **View** ist die graphische Darstellung eines Models. Eine View braucht nur die für eine graphische Repräsentation notwendigen Aspekte des Models zu kennen und darzustellen. Außerdem kann ein Model mehrere Views haben. Z.B. ist ein Radiobutton die View einer Variable eines Aufzählungstyps.

- Der **Controller** ist ein Steuerobjekt, das einer View zugeordnet ist und Benutzereingaben über Tastatur und Maus in Nachrichten an das Model der View umsetzt. Views, die keinen Benutzerfeedback empfangen, benötigen keinen Controller.

Zwischen Model, View und Controller findet ein Fluß von Nachrichten statt, der in Abbildung 4.4 dargestellt ist.

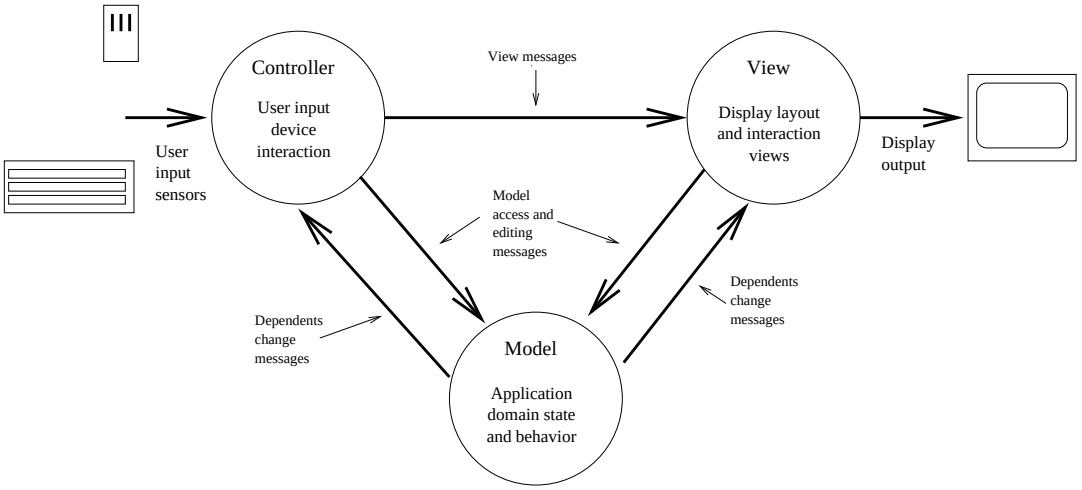

Abbildung 4.4: Nachrichtenfluß im MVC–Paradigma (aus [Krasner, Pope 1988])

Die Vorteile dieser Aufteilung sind, daß Modelobjekte unabhängig von einer Benutzeroberfläche entworfen und implementiert werden können. Auch können verschiedene Benutzeroberflächen verwendet werden, ohne daß die Applikationslogik davon berührt wird. Es können ein anderes Fenstersystem oder auch nur eine andere Art der Präsentation der Benutzerinterfaceobjekte eingesetzt werden.

Gemäß dem MVC–Paradigma (vergl. Abschnitt 4.5) existiert eine logische und programmtechnische Trennung von internen Datenstrukturen und deren Visualisierung auf der Benutzeroberfläche.

Beim in der vorliegenden Arbeit erstellten graphischen Editor VisEd wurde die Trennung von Applikationsfunktionalität und Benutzeroberfläche nach dem MVC–Paradigma vorgenommen. Der Model–Teil entspricht internen Syntaxbäumen und anderen Datenstrukturen zur Verwaltung der visuellen Sprache, die View ist die Visualisierung dieser Objekte auf dem Bildschirm. Jedoch wurde das MVC–Modell vereinfacht und auf die Trennung von View und Controller verzichtet.

4.6 Anwendung der Software–Ergonomie für den graphischen Editor VisEd

Der graphische Editor VisEd soll für die Erstellung von Präsentationen zur Produktdokumentation im betrieblichen Umfeld verwendet werden können. Hierbei ist zu beachten, daß Sachbearbeiter mit diesem Werkzeug arbeiten können sollten, die nur Gelegenheitsnutzer sind. Daher ist die Bedienung des Editors möglichst einfach zu halten, um eine schnelle Einarbeitung zu ermöglichen. Auch ist damit zu rechnen, daß Gelegenheitsnutzer, wenn sie längere Zeit nicht mehr mit dem Editor gearbeitet haben, viele Bedienungsdetails vergessen. Systeme mit direkter Manipulation sind daher für solche Benutzer sehr gut geeignet. Für VisEd wurde eine ähnliche Oberfläche wie bei einem Graphikeditor entwickelt. In einer Palette werden die vorhandenen Sprachmittel durch Icons repräsentiert, die mit der Maus auf eine Arbeitsfläche gezogen werden. Dort werden die Icons in die Ablaufstrukturen eingebunden. Der Verlauf der Präsentation ist durch Linien dargestellt, um eine Metaphorik zu schaffen, die Zeitablauf durch geometrische Anordnung repräsentiert.

Grundlegende Verfahren wie die Manipulation von Objekten mit der Maus werden nicht so schnell vergessen. Der Editor VisEd wurde daher als direkt manipulativ bedienbares System entworfen. Arbeitsschritte können leicht rückgängig gemacht werden, so daß der Autor von Präsentationen einfach eine Variante ausprobieren und auch wieder verwerfen kann.

Kapitel 5

Visuelle Programmierung

Mit dem in der vorliegenden Arbeit erstellten graphischen Editor VisEd können mittels visueller Programmierung Multimediapräsentationen erstellt werden, die auf Rechnern dargestellt werden. Präsentationen sind letztlich Computerprogramme. Daher können Forschungsresultate der visuellen Programmierung auch auf die visuelle Erstellung von Multimediapräsentationen übertragen werden. Schon lange wird versucht, mit visueller Programmierung die Erstellung von Software zu erleichtern. In diesem Kapitel wird vorgestellt, was sich hinter diesem Konzept verbirgt. Anschließend wird auf graphische Notationen, die zur visuellen Darstellung und Spezifikation von Programmen verwendet werden, und ihre Eignung für die Erstellung von Multimediapräsentationen eingegangen. Moderne visuelle Programmierung arbeitet fast immer mit Icons zur Repräsentation von Sprachelementen, daher wird beschrieben, was Icons sind und wie sie entworfen werden sollten. Schließlich werden Systeme zur Erstellung von konventionellen Programmen und von Multimediapräsentationen vorgestellt.

Visuelle Programmierung ist noch ein relativ neues Forschungsgebiet der Informatik. An Literatur sind daher vor allem Forschungsberichte vorhanden. Eine umfassende Sammlung von Veröffentlichungen über Grundlagen und in der Forschung entwickelte Systeme stellte Glinert ([Glinert 1990a, Glinert 1990b]) zusammen. Von Shu ([Shu 1988]) existiert eine Darstellung der Forschungsergebnisse in der visuellen Programmierung in Buchform.

5.1 Gründe für Visuelle Programmierung

Im Zuge der fortschreitenden Verbreitung von Personal Computern und Workstations steigt der Anteil der Rechnernutzer, die über keine oder nur geringe Programmierkenntnisse verfügen. Diese Benutzer setzen daher Standardsoftware ein, die für einen Massenmarkt entwickelt wird und daher nur in geringem Umfang an persönliche Vorlieben und Anforderungen angepaßt werden kann. Der Nutzen eines Computers hängt damit vom Nutzen der eingesetzten Standard-

software ab. Zur Lösung von speziellen Aufgaben mit dem Rechner ist damit Programmierung durch den Endbenutzer notwendig. Programmieren ist aber eine komplexe Aufgabe und erfordert eine lange Lernphase. Das Problem wird noch dadurch verschärft, daß die erstellte Software eine moderne Benutzeroberfläche bieten soll, was aber die Komplexität der Softwareerstellung bedeutend erhöht. Oft übersteigt der Aufwand für die Programmerstellung den durch das erstellte Programm erzielten Nutzen.

Eine Herausforderung für Softwareentwickler ist daher, Systeme zur Verfügung zu stellen, mit denen der Endbenutzer mit geringem Aufwand Applikationen gemäß seinen speziellen Bedürfnissen erstellen kann. Verschiedene kommerzielle Systeme versuchen dies zu erreichen, indem sie Interpreter für mächtige Programmiersprachen enthalten wie z.B. DBase oder MS Word, in der Hoffnung, daß damit der Benutzer seine Softwarebedürfnisse decken kann. Letztendlich ist dabei auch die Eingabe von Quelltext wie in konventionellen Programmiersprachen notwendig. Daher wird versucht, die Erstellung von Programmen auf graphischem Wege mittels visueller Programmierung zu ermöglichen.

5.2 Begriffsbestimmung

Myers ([Myers 1986]) bezieht visuelle Programmierung auf Systeme, die dem Benutzer die Spezifikation auf zwei- oder mehrdimensionale Weise ermöglichen. Dies ist im Gegensatz zur Erstellung von Programmquellcodes in textueller Form zu sehen, die einem eindimensionalen Zeichenstrom entspricht.

Shu ([Shu 1988]) unterscheidet zwei Richtungen visueller Programmierung. Graphische Techniken und Zeigegeräte ermöglichen die Verwendung von visuellen Umgebungen für Programmerstellung und Fehlersuche, für Informationsretrieval und –präsentation und für Softwaredesign. In der anderen Richtung werden Umgebungen entwickelt, die erlauben, mit visuellen Ausdrücken Programme zu erstellen. Meyers dagegen trennt den Bereich der Programmvisualisierung von der visuellen Programmierung ab, da in der visuellen Programmierung die Graphik das Programm selbst ist, bei der Programmvisualisierung das Programm aber in Textform dargestellt wird und Graphik nur dazu verwendet wird, Aspekte des Programms oder seines Ausführungsverhaltens zu visualisieren. Diese engere Definition von Meyers entspricht bei Shu dem Begriff visuelle Programmiersprache. Shu definiert eine visuelle Programmiersprache als Sprache, die visuelle Repräsentation (evtl. unter Hinzufügung von Wörtern oder Zahlen) verwendet, um zu spezifizieren, was sonst in einer konventionellen Sprache erstellt werden müßte.

Hirakawa und Ichikawa ([Hirakawa, Ichikawa 1994]) benutzen den Begriff Visuelle Sprachen für alle Arten, visuelle Information für Mensch–Computer Interaktion zu benutzen. Dabei eingeschlossen ist die Visualisierung, die benutzt wird, um dem Benutzer Systemzustände mitzuteilen, während visuelle Spezifikation die

Möglichkeit benennt, daß der Benutzer mit der Manipulation visueller Elemente Anforderungen an die Maschine stellt. Visuelle Spezifikation unterteilen Hirakawa und Ichikawa in Visual Shell, worunter alle Systeme mit graphischer Benutzeroberfläche und direkter Manipulation fallen, und in visuelle Programmierung. In einer visual Shell kann der Benutzer auf visuelle Art Aufgaben mit dem Rechner ausführen. Mit visueller Programmierung können dagegen selbst elementare Aufgaben zu mächtigen Verfahren zusammengefaßt werden.

In dieser Arbeit wird der Begriff visuelle Programmierung nach der Definition von Myers verwendet, um mit einer engeren Definition arbeiten zu können.

Shu ([Shu 1988]) nennt folgende Punkte als Motivation für den Übergang von traditionellen Programmiersprachen zu visueller Programmierung

- Menschen ziehen Bilder dem Gesprochenen vor.
- Bilder haben größere Ausdruckskraft als Sprache. Sie können mehr Inhalt auf kürzere Weise übermitteln.
- Bilder erfordern nicht die Kenntnis einer natürlichen Sprache und können von Menschen ohne Fremdsprachenkenntnisse verstanden werden.

Um visuelle Erstellung von Programmen zu ermöglichen, mußten graphische Repräsentationen für Programme entwickelt werden.

5.3 Graphische Repräsentationen

Das visuelle System des Menschen und die Verarbeitung visueller Information im Gehirn sind offenkundig optimiert für multidimensionale Daten (vergl. [Myers 1986]). Eindimensionale textuelle Darstellung von Computerprogrammen nutzt diese Optimierung nicht aus. Eine erste Verbesserung in dieser Hinsicht ist schon durch die Einführung von blockorientierten Sprachen und einem entsprechenden Layout des Programmquellcodes möglich. Ein gut formatiertes Listing ist wesentlich besser verständlich als eine einfache unstrukturierte Aneinanderreihung von Pgrogrammstatements.

Verschiedene graphische Repräsentationen wurden zur Programmvisualisierung entwickelt. Vor allem Datenstrukturen wie Bäume oder Listen können durch graphische Darstellung wesentlich verständlicher werden. Auch zur Visualisierung ganzer Programme wurden verschiedene graphische Repräsentationsmechanismen entwickelt. Die vorliegende Arbeit verwendet graphische Repräsentationen zur Visualisierung bestehender Softwarekonstrukte und in Gegenrichtung, um aus der Visualisierung Programmcode zu erzeugen.

Verschiedene graphische Notationen zur Programmvisualisierung stellte Tripp ([Tripp 1988]) zusammen. Er unterscheidet drei Gruppen graphischer Notationen:

1. Box– und Liniennotation

2. Boxnotation

3. Liniennotation

Boxrepräsentationen bestehen aus zusammenhängenden Boxen, die Text enthalten können. Eine Box ist ein Rechteck oder sonst eine geschlossene geometrische Figur. Eine **Liniennotation** besteht nur aus Linien und enthält keine Boxen. Die Kombination beider Notationen, die **Box und Liniennotation** ist ein Graph, in dem die Boxen auf definierte Weise durch Linien verbunden sind. Ein Vertreter von Boxnotationen sind die bekannten Nassi-Shneiderman Diagramme ([Nassi, Shneiderman 1973]), eine Box– und Liniennotation stellen die schon früh verwendeten Flußdiagramme dar. Reine Liniennotationen werden selten verwendet, Beispiele dafür werden in der Arbeit von Tripp vorgestellt.

In der visuellen Programmierung werden graphische Notationen unter dem Gesichtspunkt der einfachen Programmerstellung auch für den Endbenutzer verwendet. Ursprünglich wurden graphische Notationen für Programmierer entwickelt. Ein Flußdiagramm visualisiert einen Algorithmus besser als primitive Programmnotationen wie Fortran oder gar Assembler. Programmierer entwerfen Flußdiagramme in der Weise, daß sie einfach in eine konventionelle Programmiersprache umkodiert werden können. Nassi und Shneiderman ([Nassi, Shneiderman 1973]) führten jedoch wesentliche Nachteile von Flußdiagrammen an. Wichtige Programmkonstrukte wie Iteration haben keine direkte Entsprechung in Flußdiagrammen und müssen aus Primitiven aufgebaut werden, so daß das Flußdiagramm von vielen Einzelheiten überladen und unübersichtlich wird. Schließlich ermöglichen Flußdiagramme uneingeschränkte Sprünge, was zu Programmen mit Gotos und schließlich zum berüchtigten Spaghetti–Code führt. Nassi und Shneiderman führten eine graphische Notation zur Beschreibung von Programmen ein, welche die Nachteile von Flußdiagrammen beseitigten. Die sogenannten Nassi/Shneiderman Diagramme (NSD) repräsentieren strukturierte Programme. Programmkonstruktionen wie if-then-else und Schleifen werden als verschachtelte Blöcke repräsentiert. Abbildung 5.1 führt einige grundlegende Elemente auf. Abbildung 5.2 zeigt die Unterschiede der Darstellung zwischen strukturiertem Programmcode, Flußdiagrammen und NSDs.

Allerdings muß angemerkt werden, daß für den geübten Programmierer ein Algorithmus in der Notation einer Hochsprache besser und schneller zu verstehen ist als in graphischer Notation. Mit dem Aufkommen strukturierter Programmiersprachen wie Algol oder Pascal entfiel zum Teil die Notwendigkeit der graphischen Programmvisualisierung. Standardwerke wie das Buch von Sedgewich ([Sedgewick 1988]) verwenden nicht zuletzt aus Platzgründen eine moderne Hochsprache wie Pascal, C oder C++ zur Darstellung von Algorithmen. Der Schwerpunkt visueller Programmierung liegt heute darin, Benutzern ohne Programmierkenntnisse eine einfache Programmierung zu ermöglichen. Für diese Zielgruppe wurde auch das in der vorliegenden Arbeit entwickelte System entworfen.

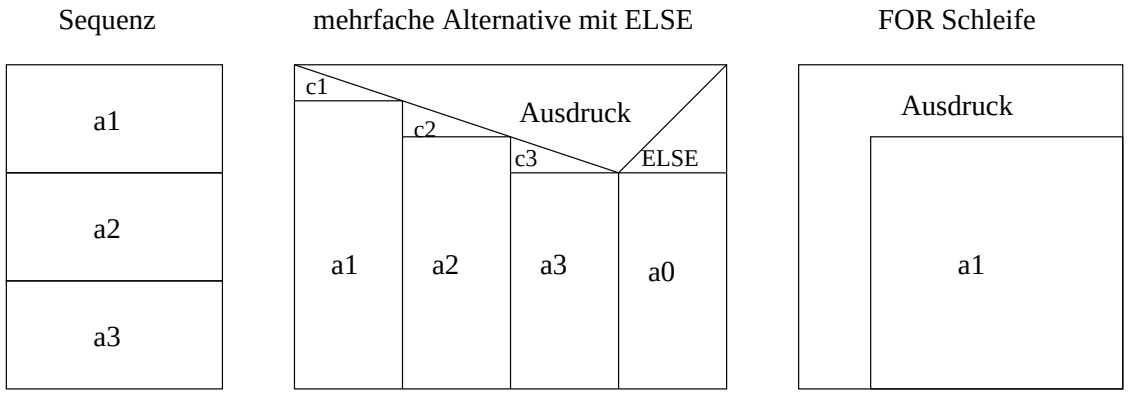

Abbildung 5.1: Grundelemente von Nassi/Shneiderman Struktogrammen

NSD zwingen dazu, nach dem Prinzip des single-entry-single-exit zu spezifizieren. Jeder Block hat genau einen Eingang und Ausgang. Belady et.a. (1980) führen als Nachteile von NSD auf, daß bei größerer Verschachtelungstiefe der Programmstrukturen die Diagramme viel Fläche belegen können. Zudem können die Blöcke unterschiedlich dicht belegt sein. Wenn z.B. der if-Teil einer Bedingung sehr groß ist und der else-Teil nur klein, belegen beide Teile einen Block gleicher Fläche.

NSD sind zur Erstellung von Multimediapräsentationen durch Nichtprogrammierer eher ungeeignet. Die Darstellung ist abstrakt und ist keine Metapher für ein Objekt der realen Welt. Bei Flußdiagrammen können jedoch auch Laien leicht erkennen, daß dadurch der zeitliche Ablauf von Aktionen repräsentiert wird. Flußdiagramme können aber leicht zu unübersichtlicher Darstellung führen. Auch ist keine Abstraktion und Hierarchisierung von Teilmodulen möglich. Daher wurde für die Entwicklung des graphischen Editors VisEd eine Synthese aus beiden Darstellungen vorgenommen. Wie in NSDs setzt sich eine Präsentation aus einzelnen Boxen zusammen. Innerhalb von Boxen wird der Ablaufpfad wie in Flußdiagrammen durch Linien dargestellt. Die Boxen sind jedoch nicht sichtbar und dienen nur für die Anordnung der Ablaufpfade. Statements, also das Darstellen von Monomedien, werden durch Icons repräsentiert, die sich auf den Ablaufpfaden befinden.

Für die Darstellung der Statements müssen daher geeignete aussagekräftige Icons entwickelt werden. Dies ist für die vorliegende prototypische Implementierung noch nicht geschehen.

5.4 Piktogramme – Icons

Ein auffallendes Kennzeichen moderner graphischer Benutzeroberflächen ist die Verwendung von Piktogrammen (Icons). Sie repräsentieren ein Objekt der Anwendung oder des Betriebssystems wie Dateien, Verzeichnisse oder wie in Abb.

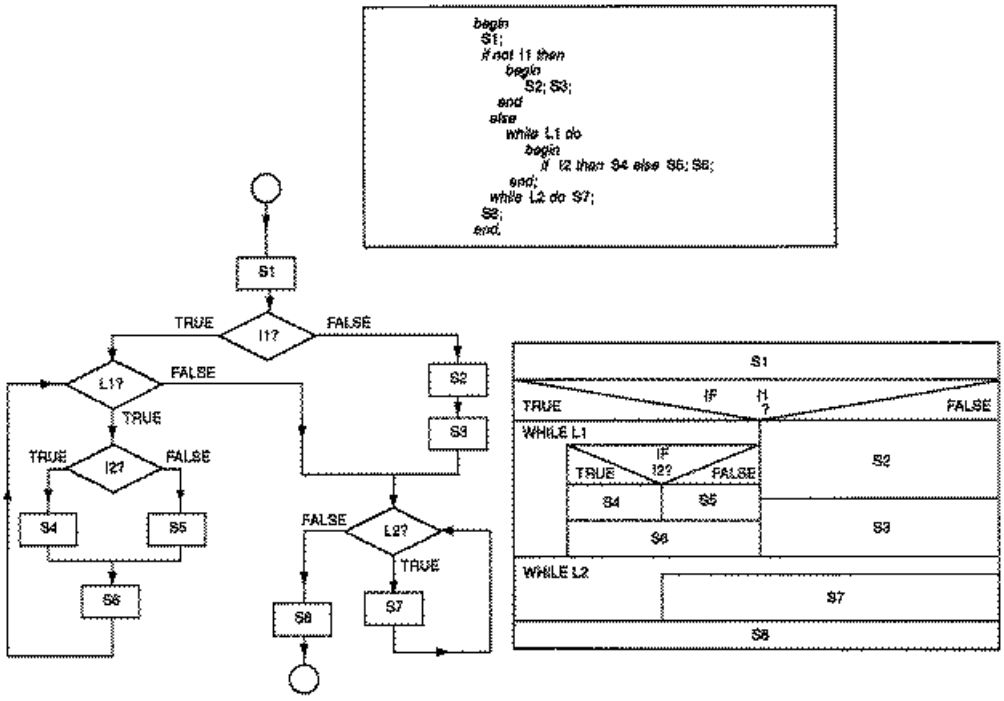

Abbildung 5.2: Vergleich verschiedener Darstellungen. Ein Programm in Pascal–, Flußdiagramm– und Struktogrammdarstellung (aus [Glinert, Tanimoto 1984])

5.3 Möglichkeiten zur Anpassung von Systemkomponenten.

Auch in der visuellen Programmierung werden häufig Icons verwendet, um Elemente der Sprache wie Statements oder Kontrollstrukturen zu repräsentieren. Icons können als Wörter der graphischen Programmiersprache angesehen werden. Dabei ist es wichtig, optisch gut designte Icons zu haben, die dem Benutzer die Repräsentation von Sprachelementen durch Icons gut verständlich machen.

Wood und Wood [Wood, Wood 1987] definieren ein Icon als ein Symbol, (Glyph oder Piktograph), mit dem Begriffe, Ideen oder Objekte repräsentiert werden können. Ein Icon ist in der Informatik also ein graphisches Symbol, das irgendein im Computer modelliertes Objekt repräsentiert. Ein Sanduhrcursor repräsentiert in diesem Sinn den Systemzustand “Computer ist gerade beschäftigt.”

Im Alltagsleben werden Icons weithin verwendet. Beispiele reichen von Symbolen in Wasch– und Bügelhinweisen auf Kleidungsstücken bis zu vielen Arten von “Rauchen verboten” Hinweisschildern. Besonders im Bereich des Straßenverkehrs wird viel Aufwand für die Entwicklung aussagekräftiger Icons verwendet, da von der guten Verständlichkeit von Icons Menschenleben abhängen können.

Wood und Wood führen folgende Kriterien zur Beurteilung der Qualität von Icons auf:

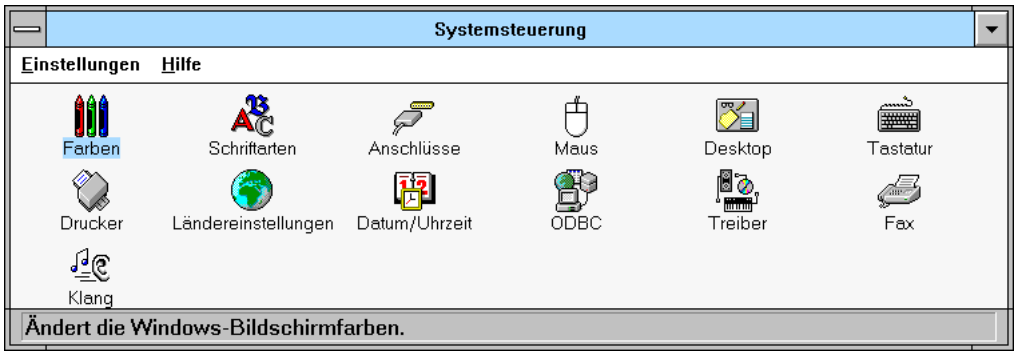

Abbildung 5.3: Icons als Repräsentation von Systemeinstellungsmöglichkeiten (MS Windows for Workgroups, Microsoft 1993)

- Kann das Symbol leicht mit der Aussage verbunden werden?

  Eine Klasse von *repräsentativen Symbolen* ist inhärent verständlich, da sie ein stilisiertes Abbild eines realen Objektes zeigen, wie z.B. ein Druckersymbol. *Abstrakte Symbole* sind oft repräsentative Symbole, die auf ihre graphischen Grundelemente reduziert wurden, oder sie sind willkürliche Graphiken, deren Bedeutung gelernt werden muß, wie z.B. mathemathische Symbole. Die Verständlichkeit von abstrakten Symbolen wächst, wenn sie mit Text kombiniert werden. Die Symbolik des Icons in Abbildung 5.4 ist nicht sofort verständlich, zusammen mit der Beschriftung wird der Sinn sofort

Abbildung 5.4: Ein Texticon

  klar und das Icon wird das nächste Mal leichter wiedererkannt. Willkürlich gewählte Symbole bieten Vorteile, wenn eine Fehlinterpretation gefährlich wäre, wie sie bei repräsentativen Symbolen besteht. Das Verkehrszeichen in Abbildung 5.5 bietet die Möglichkeit von Verwechslungen, das achteckige Stoppzeichen dagegen nicht. So könnte man beim Schild für “Einfahrt verboten” auch Bedeutungen wie Sackgasse, Stop oder “Straße kreuzt” vermuten.

- Ist das Symbol kulturabhängig?

  Das Rote Kreuz Symbol für Krankenhäuser ist für den islamischen Kulturkreis unpassend und wird dort durch einen Halbmond ersetzt.

- Wird das Symbol veralten?

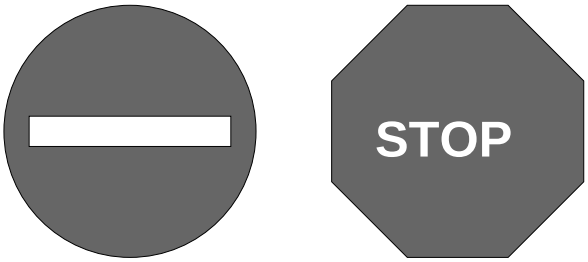

Abbildung 5.5: Verkehrszeichen mit und ohne Verwechslungsmöglichkeiten

Viele Benutzeroberflächen mit Schreibtischmetaphorik verwenden ein Aktensymbol (File), um Dateien zu repräsentieren. Kommt es aber in Zukunft einmal zum papierlosen Büro, ist diese Metaphorik nicht mehr zu verstehen.

- Ist das Symbol ansprechend und unumstritten?

  Ausschließliche Verwendung eines männlichen Umrisses zur Symbolisierung von Menschen sollte vermieden werden.

- Kann das Symbol sinnvoll mit anderen kombiniert werden?

  Einfache und universelle Symbole können zu komplexen Anordnungen zusammengesetzt werden.

- Kann das Symbol in verschiedenen Umgebungen und Situationen verwendet werden?

  Ein Icon sollte bei verschiedenen Bildschirmauflösungen und Farbtiefen verständlich bleiben.

- Kann das Symbol von anderen unterschieden werden?

  Symbole mit unterschiedlicher Semantik sollten auch syntaktisch, d.h. im Aussehen, unterscheidbar sein.

5.5 Existierende Systeme

Im folgenden werden zwei existierende Systeme zur visuellen Programmierung vorgestellt. Als eine relativ frühe Forschungsarbeit in der Entwicklung von Systemen zur visuellen Programmierung wird das Pigs System beschreiben. Es verwendet eine ablauforientierte Darstellung des visuellen Programmes. CleanSheet dagegen, ein modernes kommerzielles System, verwendet eine objektorientierte visuelle Spezifikation. Beide Systeme sind aber nicht für die Erstellung von Multimediapräsentationen geeignet. Dafür geeignete Multimedia-Authoring Systeme werden in Kapitel 6 vorgestellt.

5.5.1 Pigs: ein System mit Nassi/Shneiderman Diagrammen

Pong und Ng ([Pong, Ng 1983]) entwickelten ein System, das NSDs als Eingabeform verwendet und direkt ausführt (Pigs). Ihr Ziel war es dabei, die Programmerstellung für Programmierer effizienter zu machen. Dafür sollten Programmentwicklung, Fehlersuche, Testen, Dokumentation und Pflege innerhalb der gleichen Umgebung möglich sein.

Mit einem graphischen Editor editiert der Benutzer NSDs, die er durch einen Interpreter direkt ausführen kann. Zur Softwaredokumentation muß nun keine graphische Notation mehr verwendet werden, da das Programm schon in dieser Notation eingegeben wurde.

Pong und Ng verzichteten auf die Verwendung eines Compilers, da mit einem eingebauten Interpreter dem Benutzer weitgehende Debugging Möglichkeiten angeboten werden können. So ist es möglich, bei einem Testlauf in der graphischen Notation auf ein beliebiges Konstrukt Breakpoints zu setzen. Auch werden die aktuell ausgeführten Programmkonstrukte markiert.

Der Programmtext in den Diagrammen wird mit einem Recursive-Descent Parser analysiert und dann mit einer Stackmaschine abgearbeitet.

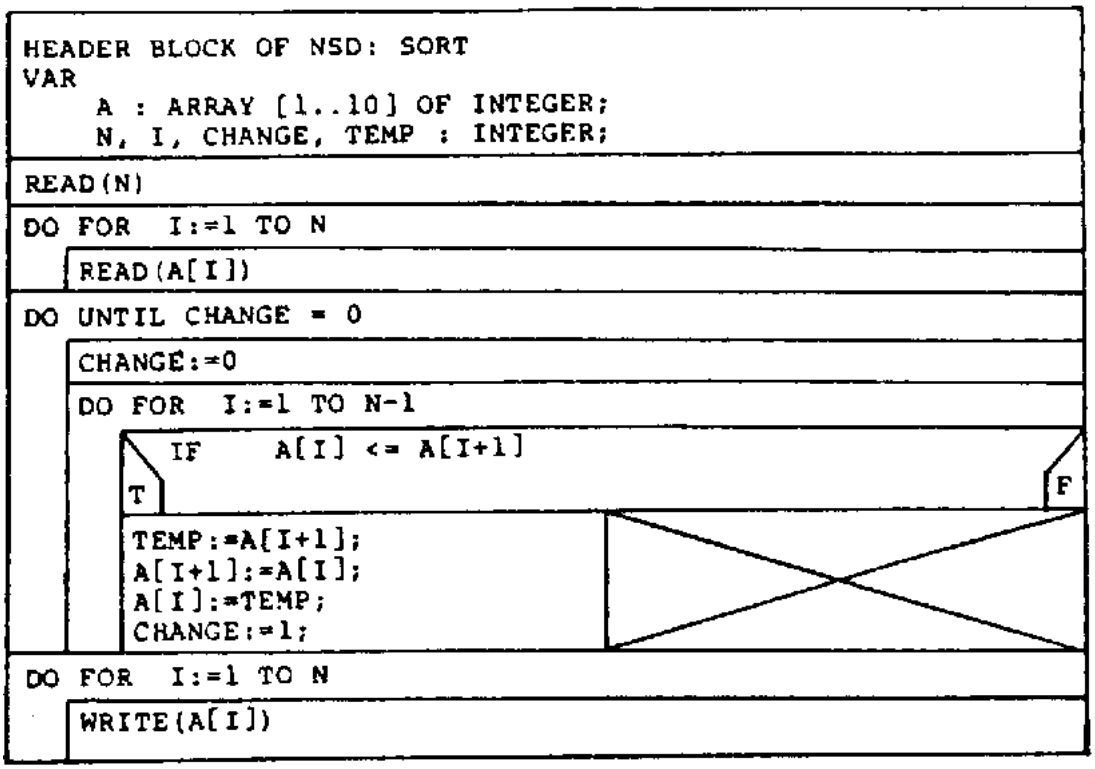

Abbildung 5.6: Ein Pigs–Programm (aus [Pong, Ng 1983])

In die Boxen der Nassi/Shneiderman Notation können dann direkt Programm-

statements eingegeben werden (vergl. Abb. 5.6 für ein Beispielprogramm). Damit zeigt es sich aber, daß auch das System von Pong und Ng nur ansatzweise visuelle Programmierung realisiert, da nur Konstrollstrukturen graphisch dargestellt werden, die Programmstatements aber nach wie vor textuell eingegeben werden.

5.5.2 Das CleanSheet System

Pountain ([Pountain 1994]) beschreibt CleanSheet, ein modernes kommerzielles System, an dem sich der Fortschritt in der visuellen Programmierung seit dem Pigs System zeigt. Um einen kurzen Einblick in neueste Entwicklungen in der visuellen Programmierung zu geben, soll CleanSheet vorgestellt werden, obwohl es nicht in das Design des in der vorliegenden Arbeit entwickelten VisEd Systems einfloß. Moderne visuelle Programmierumgebungen arbeiten vor allem mit Datenfluß zwischen Objekten und damit nicht mehr prozedural–ablauforientiert. Für Multimediapräsentationen ist aber gerade der Ablauf der Darstellung von Medien wichtig. Für Multimediapräsentationen sind daher die Forschungsergebnisse älterer visueller Programmiersysteme eher anwendbar.

CleanSheet erlaubt das Verbinden von Objekten auf dem Schirm, zwischen denen ein Datenfluß besteht. Pountain nennt CleanSheet im Prinzip einen Baukasten für Erwachsene. Ohne eine Zeile Code schreiben zu müssen, sollen so unterschiedliche und komplexe Applikationen wie Tabellenkalkulationen und Visualisierungsprogramme für wissenschaftliche Daten erstellt werden können. Der Baukasten besteht aus Objekten, die Eingabe– und Ausgabeports haben.

Diese Objekte können auf die Arbeitsfläche gelegt und mit Kabeln verbunden werden, so daß eine CleanSheet Applikation wie eine elektrische Schaltung oder ein Plan für eine industrielle Produktionsanlage aussieht (vergl. Abb. 5.7). Durch diese Kabel fließen Daten zwischen Objekten von Ausgabe– zu Eingabeports. Daten werden in Form von Feldern beliebiger Dimension verschickt. Die vorhandenen Objekte gliedern sich in Objekte, die Benutzereingaben akzeptieren, Objekte, die Daten auf dem Bildschirm anzeigen und Objekte, die Daten verarbeiten. Einfache Eingabeobjekte sind z.B. eine Eingabetabelle, in die Werte wie in eine Tabellenkalkulation eingegeben werden können, oder ein Eingabefeld für Text. Als Ausgabeobjekte bietet CleanSheet z.B. Tabellen, Balken– und Tortendiagramme oder 3D-Graphiken. Auch gibt es Objekte, die den Drucker oder eine Datei repräsentieren. Für die Verarbeitung der Daten sind Prozeßobjekte zuständig. Für ein Rechenobjekt kann ein mathematischer Ausdruck angegeben werden, der zur Erzeugung der Ausgabedaten auf die Eingabe angewendet wird. Spezialisierte Prozeßobjekte führen finanzmathematische oder statistische Operationen aus. Schließlich gibt es ein Computerobjekt, in das direkt Statements in einer C–ähnlichen Sprache eingegeben werden können. Auch diese visuelle Programmierumgebung scheint nicht ohne den Notausgang zu konventionellen Programmiermethoden auszukommen. Objekte können zu einem Gruppenobjekt zusammengefaßt werden, so daß Applikationen hierarchisch strukturiert werden

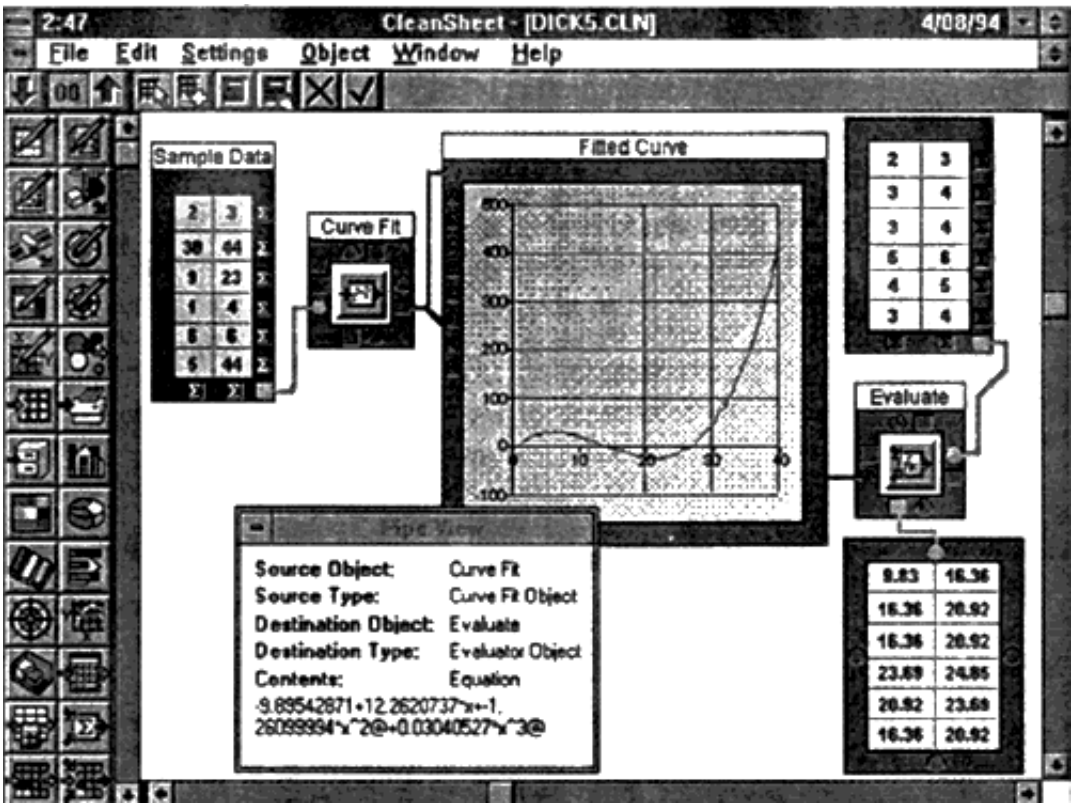

Abbildung 5.7: Eine CleanSheet Applikation (aus [Pountain 1994])

können.

Pountain vermutet, daß Ingenieure, Wissenschaftler oder Finanzmanager mit CleanSheet leichter Aufgaben lösen können als mit konventionellen Tabellenkalkulationen.

Nach dieser Vorstellung von Systemen zur visuellen Programmierung von Standardanwendungen folgen nun Systeme zur visuellen Erstellung von Multimediaanwendungen.

Kapitel 6

Existierende Multimedia-Authoring Systeme

6.1 Das XMAD System

Eirund und Hofmann [Eirund, Hofmann 1993] stellen ein System zur direkt–manipulativen Entwicklung von Hypermediadokumenten vor, das an der Universität Oldenburg entwickelt wurde. XMAD (Multimedia Application Designer) ist zur Erstellung systemkontrollierter Präsentationen von multimedialer Information vorgesehen, die z.B. an Informationskiosken, tutoriellen Systemen oder in einem medizinischen Umfeld eingesetzt werden können. Als Kennzeichen dieser Einsätze von Multimedia führen Eirund und Hofmann an, daß der Benutzer durch eine Menge von Multimediaobjekten navigiert und er an vordefinierten Verzweigungspunkten mit dem System interagieren muß, um den weiteren Weg der Reise durch die Multimediaobjekte auszuwählen. Die Autoren bezeichnen daher solche Multimediapräsentationsprogramme als Navigationsprogramme, die einen oder mehrere alternative Wege durch eine gegebene Präsentation ermöglichen. Daher kann die Funktionalität eines Navigationsprogrammes auf Multimediaobjekten als gerichteter Graph visualisiert werden, in dem die Multimediaobjekte der Präsentation die Knoten definieren und deren Anordnung in den Kanten abgebildet wird. Von einem Startknoten aus können alle anderen Knoten auf einem gerichteten Pfad erreicht werden. In Abb. 6.1 ist ein Ausschnitt aus einer Präsentation und die entsprechende Graphstruktur dargestellt.

XMAD kommt neben Multimediaobjekten mit nur vier Objekten zur Steuerung des Präsentationsablaufes aus. Diese Objekte können bei der Erstellung mit einem graphischen Editor wie auch die Multimediaobjekte direkt manipulativ auf die graphische Visualisierung der Präsentation gelegt werden.

Das **selection Objekt** ermöglicht die Spezifikation einer Programmverzweigung und entscheidet, welcher der vom Selektionsobjekt ausgehenden möglichen Abfolgepfade ausgewählt werden soll. Abb. 6.2 zeigt ein Beispiel für die gra-

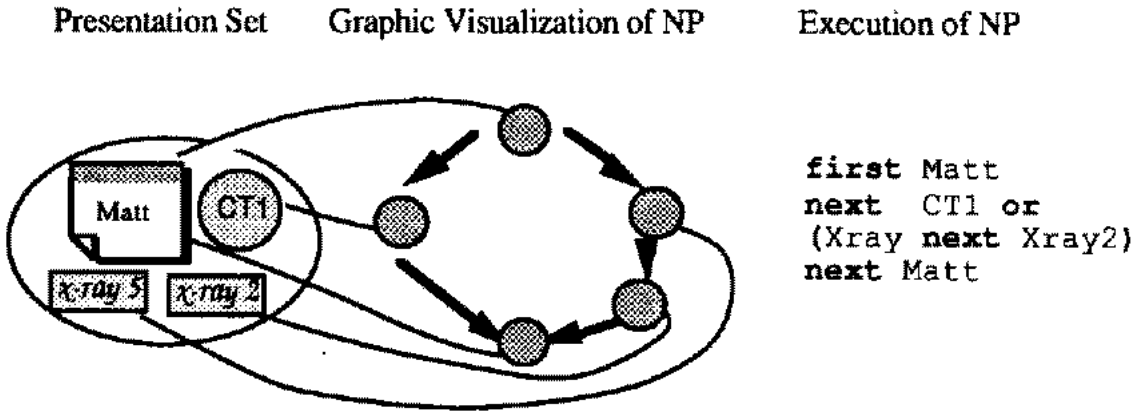

Abbildung 6.1: Eine Präsentation als Graph (aus [Eirund, Hofmann 1993])

phische Darstellung eines Selektionsobjektes. Attribute eines Selektionsobjektes

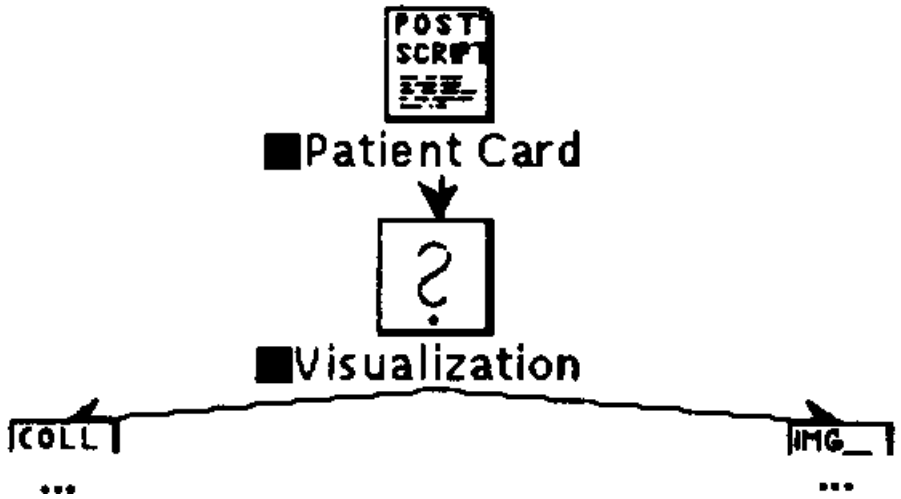

Abbildung 6.2: Ein XMAD Selektionsobjekt (aus [Eirund, Hofmann 1993])

bestimmen, ob die Auswahlentscheidung vom Benutzer in einer Popup Box getroffen oder das System die Entscheidung über den weiteren Pfad treffen soll. Der Pfad kann z.B. anhand der Auswahl des Benutzers in einem vorigen Durchlauf entschieden werden. Auch kann eine mehrmalige Iteration oder ein bestimmter Pfad spezifiert werden. Der Modus des Selektionsobjektes kann in einem Dialog bestimmt werden.

Sollen nach einer Verzweigung alle Zweige parallel ausgeführt werden, müssen die Zweige durch ein **collection** Objekt wieder zusammengeführt werden. Abb. 6.3 zeigt die Spezifizierung der gleichzeitigen Darstellung eines Röntgenbildes und einer Computertomographie. Befindet sich auf einem der Zweige vor einem collection object ein rekursives Objekt (s.u.), so werden alle darin enthaltenen

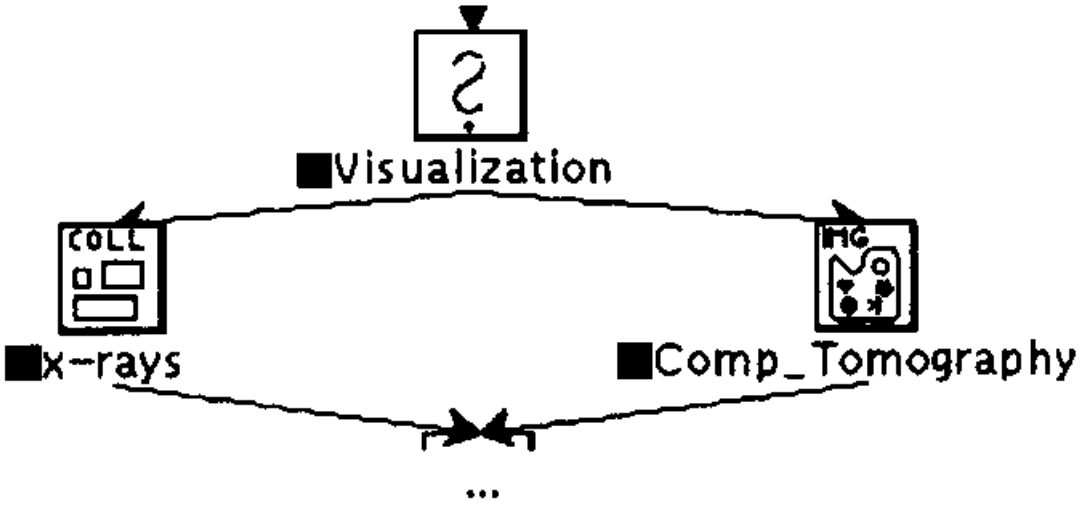

Abbildung 6.3: Spezifikation von Parallelität (aus [Eirund, Hofmann 1993])

Objekte sofort aktiviert. Mit dieser Darstellung von Parallelität kann natürlich keine harte Parallelität wie Lippensynchronität definiert wreden. An beliebiger Stelle im Präsentationsgraphen können **comment** Objekte plaziert werden, die einen Kommentartext enthalten.

Objekte können wieder zu einem Sub-NP-Objekt zusammengefaßt werden. Der Ausführung des compound Objektes entspricht dann die Ausführung der Komponenten.

Mit diesen Primitiven lassen sich mächtige Multimediapräsentationen aufbauen. Jedoch fehlen Objekte zur Konstruktion von Schleifen. Auch wäre es wünschenswert, verschiedene Programmzweige zusammenführen zu können, ohne daß dies eine parallele Ausführung der Zweige impliziert.

Um mit den vorhandenen Präsentationskonstrukten möglichst einfach größere Präsentationen zusammensetzen zu können, bietet XMAD einen graphischen Editor zu Erstellung der Präsentationsnetze an. Direkt manipulativ können Objekte auf eine Arbeitsfläche gezogen werden. Aus dem graphischen Editor heraus können Spezialeditoren zur Manipulation von Präsentationsobjekten aufgerufen werden (vergl. Abb. 6.4).

6.2 MAEstro

MAEstro ([Drapeau 1991, Drapeau 1993]) ist ein Softwarepaket zur Erstellung und Ausführung von Multimediapräsentationen. Eine Vorstellung und Evaluierung des Paketes findet sich bei Schreyjak ([Schreyjak 1994]).

MAEstro wurde dafür entworfen, auch von Präsentationsautoren ohne Programmiererfahrung eingesetzt werden zu können. Daher wurde beim Design Wert auf einfache Bedienung gelegt. Präsentationen sollten interaktiv unter einer gra-

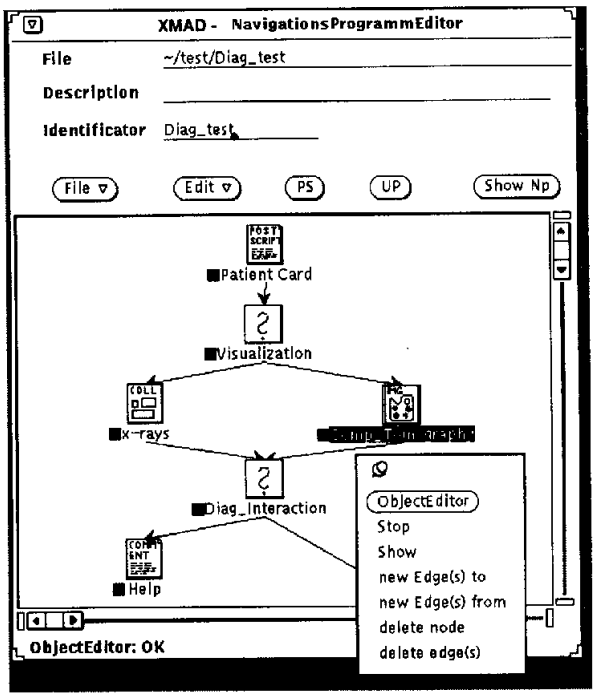

Abbildung 6.4: Der XMAD Präsentationseditor (aus [Eirund, Hofmann 1993])

phischen Benutzerschnittstelle erstellt werden. Weitere Designkriterien umfaßten leichte Erweiterbarkeit zur Einbindung neuer Medien, hohe Portabilität, Fähigkeit zum Einsatz in heterogenen Systemen und die Berücksichtigung von verteilten Medien, um auch über ein Netzwerk Wiedergabegeräte ansteuern zu können. Lippensynchronität wird von MAEstro nicht unterstützt, höhere Synchronitätsanforderungen als Differenzen von einer Sekunde werden nicht erfüllt. MAEstro wurde mit dem Open View Toolkit entwickelt ([Heller 1990]).

Das MAEstro System besteht aus vier Komponenten (vergl. [Schreyjak 1994]).

- **Multimediaeditoren** dienen zur Bearbeitung und zum Abspielen von einzelnen Medien. Z.B. ist in Abbildung 6.5 ein Editor für Tondateien dargestellt. Für jedes unterstützte Medium existiert ein spezieller Editor. Ein Editor muß auch Teilausschnitte des Mediums abspielen können. Diese Ausschnitte werden innerhalb des Editors definiert und verwaltet. Jeder Editor verfügt über eine graphische Oberfläche, über die das Abspielen des Mediums interaktiv gesteuert werden kann. Zusätzlich kann ein Editor über Remote Procedure Calls (RPC) bedient werden, so daß er in eine Multimediaanwendung eingebunden werden kann. Für die Oberflächen wurde eine Consumer-Elektronik Metapher gewählt. Im MAEstro Paket sind u.a. Editoren für Text, Standbilder und Audio enthalten. Mit weiteren Editoren können externe Wiedergabegeräte für Audio CDs, Videos und Laservideodisks gesteuert werden.

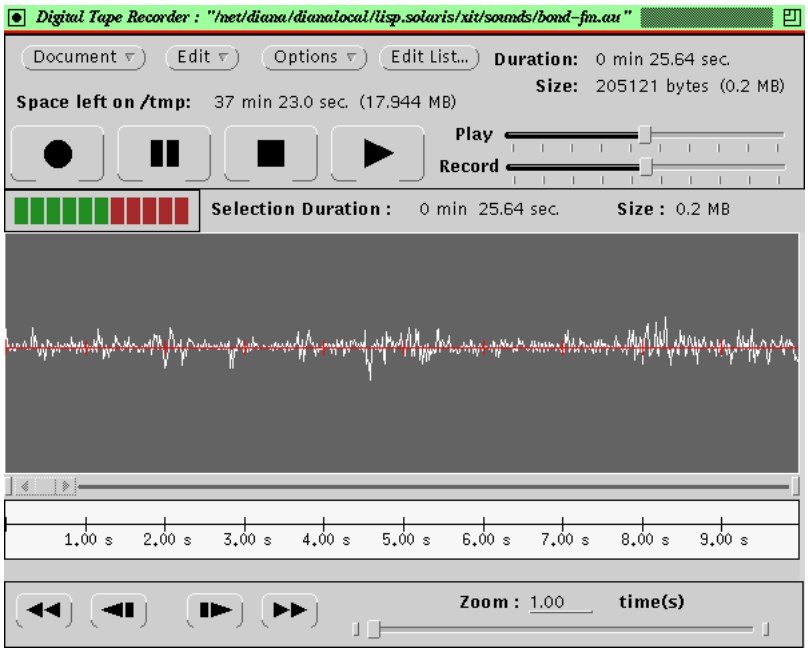

Abbildung 6.5: Der Digital Tape Recorder – ein Medieneditor aus MAEstro

- Mit einem graphischen Editor, **TimeLine**, werden Multimediapräsentation erstellt, die aus einzelnen Mediensegmenten bestehen. Zur Spezifikation der zeitlichen Ablaufsteuerung wird das Zeitachsenmodell verwendet. Die Oberfläche des Editors besteht aus mehreren übereinander angeordneten Zeitachsen (vergl. Abb. 6.6). Jede repräsentiert das Abspielen eines Medi-

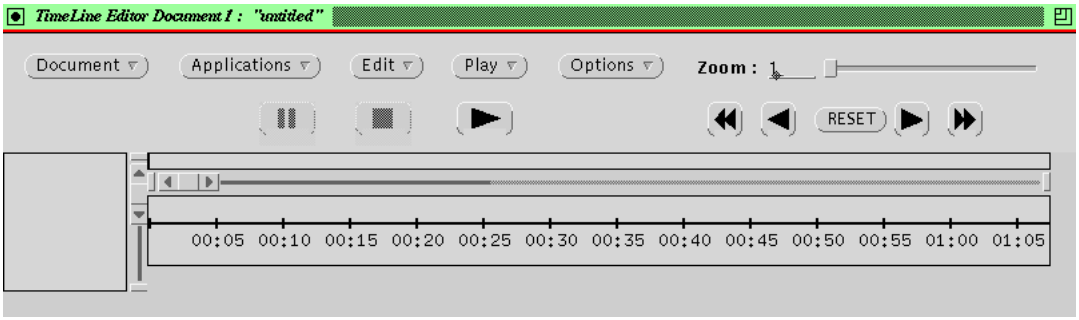

Abbildung 6.6: Der TimeLine Editor – Spezifikation von Multimediapräsentationen nach dem Zeitachsenmodell

ums. Am linken Bildschirmrand werden die Medien durch Icons angedeutet. Ganz unten wird der Zeitmaßstab in Sekunden angegeben. Für Zeiträume, in denen das Medium aktiv ist, wird ein Balken auf die zugehörige Medienzeitachse gelegt. Während der Erstellung der Ablaufdefinition können die einzelnen Medieneditoren so angesteuert werden, daß sie den zugehörigen

Medienausschnitt probeweise abspielen.

- Zur Beschreibung der Kommunikation zwischen dem Präsentationseditor und den Multimediaeditoren existiert ein **Protokoll**, auf dem
- der **Portmanager** aufbaut, der die Dienste der Medieneditoren über das Netz anbietet. Der Portmanager verwaltet eine Liste der auf dem lokalen Rechner verfügbaren Medieneditoren. Diese melden sich über IPC beim Portmanager an. TimeLine erfragt dann vom Portmanager, welche Medieneditoren aktiv sind und kann sie dann über das MAEstro Protokoll ansteuern.

Schreyjak [Schreyjak 1994] schildert Erfahrungen in der Arbeit mit MAEstro. Aufgrund der Verwendung des XView Toolkits für die Oberfläche hält er die Medieneditoren und den Präsentationserstellungseditor für einfach bedienbar. Probleme liegen darin, daß die Bedienung der Medieneditoren nicht einem einheitlichen Stil folgt, so haben die Maustasten unterschiedliche Belegung, was zu erhöhter mentaler Belastung des Benutzers und zu Bedienungsfehlern führt. Ferner besitzen einige Editoren nur eingeschränkte Funktionalität. Daher kommt Schreyjak zum Schluß, MAEstro sei nur als prototypische Entwicklung für ein Präsentationssystem nach dem Zeitachsenmodell geeignet, nicht aber für den praxisnahen Einsatz.

6.3 HyperCard

HyperCard ist eine Hypertext / Hypermediaentwicklungsumgebung für Macintosh Rechner. Da es diesen anfänglich kostenlos beigelegt wurde, erfuhr es eine große Verbreitung und wurde vielfältig eingesetzt. Fisher ([Fisher 1994]) und Gloor ([Gloor 1990]) verwenden HyperCard in ihren Büchern über Hypermedia–Authoring als Autorensystem zur Entwicklung der vorgestellten Beispiele.

Basisobjekte der Informationsdarstellung sind Karten, die zu Stapeln zusammengefaßt werden können. Bezogen auf den Aspekt der Informationsspeicherung entsprechen Stapel in üblicher Informatiksprechweise Dateien, während Karten Records in Dateien entsprechen. Karten wiederum sind in Felder unterteilt. Karten können Information in Form von Text oder Graphik enthalten. Diese Darstellungen können für Animationen auch über den Bildschirm bewegt werden. Auf Karten können sich Buttons befinden, die Sprünge zu einer anderen Karte erlauben. Diese Karte wird dann sichtbar. Kaehler ([Kaehler 1988]) vergleicht diese Informationsdarstellung mit einer großen Wand, auf der viele Zettel mit Informationen angebracht sind. Wären nun Zettel mit ähnlicher Information durch Fäden verbunden, erhält man das Prinzip der Informationsdarstellung durch HyperCard. Z.B. könnten alle Zettel mit Wohnungsangeboten durch Fäden verbunden sein. Natürlich hat Kaehler damit auch das allgemeine Netzwerkmodell

von Hypertext beschrieben. In HyperCard würde ein Zettel einer Karte entsprechen. Karten werden als ein bildschirmgroßes Fenster dargestellt (vergl. Abb. Gloor s78). Mit den Pfeilbuttons, die man in der Abbildung sieht, kann zu anderen Karten fortgeschaltet werden, womit die Funktionalität von Hyperlinks erreicht wird. Informationsdarstellung mit HyperCard kann visuell program-

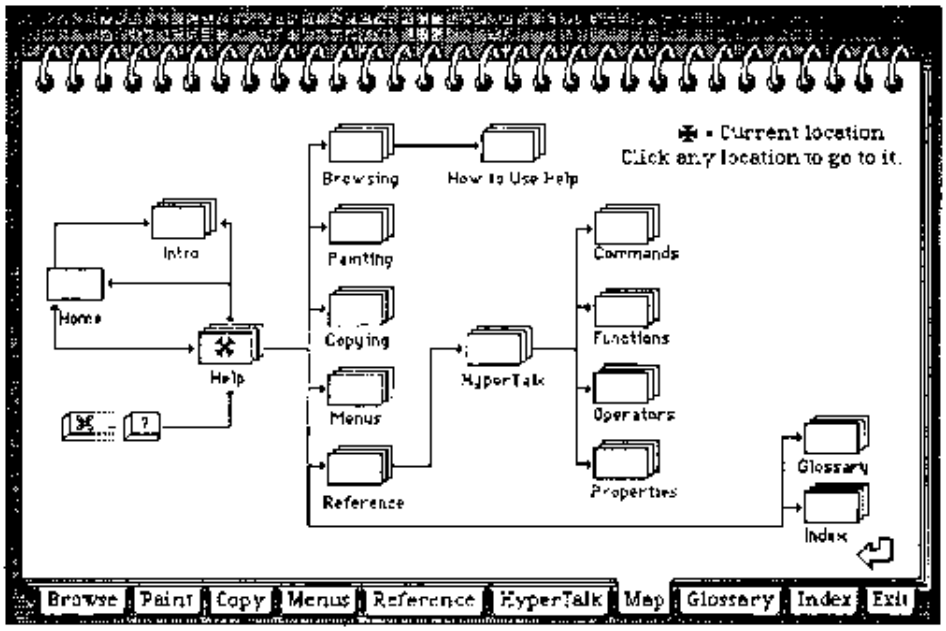

Abbildung 6.7: Ein HyperCard Stapel (aus [Gloor 1990]). Die übereinanderliegenden Karten sind in einer Ringbuchmetapher angeordnet. Durch Anklicken der Zungen kann eine andere Karte nach oben gebracht werden. Ebenfalls kann durch Anklicken der Icons navigiert werden.

miert werden. Dies geschieht durch Kopieren und Ändern bestehender Stapel oder auch nur von Teilen davon wie Karten, Bilder, Felder oder Buttons. Diese Objekte können interaktiv wie in einem Editor manipuliert werden. Neben der Möglichkeit der Erstellung von Präsentationen mit visueller Programmierung kann auch Hypertalk benutzt werden, eine objektorientierte Scriptsprache. Objekte können Nachrichten senden und empfangen. Z.B. können Mausklicks auf Buttons simuliert werden, indem dem Button eine Mausevent geschickt wird:

```
send "mouseUp" to button "start the game"
```

HyperCard war das erste System, das einem größeren Benutzerkreis eine einfache Möglichkeit bot, selbst eine hypermediaartige Präsentation von Informationen zu erstellen. Präsentationen können über Buttons vom Benutzer gesteuert ablaufen oder auch maschinengesteuert mit der Hypertalk Scriptsprache.

6.4 Authorware Professional

Authorware Professional ist eine Programmierumgebung zum visuellen Erstellen von Multimediaanwendungen. Es ist ein sehr ausgereiftes Produkt und wird daher als Referenzprodukt ausführlich vorgestellt.

Der Benutzer erstellt mit Authorware Professional den Ablauf der Multimediapräsentation durch die interaktive Spezifikation eines Flußgraphen. Authorware unterstützt die Verwendung von Text, Graphiken, Video und Sound. Alle Medien werden innerhalb eines Authorwarefensters dargestellt. Parallele Ausgabe von Medien ist nicht möglich, zumal die von Authorware unterstützten Plattformen, der Apple Macintosh und Microsoft Windows, kein echtes Multitasking bieten. Durch fehlende Parallelität entfällt natürlich auch die Problematik der Synchronisierung von Medien.

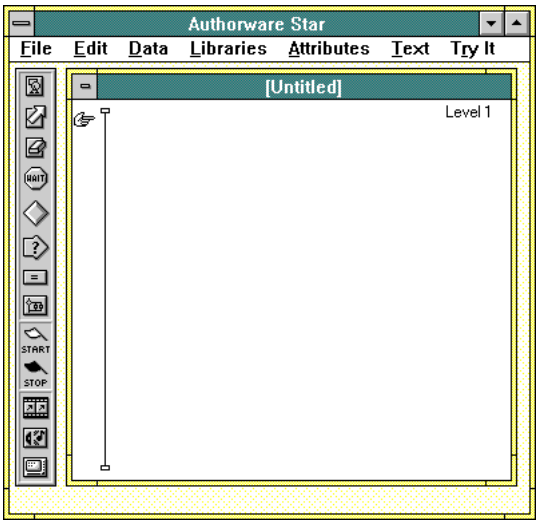

Abbildung 6.8: Die Benutzeroberfläche von Authorware Professional

Ähnlich wie bei einem Zeichenprogramm enthält das Authorware Hauptfenster ein Arbeitsfenster, in dem die Authorware Anwendung erstellt wird. Über dem Arbeitsfenster befindet sich eine Menüleiste mit Kommandos und am linken Bildschirmrand eine Palette von Werkzeugen in Form von Icons (vergl. Abb. 6.8). Die Icons repräsentieren Instruktionen, die Authorware beim Programmablauf ausführt, wie Abspielen von Medien, Verarbeitung von Benutzereingaben oder Programmkontrollstatements. Icons bilden damit zusammen mit den Ablauflinien die Elemente der visuelle Sprache, mit der Präsentationen erstellt werden können. Der Präsentationsablauf wird durch eine vertikal verlaufende Flußlinie dargestellt.

Mittels Drag und Drop zieht der Benutzer Icons auf die Flußlinie. In Abb. 6.9 wurden Icons für die Darstellung von Text/Graphik, Warten, Benutzereingabe und Abspielen von Filmen auf die Flußlinie gezogen. Bei Abarbeitung dieser

Präsentation werden die Icons oben vom Anfang der Linie an abgearbeitet. Alle Ausgaben der Medien erfolgen in einem Präsentationsfenster. Im dargestellten Beispiel wird zuerst eine Graphik ausgegeben, danach hält die Präsentation an, bis der Benutzer durch einen Mausklick das Abspielen des nachfolgenden Films startet. Im Folgenden kann einer von zwei Filmen ausgewählt werden. Die Auswahl kann mit Mausklicks auf zwei Buttons getroffen werden, die dazu erscheinen.

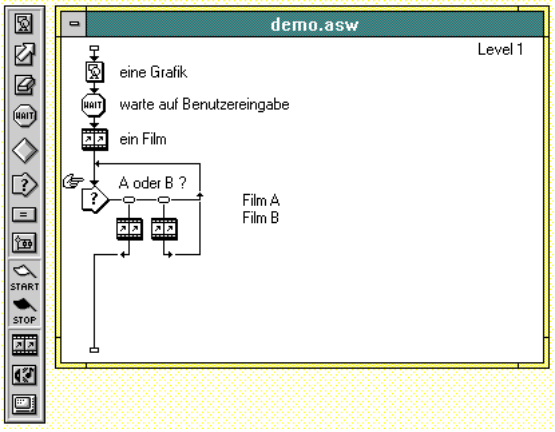

Abbildung 6.9: Programmierung einer Beispielpräsentation

Durch einen Doppelklick auf die Icons erscheint ein Dialogfenster, in dem Attribute für die durch das Icon repräsentierte Interaktion bearbeitet werden können. So lassen sich für Graphiken und Videos Bildübergänge bzw. Bildstandzeiten oder Bilder pro Sekunde festlegen. In Abb. 6.10 ist eine Dialogbox für die Attribute des

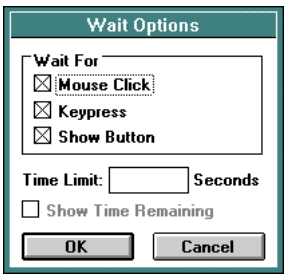

Abbildung 6.10: Wait Options

Wait Icons dargestellt. Zu jedem Icon wird auf dem Schirm eine benutzerdefinierbare textuelle Beschreibung dargestellt. Iconen können verschoben, gelöscht oder in die Zwischenablage kopiert werden. Die Präsentation kann als Ganzes oder in wiederverwertbaren Teilen, sogenannten Models, abgespeichert werden.

Für die Erstellung von intelligenten tutoriellen Systemen läßt sich einfach eine Kontrolle des Nutzerverhaltens durchführen, indem Informationen über Antwortzeiten für Fragen oder Interaktionen oder über die Anzahl korrekter Antworten

gesammelt werden. So läßt sich z.B. spezifizieren, daß das System nach einer bestimmten Anzahl Fehlversuche die richtige Antwort ausgibt.

Innerhalb der Authorware Entwicklungsumgebung ist ein sofortiger Test der erstellten Präsentation möglich. Dabei kann die ganze Präsentation oder ein durch Start- und Stoppflaggen markierter Bereich ausgeführt werden.

Beschreibung der Authorware Icons

Das **Display Icon** repräsentiert die Darstellung von Text oder Graphik. Nach einem Doppelklick auf dieses Icon öffnet sich ein Präsentationsfenster, in dem Text oder Graphik erstellt oder importiert werden können. Beim Abspielen wird der Fensterinhalt dargestellt. In Abbildung 6.11 wurde mit den Werkzeugen von Authorware ein Text mit umgebendem Rahmen erstellt. Die Bitmap links unten wurde mit einem Malprogramm erstellt und über das Clipboard in das Authorware System importiert.

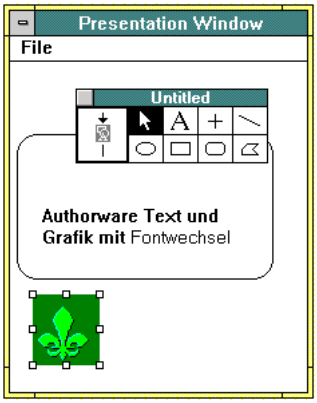

Abbildung 6.11: Graphik- und Texteingabe

Interessante Effekte ergeben sich, wenn eine Bitmap auf dem Bildschirm ihre Position verändert. Diese Aktion wird durch das **Drag Display** Icon repräsentiert. Dabei kann angegeben werden, ob sich die Bitmap zu einem bestimmten Ziel oder auch auf einem bestimmten Pfad bewegen soll.

Das **Erase** Object löscht ein dargestelltes Objekt. Nach Anklicken des Erase Icons fordert eine Dialogbox (vgl. Abb. 6.12) den Benutzer auf, das zu löschende Icon anzuklicken. Für das Löschen stehen verschiedene Effekte zur Verfügung.

Das **Wait** Objekt dient zur Unterbrechung des Programmablaufes. Es kann spezifiziert werden, daß das Programm nach einem Maus- oder Tastenklick oder einem Zeitlimit fortgesetzt wird (vergl. Abb. 6.10).

Das **Decision** Icon dient zur Kontrolle von Programmverzweigungen. Es hat eine ähnliche Semantik wie das User Interaction Icon (s.u.), jedoch erfolgen die Verzweigungen unter Programmkontrolle und nicht unter Kontrolle des Benutzers. Dabei können sequentielle Wiederholungen, zufällige Pfade mit und ohne

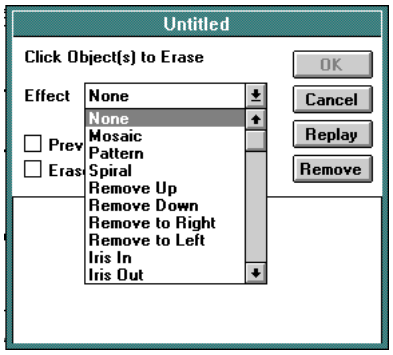

Abbildung 6.12: Löschen von Objekten

Wiederholung oder eine bestimmte Wiederholungszahl angegeben werden.

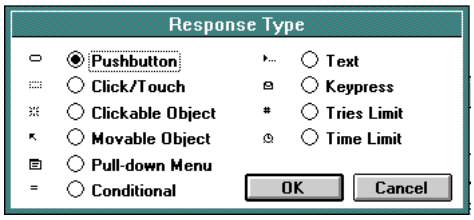

Abbildung 6.13: Möglichkeiten der Benutzerinteraktion

Das **User Interaction** Icon repräsentiert eine Programmverzweigung mit ähnlicher Semantik wie das case Statement in Pascal. Kommt der Präsentationsablauf an die durch das Interaction Icon spezifizierte Stelle, bekommt der Benutzer im Ablauffenster für jeden möglichen Zweig einen Auswahlbutton angeboten und kann sich dann durch Anklicken eines Buttons für einen Zweig entscheiden. Statt der Auswahlbuttons können auch andere Möglichkeiten der Benutzerinteraktion verwendet werden, die im Dialog von Abbildung 6.13 spezifiziert werden können.

Die Spezifikation der möglichen Verzweigungen erfolgt durch Anhängen von Icons an das Interaktions Icon (vergl. Abb. 6.14).

Hierbei sind alle Icons außer dem Decision Icon und einer weiteren Interaction möglich. Letztere ist aber auf dem Umweg über die Spezifikation einer Group möglich, die dann wiederum ein Interaction Icon enthält.

Während des Programmablaufs kann der Benutzer dann, wie in Abbildung 6.15 gezeigt, einen Zweig auswählen. In diesem Beispiel erfolgt die Auswahl für den ersten Zweig durch einen Button und die anderen Zweige über ein Pulldownmenü.

Durch das Anklicken der Verweigungsknoten, welche die Auswahlbuttons bzw. Menüpunkte repräsentieren, öffnet sich ein Dialog, mit dem Attribute der Buttons und damit die Art der Benutzerantwort spezifiziert werden können (vergl. Abb.

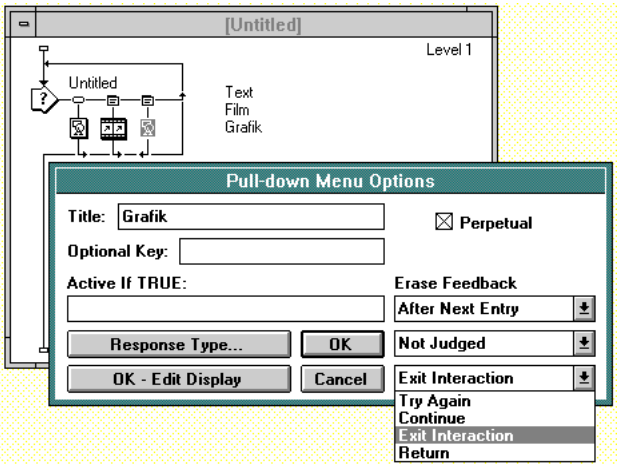

Abbildung 6.14: Auswahl der Benutzerinteraktion

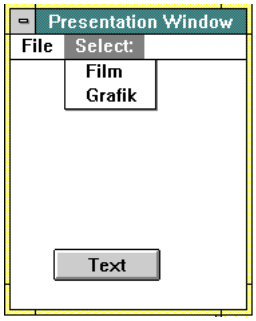

Abbildung 6.15: Ergebnis der Auswahlprogrammierung

6.14). Hier kann auch spezifiziert werden, was nach Abarbeiten des Präsentationszweiges geschehen soll: Try again wiederholt das case Konstrukt, Continue führt den selben Zweig nochmal aus und Exit Interaction verläßt das case Konstrukt.

Ab einer bestimmten Anzahl von angehängten Icons werden aus Platzgründen nur noch die zuletzt angehängten Icons angezeigt und die Fortsetzung nach links durch Punkte angedeutet.

Das **Calculation** Icon steht für textuelle Eingabe von Statements in der Authorware Programmiersprache. Nach Anklicken des Icons öffnet sich ein Quellcodefenster, das editiert werden kann. Hier können wie in einer konventionellen Programmiersprache Funktionen und Variablen verwendet werden.

Das **Map** Icon steht für Compound–Statements und dient damit der Abstrahierung. Nach Anklicken dieses Icons öffnet sich ein weiteres Fenster (vergl. Abb. 6.16), in dem die Statements dieses Teilmoduls spezifiziert werden können. In anderen Statementfenstern können Iconen markiert und in dieses neue Fenster gezogen werden.

Die **Start** und **Stop Flaggen** können auf den Programmablaufgraphen gezogen werden und grenzen einen Teilbereich ein, der damit für Debuggingzwecke

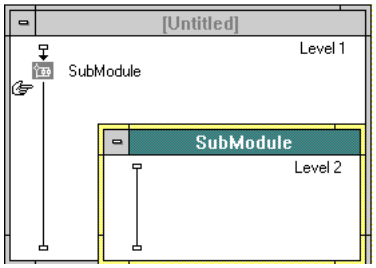

Abbildung 6.16: Ein Submodul

seperat getestet werden kann.

Mit den **Movie** und **Audio** Icons werden abzuspielende Filme, Sounddateien oder Standbilder repräsentiert. In Dialogboxen werden diese Dateien ausgewählt und Parameter für deren Darstellung eingestellt.

Mit dem **Video** Icon wird die Ansteuerung eines externen Video– oder Bildplattenspielers repräsentiert.

Authorware Professional ist ein mächtiges Werkzeug zur Erstellung von Tutoriellen Systemen und Präsentationen mit multimedialen Elementen. Es bietet mächtige und einfach zu bedienende Möglichkeiten der visuellen Programmierung von Applikationen, die mehrere Medien integrieren. Einem Einsatz bei der Produktdokumentation im betrieblichen Umfeld steht entgegen, daß nur ein begrenzter Umfang von Datenformaten unterstützt wird. Hierfür wäre die leichte Einbindung von weiteren Medien und externen Viewern nötig, wie sie im Multimedia Informations System von Bröckel ([Bröckel 1994]) vorgesehen ist. Bei Authorware sind zwar Erweiterungen möglich, aber dafür ist Systemprogrammierung erforderlich (vergl. [Steinbrink 1993]).

Das Konzept, Präsentationen durch eine Flußlinie zu visualisieren, auf der die Darstellung von Monomedien durch Icons visualisiert wird, hat sich als so einfach und so intuitiv bedienbar erwiesen, daß es bei der Entwicklung des graphischen Präsentationseditors VisEd übernommen wurde. Allerdings können Graphen nach dem Flußdiagrammprinzip leicht unübersichtlich werden. Bei Authorware wird dies noch dadurch verschärft, daß im Graphen Text zur Beschreibung der Icons enthalten sein kann. Bei VisEd wurden daher der Flußline Boxstrukturen hinterlegt, die zwar nicht sichtbar sind, dennoch aber für einen strukturierteren Aufbau des Graphen sorgen (vergl. Abschnitte 5.3 und Kapitel 11).

Kapitel 7

Die Programmierumgebung

7.1 X / Motif

Das X Window System ist ein am Massachusetts Institute of Technology (MIT) entwickeltes Standardsoftwaresystem zur Erstellung portabler, graphischer Benutzeroberflächen (vergl. [Scheifler et al. 1988] und [Ruhland 1990]). Wesentliche Charakteristika von X sind:

- X ist freie Software und darf ohne Gebühren kopiert und verwendet werden.
- X stellt nur Grundoperationen zur Erzeugung von Fenstern bereit. Im Unterschied zu anderen, proprietären Fenstersystemen wie dem des Apple Macintosh oder dem Presentation Manager von OS/2 legt X keinen Stil von Fenstern und Bedienung fest. Dies ist eine Angelegenheit der Applikation bzw. des Windowmanagers (s.u.). Auch Fensterteile wie Titelbalken oder andere Dekorationen erzeugt erst der Windowmanager.
- X hat eine geräteunabhängige Architektur. Applikationen können unter X ihre Bildschirmausgabe auf unterschiedlicher Hardware darstellen. Selbst für PCs gibt es unter MS Windows oder OS/2 X Server. Natürlich müssen die Module des X Servers, welche die Graphikhardware programmieren, an die zugrundeliegende Hardware angepaßt werden. Diese Anpassung ist aber für Anwendungen unter X völlig transparent.
- X ist netzwerkfähig. Applikationen sprechen ein Graphikterminal über ein Netzprotokoll an.
- X Anwendungen können untereinander mittels des X Netzwerkprotokolls kommunizieren.
- Der Ablauf von Software erfolgt nicht programmgesteuert, sondern benutzergesteuert, der Programmcode muß auf Benutzereingaben reagieren. Dies entspricht der natürlichen Arbeitsweise des Menschen.

7.1.1 Das Client–Server–Modell

Das X Window System verwendet das Client–Server–Modell. Der Server ist eine Software, die Ausgaben auf dem Graphikbildschirm vornimmt und Eingaben vom Benutzer entgegennimmt. Die Bildschirmausgaben erfolgen in Form von Graphikprimitiven wie Punkte, Kreise oder Textzeichen. In Datenstrukturen des Servers werden Attribute dieser Primitive wie Linienstärke, Farbe oder Font verwaltet. Vor allem kümmert sich der X Server um die Erzeugung und Verwaltung von Fenstern. Eingaben des Benutzers über Tastatur oder Maus nimmt der X Server entgegen und schickt sie an die entsprechenden Clients in Form von Events.

Clients sind Programme, die diese Dienste in Anspruch nehmen, also Ausgaben auf den Graphikbildschirm vornehmen wollen. Die Kommunikation zwischen Clients und Servern geht über ein Netzwerkprotokoll. Das hat zur Konsequenz, daß Server und Client Prozesse des selben Rechners oder auch Prozesse auf weit voneinander entfernten verschiedenen Rechnern sein können. Aus Sicht des Betriebssystems von Client und Server erfolgt eine Ausgabe auf den Grafikschirm wie eine beliebige andere Netzwerkkommunikation wie `telnet` oder `ftp`.

Der Code zum Ansprechen des X Servers über das Netzwerk befindet sich in einer speziellen Library, der `xlib`, die an X Applikationen gebunden wird. Die Kommunikation zwischen Client und Server über das Netzwerk ist schematisch in Abbildung 7.1 dargestellt. Für den Benutzer ist es prinzipiell transparent, ob

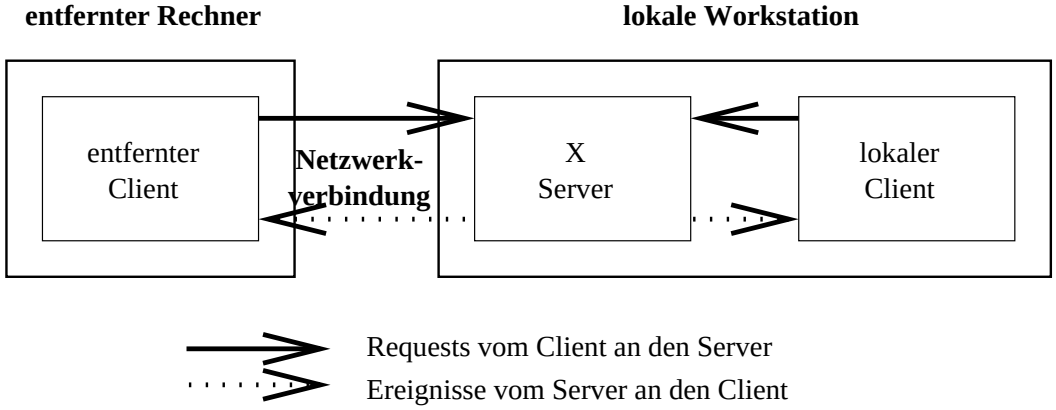

Abbildung 7.1: Lokale und entfernte Netzwerkverbindungen (nach [Ruhland 1990])

das Anwendungsprogramm, mit dem er arbeitet, auf lokalen oder weit entfernten Rechnern läuft. Natürlich können sich Engpässe in der Netzwerkkapazität als Verzögerungen bei Eingabe und Ausgabe bemerkbar machen.

7.1.2 X Fenster

Fenster (Windows), die Grundobjekte der Benutzeroberflächen von Applikationen unter X, sind ein rechteckiger Bereich des Graphikbildschirms. Auf einem Bildschirm können gleichzeitig viele verschiedene Fenster existieren, die sich überlappen oder verdecken dürfen. Fenster können verschiedenen Applikationen gehören, die möglicherweise auf unterschiedlichen Rechnern unter verschiedenen Betriebssystemen laufen. X Fenster stellen einen virtuellen Bildschirm dar, der hierarchisch organisiert wieder Unterfenster enthalten kann. Rollbalken, Menüs oder Buttons sind Beispiele für Unterfenster. Fenster haben im wesentlichen zwei Funktionen: sie gruppieren Dialogelemente nach logischen Zusammenhängen und nehmen Benutzereingaben an. Der X Server sammelt, wie in Abbildung 7.2 dargestellt, Benutzereingaben und gibt sie dann an die Warteschlange eines Cli-

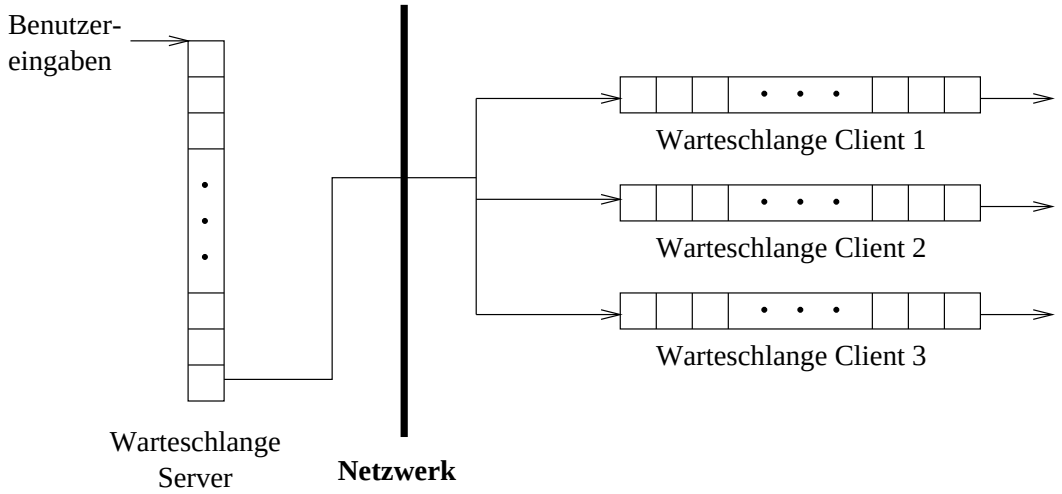

Abbildung 7.2: X Server– und Client–Warteschlangen (aus [Ruhland 1990])

enten weiter. Der Empfänger ist dabei derjenige Client, dem das aktive Fenster gehört. Üblicherweise ist ein Fenster aktiv, wenn es den Mauszeiger enthält.

Fenster und ihre Unterfenster sind in einer Baumhierarchie organisiert (vergl. Abb. 7.3). Das Wurzelfenster (root) ist dabei der gesamte Bildschirm. Fenster können sich nur innerhalb des Vaterfensters befinden. Teile, die darüber hinausragen würden, schneidet der X Server ab. Unterfenster eines Vaterfensters können auch ihre Lage übereinander ändern. Dabei ist nur das oberste Fenster sichtbar.

7.1.3 Programmierschnittstellen zu X

Das X Netzwerkprotokoll besteht aus Requests und Events, die in Netzwerkpaketen abgelegt werden. Die `xlib` stellt C Funktionen zur Verfügung, die diese Requests ausführen und Events abarbeiten. Im wesentlichen haben diese Funktionen die gleiche Mächtigkeit wie das X Protokoll, arbeiten also auch auf der

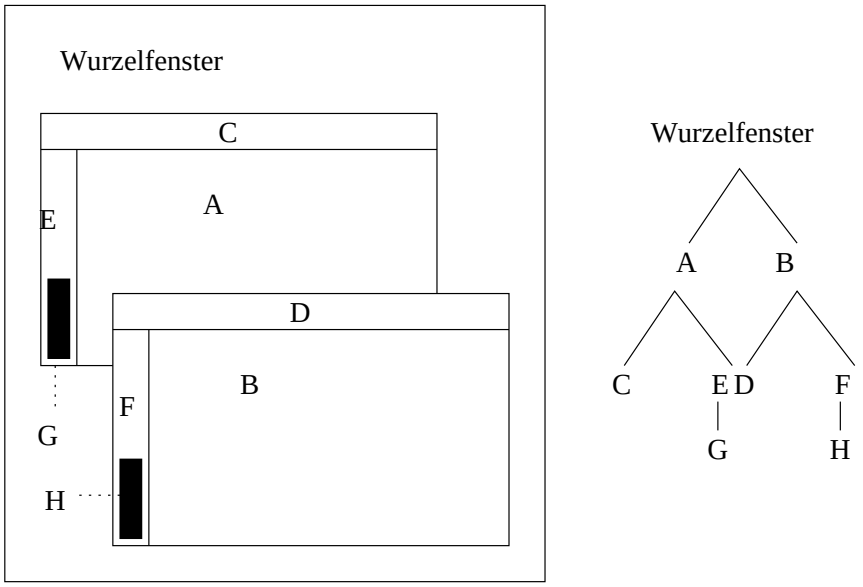

Abbildung 7.3: Eine typische Fensterhierarchie (aus [Ruhland 1990])

Ebene von Graphikprimitiven und Fenstern. Es gibt zwar noch einige sogenannte Convenience Funktionen, die mehrere Requests zusammenfassen, aber dennoch bleibt die Xlib auf so niedriger Ebene, daß Anwendungsprogramme, die nur die Xlib verwenden, einen sehr hohen Programmieraufwand erfordern. Das X Protokoll und damit auch die Xlib bieten keinerlei Dialogobjekte wie Buttons oder Menüs an. Diese müssen bei Verwendung der Xlib vom Anwendungsprogrammierer aus Fenstern und Graphikprimitiven zusammengesetzt werden.

Es wurden jedoch schon früh Toolkits entwickelt, die solche Funktionalität anbieten. Toolkits sind Bibliotheken, die mächtige Dialogobjekte enthalten, die in diesem Zusammenhang `widgets` genannt werden. Toolkits bauen auf der Xlib auf. Beispiele für Toolkits sind der X Toolkit, XView ([Heller 1990]), XIT ([Herczeg et al. 1992]), das auf einer Reimplementation der Xlib in Lisp aufbaut, sowie natürlich Motif ([Gottheil et al. 1992]).

Der X Toolkit besteht aus den Xt Intrinsics und dem Athena Widget Set. Die Xt Intrinsics stellen Grundmechanismen zur Verfügung, mit denen Widgets erzeugt oder bestehende erweitert werden können. Die meisten Toolkits bauen daher auf den Intrinsics auf. Die Athena Widgets sind eine Beispielimplementation für ein Widget Set. Sie stellen aber nur relativ primitive und ästhetisch wenig ansprechende Dialogobjekte zur Verfügung.

Mit der Zeit wurden so viele verschiedene Toolkits entwickelt, daß dadurch eine konsistente Bedienung von Rechnern mit der X Oberfläche verhindert wurde. Auf Apple Macintosh Rechnern legten strenge Designrichtlinien fest, wie Oberflächen auszusehen hatten und zu bedienen sein sollten. Eine Organisation von Workstationherstellern, die Open Software Foundation (OSF), definierte daher

ein einheitliches Look and Feel von X Anwendungen und legte die Definition im Motif Style–Guide fest. Die Motif Spezifikation von Benutzeroberflächen unter X hat sich weitgehend durchgesetzt (vergl. [Beil 1993]). Gemäß dieser Definition wurde der Motif Toolkit in C programmiert. Die Dialogobjekte, die er zur Verfügung stellt, heißen Motif Widgets. Der Motif Toolkit arbeitet auf dem X Toolkit. Die logischen Bestandteile einer Applikation, die auf dem Motif Toolkit aufbaut, sind in Abbildung 7.4 dargestellt. Für die Darstellung von Graphik muß

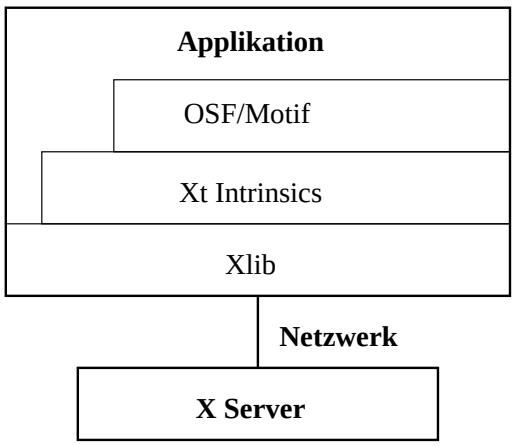

Abbildung 7.4: Architektur einer X Motif Applikation (aus [Ruhland 1990])

die Applikation jedoch weiterhin direkt die Xlib programmieren, da Motif keine Möglichkeiten zur Erzeugung von Graphik anbietet.

7.1.4 Programmieren mit Motif Widgets

Für die Programmierung von Benutzerschnittstellen stellt Motif eine Menge von Dialogobjekten bereit. Sie werden Widgets genannt, ein Kunstwort aus window und gadget (Fenster und Gerät). Solche Dialogobjekte wie z.B. Menüs oder Scrollbars sind Graphikfenster mit bestimmten Attributen und Fähigkeiten. Die Funktionalität zur Manipulation von Widgets wird von den Xt Intrinsics bereitgestellt. Wie alle X Fenster sind auch die für eine Benutzerschnittstelle verwendeten Widgets in einer Baumstruktur angeordnet. An Eltern Widgets werden weitere Widgets angehängt. Widgets besitzen ein Verhalten. Sie haben Methoden, mit denen sie auf Ereignisse reagieren. Wird z.B. ein bisher durch ein anderes Fenster überdeckter Bereich eines Widgets sichtbar, schickt der X Server an dieses Widget ein Expose–Event. In Tabellen im Widget ist festgelegt, wie es auf bestimmte Events reagiert. Im Falle von Expose–Events wäre ein Neuzeichnen des sichtbar gewordenen Fensterteiles nötig. Neben dem vordefinierten Verhalten von Widgets, den Actions, kann der Anwendungsentwickler auch ein spezielles Verhalten von Widgets programmieren. Er kann den Widgets Adressen von Funktionen, sogenannte

Callbacks, übergeben, die aufgerufen werden, falls ein bestimmtes Event eintritt. Die Aufrufkette von Events und Callbacks ist in Abbildung 7.5 dargestellt.

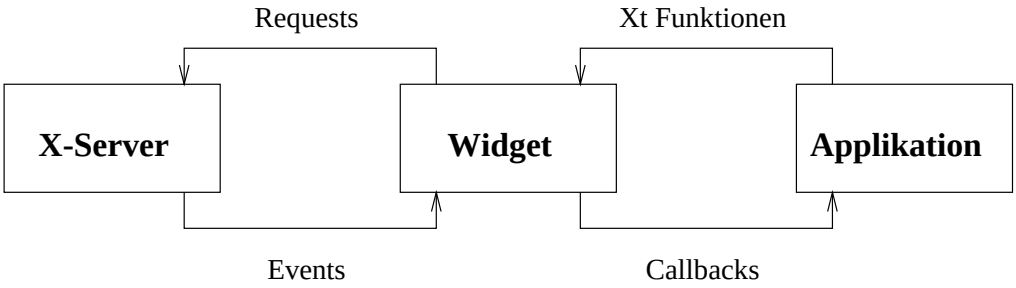

Abbildung 7.5: Ereignisfluß bei Widgets (nach [Beil 1993])

Motif Widgets sind objektorientiert entworfen, jedoch in der prozeduralen Sprache C implementiert. Widgets mit denselben Eigenschaften sind in Klassen zusammengefaßt. Von Widgetklassen gibt es Instanzen, die ihre privaten Daten enthalten. Die Widgetklassen sind in einer Hierarchie organisiert. Mit Instanzen von Klassen aus dieser Hierarchie wird ein Baum von Widgets aufgebaut, welche die Benutzerschnittstelle einer Applikation bilden. Die Wurzel dieses Baumes ist ein `Shell`–Widget, das ein Toplevel X Fenster erzeugt. Häufig enthält dann das Shell–Widget ein `Container`–Widget, das andere Dialogobjekte wie `Pushbuttons`, Menüs oder DrawingAreas enthält, in die mit Xlib Aufrufen Graphik gezeichnet werden kann.

Die Erzeugung von Widgets geschieht in vier Schritten. Zuerst muß der Datenstrukturteil des Widgets mit der `create` Funktion erzeugt werden. Dabei wird das Widget auch in die Hierarchie von Dialogobjekten der Applikation eingebunden. Durch einen Aufruf der `manage` Funktion wird das Widget im Layout des Elternwidget berücksichtigt. Noch existiert das Widget nur als Datenstruktur auf der Seite des X Clienten. Durch Aufruf von `realize` wird auch ein X Fenster auf Serverseite erzeugt. Sichtbar wird das Widget erst durch Aufruf von `map`, aber nur dann, wenn es nicht durch andere Fenster überdeckt wird und wenn auch das Eltern–Widget sichtbar, d.h. gemapped ist.

Um eine Benutzeroberfläche zu erzeugen und dann Benutzereingaben entgegenzunehmen, muß jedes Motif Programm die folgenden Aktionen ausführen:

- Initialisierung der Intrinsics Routinen
- Ermittlung eines Application Context
- Anmelden des Programms beim X Server unter Angabe der Displaynummer
- Öffnen der Toplevel Shell
- Erzeugen des Widget Baumes

- Realisierung des Widgetbaumes, d.h. Darstellung auf dem Schirm
- Bearbeitung der Events in einer Hauptschleife

Mit Motif ist die Programmierung von Oberflächen unter X wesentlich einfacher als bei alleiniger Verwendung der Xlib. Zwar ist Motif objektorientiert entworfen, aber da es in C implementiert ist, können Benutzerobjekte trotz Verwendung von C++ nicht wie Klassen instanziiert wird. Dies wird erst durch Verwendung von in C++ geschriebenen Toolkits möglich, die auf Motif aufbauen.

7.1.5 Die Motif++ und Xm++ Toolkits

An der Universität Lowell wurde Motif++ ([van Loon 1993]) entwickelt, ein Toolkit der Motif Widgets in C++ Klassen kapselt. Jeder Art von Widgets entspricht dabei eine C++ Klasse. Die Klassenhierarchie von Motif++ entspricht damit genau der der Motif Widgets. Alle Ressourcen von Motif Widgets werden durch Klassenmethoden ausgewertet oder gesetzt.

Bei Verwendung von Motif++ müssen nicht mehr wie in Motif C Funktionen zur Erzeugung von Widgets aufgerufen werden. Es genügt, Instanzen von Klassen zu deklarieren. Die Erzeugung und Initialierung des Widgets übernimmt dann der Konstruktor. Callbacks müssen bei Motif++ globale Funktionen sein, das heißt, es können keine Klassenmethoden zur Reaktion auf Events verwendet werden. In Abbildung 7.6 befindet sich ein Codebeispiel für Verwendung von Motif++.

```
#include "XMAppplicationShell.hh"
#include "XMPushB.hh"

main(int argc, char **argv)
{
  XMApplicationShell shell("main",&argc,argv);
  XMPushButton button(&shell,"button",100,30,NULL);

  button.Manage();
  shell.Realize();
  shell.AppMainLoop();
}
```

Abbildung 7.6: Widgets mit Motif++

Ein anderer Toolkit für die Entwicklung von Benutzerschnittstellen in C++, der an der Universität Wien entwickelt wurde, ist Xm++. Er stellt Klassen für häufig gebrauchte Dialogobjekte zur Verfügung. Dies geschieht unter Verwendung eines zugrundeliegenden Toolkits. Bisher werden die Athena und Motif Widgets unterstützt. Nach erneuter Übersetzung und neuem Binden kann eine Applikation im Athena oder Motif Style erscheinen. Xm++ arbeitet jedoch nicht damit, daß es C++ Schnittstellen für diese Widgets bereitstellt. Vielmehr enthält es eine eigene von den zugrundeliegenden Toolkits unabhängige Hierarchie und bildet somit einen Meta–Rahmen zur Erzeugung von Benutzerschnittstellenobjekten. Xm++ stellt auch komplexere Dialogobjekte (vergl. [Strassl, Binder 1994b]) zur Verfügung, die nicht in den Athena oder Motif Widgets vorhanden sind. Dabei werden einfachere Dialogobjekte aus diesen Widgetsets zu mächtigeren kombiniert.

Der Anwendungsprogrammierer arbeitet nur mit den Schnittstellen des Xm++ Toolkit. Wissen über die Toolkits, auf denen Xm++ aufbaut, wird dabei nicht gebraucht. In Abbildung 7.7 findet sich ein kleines Codebeispiel, das eine unter X lauffähige Xm++ Applikation erzeugt, deren Oberfläche in Abbildung 7.8 dargestellt ist.

Der Xm++ Toolkit enthält eine Klasse `XmApp`, deren `initialize` Methode noch nicht definiert ist und daher vom Applikationsentwickler bereitgestellt werden muß. Diese Methode wird von der im Toolkit enthaltenen C++ `main` Funktion aufgerufen und ist der Ausgangspunkt für alle anderen Initialisierungen. Nach deren Abschluß liegt die Kontrolle beim Toolkit, der ereignisgesteuert Callbacks aufruft.

Eine Erweiterung von Xm++, XmCi2, enthält Klassen zur Erzeugung und Verwaltung von Graphikobjekten wie Rechtecken oder Kreisen. Diese Objekte arbeiten wie die aus manchen Toolkits bekannten virtuellen Fenster. Die Graphikobjekte können nur in speziellen Containerfenstern existieren. Diese erkennen, ob z.B. ein Mausklick auf einem Graphikobjekt erfolgte und simulieren dann ein Verhalten von echten X Fenstern. Mit dieser Graphikerweiterung ist die Verwendung von Graphik möglich, ohne auf die Xlib zugreifen zu müssen. Motif unterstützt selbst keine Graphikoperationen. Vor allem ermöglichen diese Graphikobjekte Drag und Drop. Dies wird von normalen Dialogobjekten des Xm++ Toolkits nicht unterstützt. Daher ist die Verwendung der Graphikobjekte unerläßlich, falls eine Applikation Drag und Drop verwenden soll. Leider ist diese Erweiterung des Toolkits nicht dokumentiert und ihre Verwendung muß anhand von Beispielprogrammen und des Quellcodes erschlossen werden.

Bei Beginn der vorliegenden Arbeit standen keine Erfahrungen zur Verfügung, welcher der beiden Toolkits besser für die Erstellung des graphischen Editors VisEd geeignet war. Bröckel ([Bröckel 1994]) hatte bei seiner Arbeit gute Erfahrungen mit dem Xm++ Toolkit gemacht. Daher wurde auch für diese Arbeit dieser Toolkit verwendet. Im Laufe der Arbeit zeigt sich aber, daß beim Xm++ Toolkit einige wünschenswerte Funktionen nicht vorhanden sind. Vor allem ist Drag

```
#include "xmObject.h"


class XmTest : public XmWindow
{
    void initialize();
public:
    XmTest() : XmWindow("Xm++ Test Window")  { }
};

void XmTest::initialize()
{
    addSubpane(Edit, "TestEdit", "Type in here:");
    edit("TestEdit")->setText("Hello World.");
}

void XmApp::initialize()
{
    (new XmTest)->open();
};
```

Abbildung 7.7: Eine kleine Xm++ Applikation (aus [Strassl, Binder 1994a])

und Drop nur bei den oben erwähnten virtuellen Fenstern möglich und nicht bei echten Widgets. Motif++ unterstützt jedoch Drag und Drop für Widgets. Auch stellte sich heraus, daß der Xm++ Toolkit wahrscheinlich nicht weiterentwickelt wird.

7.2 Objektorientiertes Programmieren mit C++

In den letzten Jahren wurden Softwareprodukte immer komplexer und umfangreicher und damit teuerer. Dies machte die Entwicklung von Programmiersprachen nötig, welche die Komplexität der Softwareentwicklung verringern. Ebenfalls wird gewünscht, daß neue Programmiersprachen die Kosten für Wartung und Pflege von Software verringern. Dies soll zum einen durch geringere sprachbedingte Fehleranfälligkeit erreicht werden. Des weiteren soll die Wiederverwertung existierender Software erleichtert werden. Elementare Datenstrukturen wie Stapel oder Listen sollten einmal implementiert und ausgetestet werden und dann immer

Abbildung 7.8: Oberfläche der Beispielapplikation

wieder verwendet werden können.

Ein Ansatz zur Erreichung dieser Ziele scheint das Paradigma der objektorientierten Programmierung zu sein. Prozedurale Programmieransätze gehen algorithmenorientiert vor. Prozeduren operieren auf meist globalen Datenstrukturen. Der objektorientierte Ansatz versucht einen Problembereich durch Objekte zu modellieren, die miteinander kommunizieren können. Objekte haben lokale Daten und verfügen über darauf arbeitende Operatoren.

Ein ähnlicher Ansatz wurde schon früher mit dem Konzept der abstrakten Datentype (ADT) unter Verwendung prozeduraler Sprachen wie Modula–2 oder auch C unternommen. Abstrakte Datentypen kapseln auch Daten ab und erlauben Zugriffe nur über explizite Zugriffsschnittstellen. ADTs bilden semantisch einen benutzerdefinierten Datentyp, meist sind sie als Zeiger auf Strukturen implementiert. Das X Fenstersystem und in C implementierte Benutzerschnittstellentoolkits implementieren ihre Datenobjekte als ADTs.

Objekte sind wie Strukturen implementiert. Tatsächlich sind in C++ die `class` und `struct` Schlüsselwörter austauschbar. ADTs sind Instanzen eines Typs, Objekte Instanzen einer Klasse. Eine wesentliche Erweiterung von Klassen gegenüber ADTs ist die Möglichkeit von Vererbung. Technisch geschieht dies durch Strukturerweiterung. Eine abgeleitete Klasse besitzt die Member der Basisklasse und eventuell noch weitere Member. Eine weitere wichtige Eigenschaft von objektorientierten Sprachen ist die Polymorphie, das ist die Möglichkeit, mehrere Funktionen mit gleichem Namen, aber unterschiedlichen Argumenttypen zu implementieren. Der Compiler entscheidet dann anhand der tatsächlich übergebenen Argumente, welche Funktion aufgerufen werden soll. Diese Eigenschaft ist unerläßlich bei Verwendung von Vererbung, da Methoden von Basisklassen durch gleichnamige Methoden der abgeleiteten Klassen überdefiniert werden können. Sehr nützlich sind auch Konstruktoren und Destruktoren. Dies sind benutzerdefinierbare Methoden, die beim Erzeugen bzw. Zerstören von Klasseninstanzen

automatisch aufgerufen werden. Hierbei können Initialisierungen von Objekten oder andere Operationen durchgeführt werden.

C++ ist eine Erweiterung von C (vergl. Abb. 7.9), die diese gewünschten

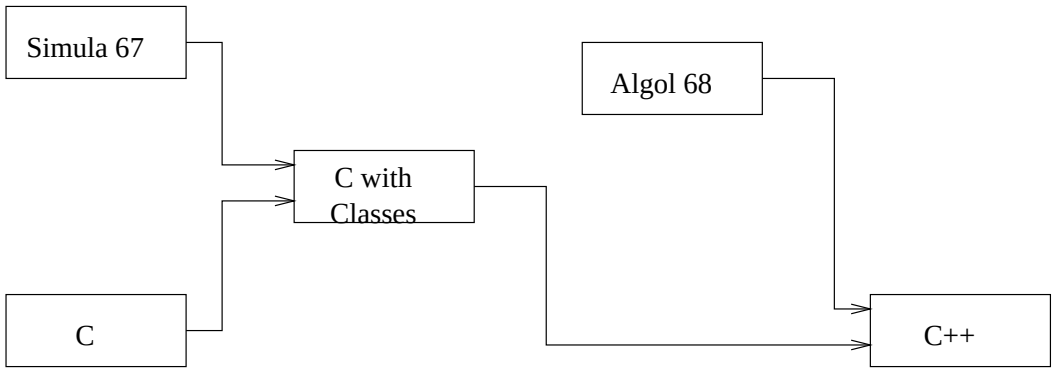

Abbildung 7.9: Die Entwicklung von C++ (aus [Krienke 1994])

Eigenschaften von objektorientierten Sprachen besitzt und dabei die Effizienz von C behält. C++ ist im wesentlichen eine Obermenge von C. C++ Compiler können also C Programme übersetzen, aber praktisch ist dies nur für Ansi–C Programme möglich, da C++ u.a. Funktionsprototypen verlangt. Eine kurze Einführung in C++ liefert

Bei der Implementierung von VisEd wurde das Klassenkonzept von C++ intensiv ausgenutzt. Der Quellcode von VisEd besteht fast nur aus Definitionen von Klassen und zugehörigen Methoden. VisEd arbeitet auf einer Autorensprache, die in einem Syntaxbaum repräsentiert wird. Die Knoten dieses Baumes sind als C++ Klassen implementiert. Der Syntaxbaum muß in eine Modellierung der visuellen Sprache übergeführt werden, mit welcher der Benutzer arbeitet. Diese Modellierung arbeitet ebenfalls objektorientiert. Auf– und Abbau dieser Datenstrukturen erledigen C++ Konstruktoren und Destruktoren. Bei den unterschiedlichen Sprachobjekten treten immer wieder gleiche Aufgaben auf. So müssen alle Objekte in der Lage sein, den ihnen entsprechenden Quellcode der Autorensprache zu erzeugen. Die dafür notwendige Methode hat bei allen Klassen den selben Namen. Hierfür von der Möglichkeit der Polymorphie von C++ Gebrauch gemacht.

Eine wichtiges Konzept von C++ ist die Bereitstellung von virtuellen Methoden. Dadurch kann auch für Objekte, die über einen Zeiger angesprochen werden, eine polymorph definierte Methode aufgerufen werden. Dies war bei der Implementierung von VisEd sehr wichtig, da die verwendeten Repräsentationen mit Zeigern auf Objekte arbeiten. Z.B. enthält ein Objekt, das ein Compound–Statement repräsentiert, einen Zeiger auf die davon abhängige Statementsequenz. Methoden dieses Sequenzobjektes werden über den Zeiger aufgerufen. Das ist nur bei Verwendung virtueller Methoden möglich.

Für die Übersetzung des Quellcodes von VisEd wurde der Gnu C++ Compiler verwendet.

7.3 Der Gnu C++ Compiler

Gnu Software ist eine Sammlung hochwertiger freier Software der Free Software Foundation, FSF. Die Programme sind von außerordentlich hoher Funktionalität und Portabilität (vergl. [Hühne 1994]). Die bekanntesten Programmpakete sind wohl der Gnu Emacs Editor und der Gnu C Compiler `gcc` ([Stallman 1993]). Gnu Software wird häufig an Hochschulen verwendet. Da es sich um freie Software handelt, gibt es natürlich keine Gewährleistung für Gnu Software. Auftretende Probleme werden aber meist sehr schnell in den Internet News Gruppen über Gnu Software gelöst. "Und das meistens schneller, als an Zeitaufwand nötig ist, um einen zuständigen und verständigen Mitarbeiter in der Compilerentwicklung der großen Unix–Lieferanten zu finden" ([Hühne 1994]). Jeder der großen kommerziellen Unix Hersteller liefert seine eigene proprietäre C/C++ Entwicklungsumgebung, die nie ganz kompatibel zu der der Konkurrenz ist, meist verschiedenartige Compilerschalter verstehen und unterschiedliche Fehlermeldungen liefern. Der Gnu `gcc` Compiler (bzw. die C++ Variante `g++`) läuft dagegen auf allen Systemen, produziert überall Code guter Qualität und wird überall gleich bedient. Die Qualität des Gnu C Compilers zeigt sich daran, daß das zur Zeit stabilste und leistungsfähigste Unix für Intel PCs, das frei erhältliche Linux, mit dem Gnu C Compiler entwickelt wurde. Mit dem Gnu C Compiler steht damit ein Ansi–C Compiler zur Verfügung, der die vollständige Portierbarkeit von Software garantiert. Nach einfacher Neuübersetzung auf der Zielmaschine sollte die im Rahmen dieser Arbeit erstellte Software auf allen System unter Unix und X11/Motif lauffähig sein, auf die der Gnu C Compiler portiert wurde, also auf praktisch allen existenten Unix Systemen. Getestet wurde die Lauffähigkeit der erstellten Software unter HP-UX, Linux und SunOs. Aus diesen Gründen wurde bei der vorliegenden Arbeit dieser Compiler verwendet.

Zur Fehlersuche in C++ Programmen auf Source Code Ebene kann der ausgezeichnete Gnu Debugger `gdb` benutzt werden. Er bietet umfangreiche Möglichkeiten zum Einzelschrittdurchlauf eines Programmes. Dabei können auch Variablen untersucht werden. Bedient wird der Debugger über textuelle Befehle. Unter X Windows existiert jedoch eine graphische Oberfläche für den Debugger, so daß die wichtigsten Befehle beim Debuggen mit der Maus ausgeführt werden können (vergl. Abb. 7.10). Leider stand der Gnu Debugger nur unter Linux und SunOs zur Verfügung.

Weitere Gnu Entwicklungswerkzeuge, die für diese Arbeit verwendet wurden, sind der Scannergenerator `flex` und der Parsergenerator `bison`, die kompatibel zu Lex bzw. Yacc sind.

Gnu Software ist nicht verkäuflich, darf aber frei weitergegeben werden. Ei-

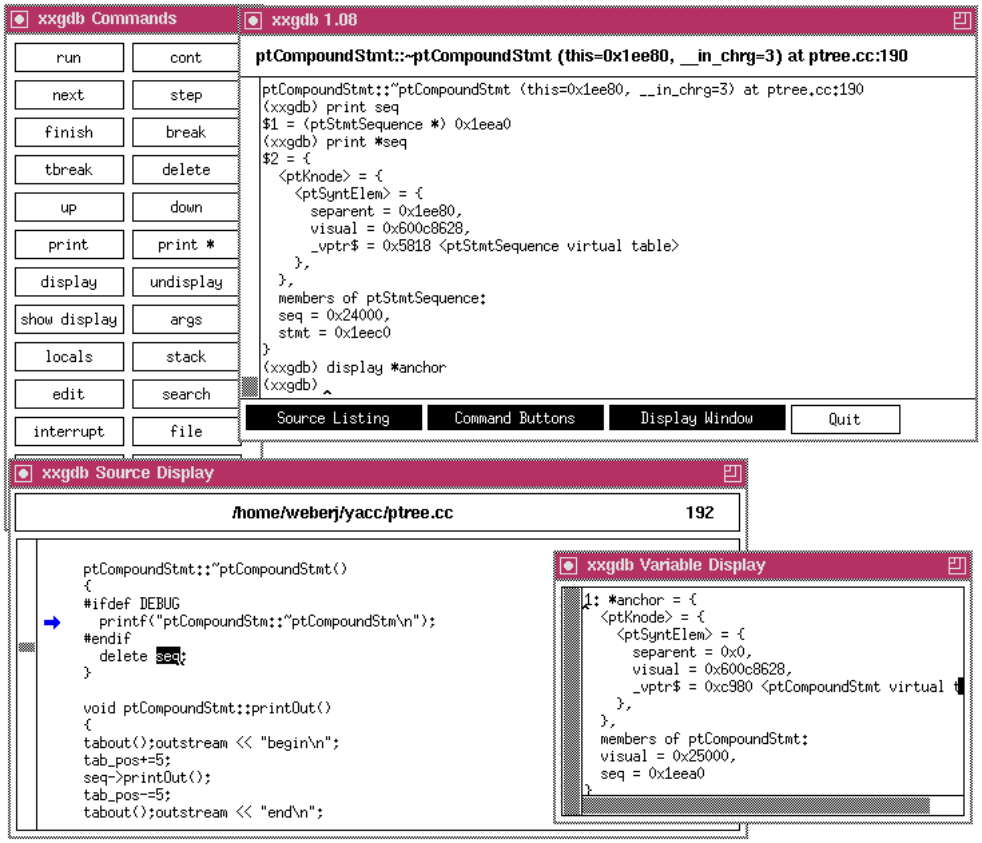

Abbildung 7.10: Fehlersuche mit dem Gnu Debugger Im Hauptfenster rechts oben können Befehle in textueller Form eingegeben werden. Mit der Buttonleiste links daneben können viele wichtige Befehle auch mit der Maus erreicht werden. Das Fenster links unten zeigt den Teil des Quellcodes an, der gerade ausgeführt wird. Im Fenster rechts unten ist das Objekt beschrieben, auf das der im Quellcodefenster markierte Zeiger verweist.

gentümer der Gnu Software bleibt die Free Software Foundation, die jedoch die kostenlose Nutzung der Software erlaubt. Kommerzielle Software darf mit Gnu Tools entwickelt werden. Von verwendeten Gnu Libraries muß aber der Quellcode beigelegt werden.

Kapitel 8

Konzepte und Werkzeuge des Compilerbaus

Mit dem in dieser Arbeit entwickelten graphischen Editor VisEd, der zur Erstellung von Multimediapräsentationen vorgesehen ist, soll es sowohl möglich sein, eine Präsentation zu erstellen als auch eine bestehende Präsentation zu verändern. Die mit visueller Programmierung erstellten Präsentationen werden in VisEd in Form von baumähnlichen Datenstrukturen repräsentiert. Auf Sekundärspeichern sollen Präsentationen in Form von Quellcode einer von Bröckel ([Bröckel 1994]) erstellten und in der vorliegenden Arbeit weiterentwickelten Autorensprache gesichert werden können. Umgekehrt müssen Quellcodes der Autorensprache eingelesen und im Editor zur Bearbeitung dargestellt werden. Für Abspeichern, Einlesen und Aufbau der internen Datenstrukturen des Editors sind damit Techniken aus dem Compilerbau notwendig. Im Folgenden werden daher einige Aspekte von Compilern kurz dargestellt.

8.1 Grundbegriffe aus dem Compilerbau

Ein Compiler ist ein Softwareprogramm, das einen in einer Quellsprache geschriebenen Quelltext einliest und diesen in ein Zielprogramm übersetzt. Eventuell weden dabei noch Fehlermeldungen ausgegeben (vergl. Abb. 8.1).

Üblicherweise ist der zu übersetzende Quellcode in einer höheren Programmiersprache geschrieben, der in eine Sprache niederer Ebene wie Assemblercode übersetzt werden soll. Dabei durchläuft der Compiler mehrere Phasen der Übersetzung (vergl. Abb. 8.2).

Für VisEd werden nur die ersten beiden Phasen benötigt, da danach ein Syntaxbaum aufgebaut ist, mit dem VisEd intern die Autorensprache repräsentiert.

Der Quellcode, die Eingabe des Compilers, wird zeichenweise gelesen. Bei der lexikalischen Analyse wird diese Zeichenfolge in eine Folge von lexikalischen Elementen (Token bzw. Lexeme) zerlegt. Das Token ist eine Zahl, die ein Lexem

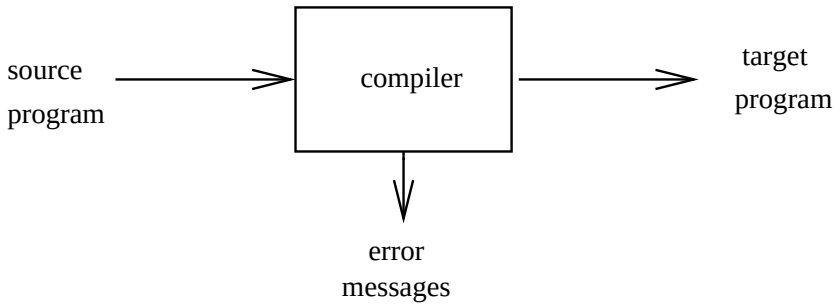

Abbildung 8.1: Aufbau eines Compilers (aus [Aho et al. 1986])

identifiziert. Statt dem ganzen Eingabewort, z.B. einem Schlüsselwort, gibt der Scanner dann nur noch die Zahl weiter. Der Scanner wird vom Parser aufgerufen und liefert an ihn eine Folge von Token.

Der Parser führt die syntaktische Analyse durch. Dabei wird versucht, aus dem vom Scanner gelieferten Tokenstrom gemäß der Grammatik, die die Sprache beschreibt, einen Syntaxbaum aufzubauen. Dazu muß getestet werden, ob der Tokenstrom von der gegebenen Grammatik erzeugt werden kann. Der Parser fordert dabei laufend ein geeignetes Token an (vergl. Abb. 8.3).

Verschiedene Verfahren wurden entwickelt, um Sprachen zu parsen. Diese Verfahren können in top–down Parsing und bottom–up Parsing unterschieden werden. Top–down Parsing versucht, ausgehend von der Spitze des Syntaxbaumes die unteren Teile des Baumes aufzufüllen. Dagegen erzeugt bottom–up Parsing zuerst die Blätter des Baumes und baut dann höhere Ebenen auf. Top–down Parser lassen sich leicht nach der Methode des rekursiven Abstiegs (vergl. [Aho et al. 1986]) implementieren, wobei für jede Regel der Grammatik eine Prozedur des Parsers existieren muß. Bottom–up Parser sind dagegen effizienter, aber aufwendig zu implementieren.

8.2 Compiler Generatoren

Zur Unterstützung bei der Entwicklung von Compilern wurden schon früh Werkzeuge entwickelt. Diese werden üblicherweise Compilergeneratoren oder Compiler-Compiler genannt. Die Verwendung von Compilergeneratoren liegt dadurch nahe, daß verschiedene Bestandteile des Compilers unabhängig von der übersetzten Sprache und dem Einsatz oft sehr ähnliche Funktion haben. So unterscheiden sich Scanner für die Erkennung verschiedener Programmiersprachen nur in den erkannten Schlüsselwörtern. Parser für Programmiersprachen unterscheiden sich nur in der Grammatik der erkannten Sprache. Daher können Scannergeneratoren aus einer Definition von regulären Ausdrücken einen Scanner erzeugen. Parsergeneratoren benötigen zum Bauen eines Parsers die Definition einer kontextfreien Grammatik.

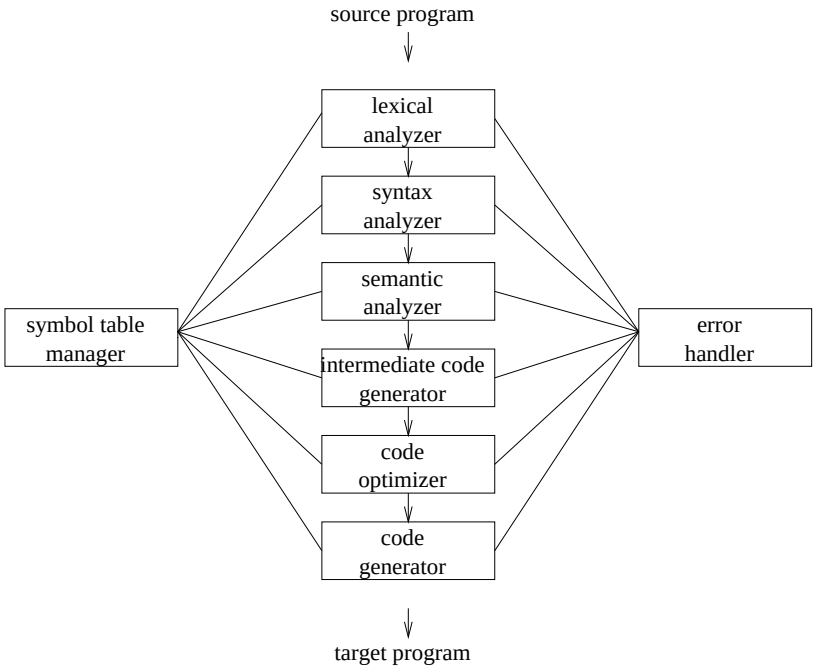

Abbildung 8.2: Übersetzungsphasen eines Compilers (aus [Aho et al. 1986])

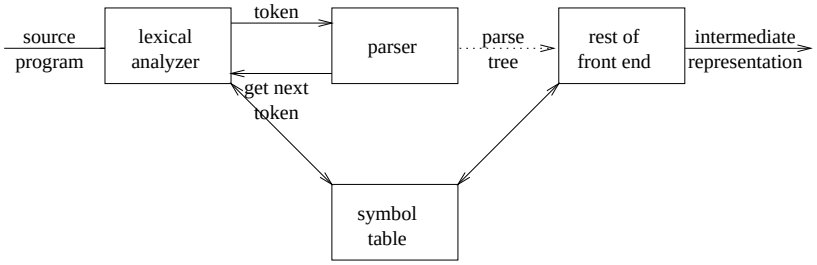

Abbildung 8.3: Position des Parsers im Compiler–Modell (aus [Aho et al. 1986])

Ein wichtiger Vorteil bei der Verwendung von Generatoren ist, daß Schlüsselwörter und Grammatik der Sprache leicht geändert werden können, ohne daß wesentliche Änderungen am Programmcode nötig werden.

Häufig verwendete Generatoren sind der Scannergenerator `lex` und der Parsergenerator `yacc`, die als Teil von Unix Betriebssystemen ausgeliefert werden. Für die vorliegende Arbeit wurden die frei erhältlichen Scanner- bzw. Parsergeneratoren `flex` und `bison` verwendet, die kompatibel zu `lex` und `yacc` sind, jedoch C–Code erzeugen, der auch den strengeren semantischen Prüfungen eines C++ Compilers standhält. Im weiteren wird dennoch immer von Lex und Yacc gesprochen. Damit sind auch alle zu Lex und Yacc kompatible Generatoren abgedeckt.

8.2.1 Lex

Zum Einlesen des Quellcodes der Autorensprache aus einer Datei wird ein Scanner verwendet. Scanner sind Werkzeuge, mit denen lexikalische Muster von Zeichenfolgen in einem Eingabetext erkannt werden können. Mit `lex` können solche Scanner automatisch aus einer textuellen Beschreibung erzeugt werden. Die Beschreibung besteht aus Regeln, die Paare von regulären Ausdrücken und C–Code sind. Die regulären Ausdrücke definieren eine Menge von Zeichenfolgen. Wird eine Zeichenfolge aus dieser Menge erkannt, wird der zur Regel gehörende C–Code ausgeführt. Eine einfache Regel kann so aussehen:

```
halloh   printf("halloh erkannt\n");
```

Der reguläre Ausdruck ist hier nur eine Verkettung der Zeichen des Wortes halloh und deckt genau dieses Wort ab. Erkennt der von Lex erzeugte Scanner diese Zeichenfolge im Eingabetext, wird der zur Regel gehörige C–Code ausgeführt. Eine Beschreibungsdatei für einen von Lex zu erzeugenden Scanner besteht aus drei Abschnitten, die durch `%%` getrennt sind:

Definitionen
%%
Regeln
%%
Benutzercode

Im Abschnitt für Definitionen kann regulären Ausdrücken ein symbolischer Name zugewiesen werden, z.B.

```
DIGIT  [0..9]
ID     [a-z][a-z0-9]*
```

Der Regelteil enthält eine Folge von Regeln der Form `pattern action`. Pattern sind reguläre Ausdrücke. Ihre Lex Definition ist in Abbildung 8.4 dargestellt.

Die Aktion geht bis zum Zeilenende, bzw. wenn sie mit einer öffnenden geschweiften Klammer begann, bis zur korrespondierenden schließenden Klammer. Üblicherweise wird die Aktion darin bestehen, an den aufrufenden Parser ein Token als Code für ein erkanntes Lexem zu liefern. In Abbildung 8.5 befindet sich eine Lex Beschreibung für einen Scanner, der in einem Eingabetext Folgen von Blanks und Tabs durch einzelne Leerzeichen ersetzt und am Zeilenende entfernt. An diesem Beispiel ist ersichtlich, wie mit Hilfe von Lex und einem C Compiler ein Werkzeug erzeugt werden kann, das unter ausschließlicher Verwendung einer konventionellen Programmiersprache deutlich mehr Aufwand erfordert, wobei natürlich auch mehr Fehlermöglichkeiten vorhanden sind.

`x`	paßt auf das Zeichen `x`
`.`	paßt auf alle Zeichen außer Newline
`[xyz]`	die Klasse der Zeichen `x`, `y` und `z`
`[abj-oZ]`	eine Zeichenklasse mit einer Bereichsangabe
`[^A-Z]`	eine negierte Klasse
`r*`	n–fach `r` hintereinander, wobei `r` ein regulärer Ausdruck, $n \geq 0$
`r+`	wie `r*`, nur $n > 0$
`r?`	`r` ist optional
`rs`	Verkettung der Ausdrücke `r` und `s`
`r|s`	`r` oder `s`
`^r`	`r` am Zeilenanfang
`r$`	`r` am Zeilenende

Abbildung 8.4: Wichtige reguläre Ausdrücke in Lex

Lex ist ein mächtiges Werkzeug, das dem Entwickler erspart, selbst unter hohem Aufwand einen Scanner zu schreiben. Der Entwickler kann sich auf die Definition der regulären Ausdrücke konzentrieren. Bei der Entwicklung des graphischen Editors VisEd mußten nur reguläre Ausdrücke für die Schlüsselwörter der Autorensprache angegeben werden.

8.2.2 Yacc

Zur Erzeugung von Parsern aus einer Beschreibungsdatei wurden Parsergeneratoren entwickelt. Der am weitesten verbreitete ist `yacc`, der auch mit dem Unix Betriebssystem ausgeliefert wird. Die Eingabe von Yacc besteht aus Regeln einer kontextfreien Grammatik, in die semantische Aktionen in Form von C–Statements eingebettet werden können. Ähnlich wie ein Lex Eingabefile besteht auch ein Eingabefile für Yacc aus mehreren Abschnitten.

```
%{
C Deklarationen
%}
Yacc Deklarationen %%
Grammatik Regeln
%%
Benutzercode
```

```
%%
[ \t]+        putchar( ' ' );
[ \t]+$       /* ignore this token */
%%
main( argc, argv ) int argc,char **argv;
{
  ++argv, --argc;  /* skip over program name */
  if ( argc > 0 )
    yyin = fopen( argv[0], "r" );
  else
    yyin = stdin;
  yylex();
}
```

Abbildung 8.5: Ein Lexquellcode (aus [Paxson 1993])

In den C–Deklarationen können Variablen oder Datentypen definiert oder Header eingebunden werden. Die Yacc–Grammatikregeln bestehen aus kontextfreien Produktionen. So könnte eine Beschreibung für das C `return` statement wie folgt aussehen.

```
stmt: RETURN expr ';' { /* C Code zur Auswertung */ }
      ;
```

`RETURN` ist ein als C Konstante definiertes Token, das vom Scanner geliefert wird, `expr` muß noch in einer weiteren Regel definiert werden.

8.2.3 Erzeugung von Generatoren mit Lex und Yacc

Lex und Yacc erzeugen aus einer Beschreibung der Schlüsselwörter bzw. der Grammatik einer Sprache C–Quellcode. Beim graphischen Editor VisEd werden dadurch Module erzeugt, die eine Autorensprache lexikalisch und syntaktisch analysieren. Bei der syntaktischen Analyse wird aus dem eingelesenen Quellcode der Autorensprache ein Syntaxbaum aufgebaut. Daraus wird in einem anderen Modul eine für die Manipulation durch den Benutzer geeignete visuelle Darstellung der Präsentation erzeugt. Die von den Generatoren erzeugten C–Quelltexte werden mit einem C–Compiler übersetzt und mit den restlichen Modulen des Editors zu einem lauffähigen Programm gebunden (vergl. Abb. 8.6). Soll ein Quellcode der Autorensprache eingelesen werden, wird eine Routine `yyparse()` aufgerufen, welche die Hauptfunktion im von Yacc erzeugten Parser ist. Diese fordert laufend

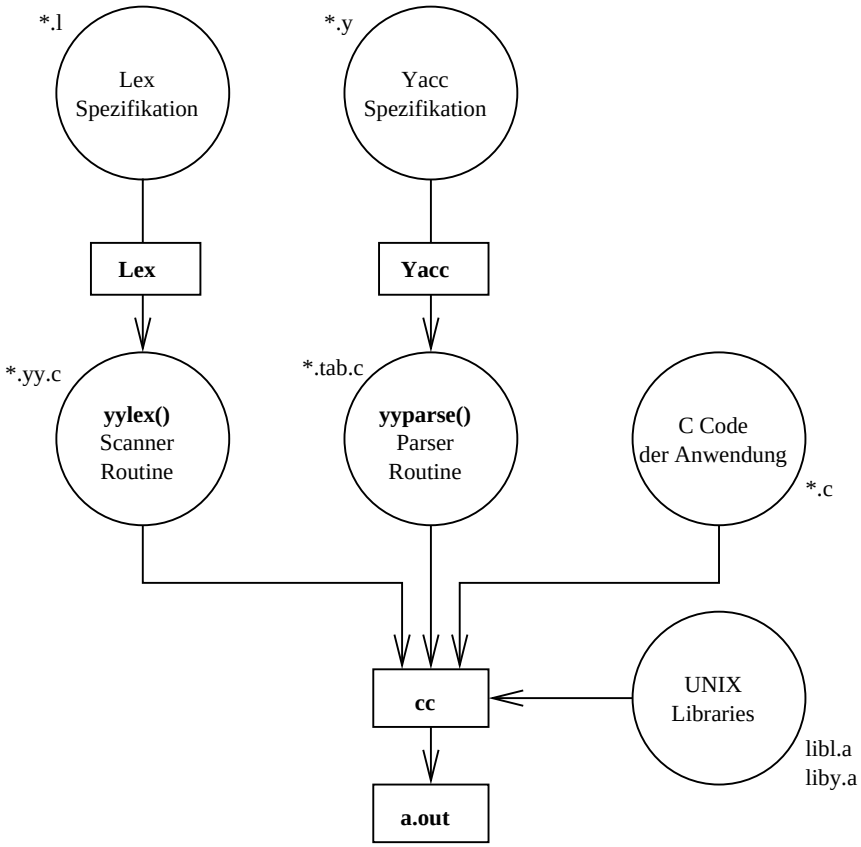

Abbildung 8.6: Make mit Lex und Yacc (aus [Mason, Brown 1990])

durch Aufruf der Routine `yylex()`, die im von Lex erzeugten Scanner ist, Lexeme an. Die Parserroutine `yyparse()` testet dabei, ob der erhaltene Strom von Lexemen mit der Grammatik erzeugbar ist, welche die Autorensprache beschreibt. Ist dies der Fall, handelt es sich um eine korrekte Spezifikation einer Präsentation. Während des Parsens wird ein Syntaxbaum aufgebaut. In Abbildung 8.7 wird der Kontroll– und Datenfluß zwischen Scanner und Parser gezeigt. Falls `yyparse()` einen Wert ungleich Null zurückliefert, enthielt die eingelesene Präsentation Fehler. Dann muß der bisher aufgebaute Teil des Syntaxbaumes verworfen werden.

Die Sprache, die der von Yacc erzeugte Parser erkennt, wird im folgenden Kapitel beschrieben.

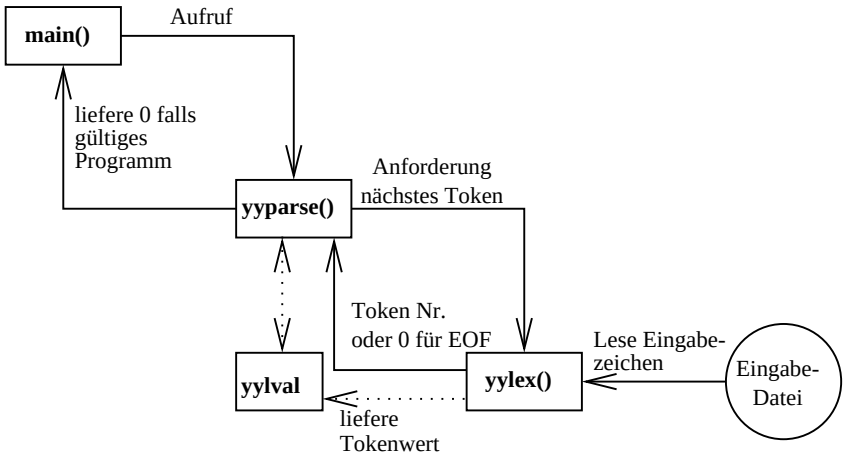

Abbildung 8.7: Kontrollfluß im mit Lex und Yacc erzeugten Compiler (aus [Mason, Brown 1990])

Kapitel 9

Das verwendete Lösungskonzept

In der vorliegenden Arbeit wurde ein graphischer Editor zur Erstellung von multimedialen Präsentationen entwickelt. Diese ist auf dem Weg der visuellen Programmierung möglich. Mit dem Editor wird eine graphische Darstellung der Präsentation bearbeitet. Zur Darstellung wird eine Petrinetz–analoge Repräsentation verwendet.

9.1 Konzepte der Benutzeroberfläche

Der graphische Editor VisEd verwendet eine ähnliche Benutzeroberfläche wie andere graphische Editoren. Es gibt eine Palette von Sprachobjekten, aus denen eine Präsentation zusammengestellt werden kann. Bei VisEd sind hierfür Icons in einer Reihe übereinander dargestellt. Mit der Maus können die Icons auf eine Arbeitsfläche bewegt werden. Präsentationen können durch Compound–Statements modularisiert werden. Sie werden wie alle Sprachmittel durch Icons repräsentiert. Compound–Statements können geöffnet werden. Es ist dann eine neue Arbeitsfläche zu sehen, auf der die Sprachobjekte dargestellt sind, welche durch das Compound gruppiert werden. Für jedes geöffnete Compound–Statement ist eine Arbeitsfläche zu sehen. In Abbildung 9.1 ist bisher nur das globale Compound–Statement erstellt.

Alle Icons besitzen ein Menü. Damit können Attribute des zugehörigen Sprachobjektes bearbeitet werden. Attribute sind z.B. der Pfad der Daten, die beim View–Statement dargestellt werden sollen. Auch kann dadurch dem Icon für das Compound–Statement der Inhalt über das Menü sichtbar gemacht werden.

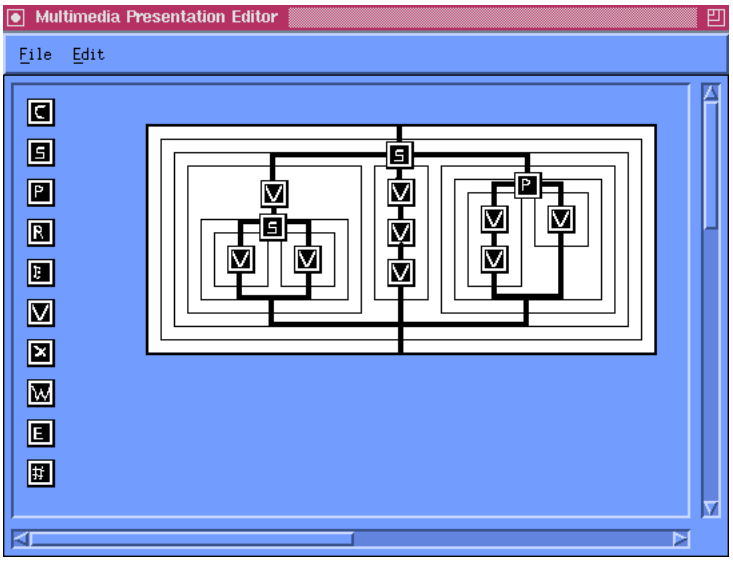

Abbildung 9.1: Die Benutzeroberfläche von VisEd. Die Strukturen der visuellen Sprache sind in Rechtecken angebracht. Sie sind hier noch sichtbar, können aber leicht auf die Hintergrundfarbe gesetzt werden.

9.2 Das verwendete Erstellungsmodell für Präsentationen

Zur Erstellung einer multimedialen Präsentation sind verschiedene Modelle möglich (vergl. Abschnitt 3.5). Für VisEd wurde eine Abwandlung des Petrinetzmodelles verwendet. Mit Petrinetzen lassen sich parallele Aktionen spezifizieren, die beendet sein müssen, wenn Verzweigungen des Netzes wieder zusammentreffen. Zwischen Anfang und Ende von Verzweigungen laufen Aktionen ohne weitere Restriktionen ab. Im Projekt POWER werden externe Viewer eingesetzt, die keine Funktionalität für Synchronisation mit anderen Viewern besitzen. Daher kann eine Darstellung eines Monomediums als atomare Aktion betrachtet werden und entspricht damit einer Stelle im Petrinetzmodell.

Die Darstellung von Petrinetzen wurde soweit verändert, daß Transitionen einfach als Verzweigungen der Flußlinie erscheinen. Auf den Verzweigungspunkten erscheinen Icons, welche die Art der Verzweigung erklären. Damit kann z.B. unterschieden werden, ob es sich bei der Verzweigung um ein Selection– oder Parallel–Statement handelt. Beide werden als Verzweigung und anschließende Zusammenführung des Flußgraphen dargestellt. Beim Selection–Statement wird jedoch nur der vom Benutzer zur Laufzeit der Präsentation gewählte Teilpfad ausgeführt.

9.3 Modularisierung des Editors

Der graphische Editor VisEd hat mehrere Teilaufgaben zu erfüllen:

- Einlesen und Parsen eines Quellcodes der Autorensprache
- Aufbau und Verwaltung eines Syntaxbaumes
- Aufbau und Verwaltung einer Datenstruktur von Objekten der visuellen Programmiersprache
- Aufbau der graphischen Benutzeroberfläche, Ausgabe der Sprachobjekte auf den Bildschirm und Ermöglichung von Benutzermanipulation

Entsprechend dieser Aufgabentrennung wurde die Software modular entwickelt. VisEd besteht aus vier Teilmodulen (vergl. Abb. 9.2):

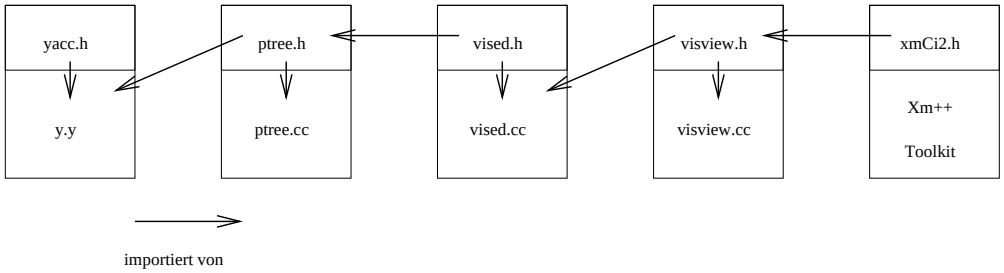

Abbildung 9.2: Module von VisEd (vereinfacht dargestellt)

- Aus `lex.l` und `y.y` wird mit den Generatoren Lex und Yacc ein Parser für die Autorensprache erzeugt, der einen Syntaxbaum aufbaut.
- Im Modul `ptree` sind Klassen von Objekten, aus denen der Syntaxbaum besteht und Methoden, um diesen Syntaxbaum in die Repräsentation der visuellen Programmiersprache überzuführen, enthalten.
- `vised` bietet die Funktionalität zur Erzeugung und Manipulation der visuellen Programmiersprache.
- `visview` bildet die Schnittstelle zum benutzten Oberflächentoolkit und stellt die Objekte der visuellen Programmiersprache auf dem Bildschirm dar.

Die erste Aufgabe wird durch den von den Generatoren Lex und Yacc erzeugten Parser erfüllt. Auch der Aufbau des Syntaxbaumes ist relativ einfach. Verfahren hierzu werden in der Literatur über Compilerbau ([Aho et al. 1986]) beschrieben. Die Literatur über visuelle Programmierung ([Glinert 1990a], [Glinert 1990b],

[Shu 1988]) beschreibt aber nur die Benutzeroberflächen von Systemen zur visuellen Programmierung, der innere Aufbau und die Datenstrukturen werden nicht beschrieben. Daher mußten für diese Arbeit geeignete Datenstrukturen zur Repräsentation der visuellen Programmiersprache und eine Methode entwickelt werden, aus dem Syntaxbaum der Autorensprache einen Baum von Sprachobjekten der visuellen Programmiersprache aufzubauen. Für die visuelle Programmiersprache kann kein konventioneller Syntaxbaum verwendet werden.

Die Objekte der visuellen Programmiersprache werden nicht direkt auf dem Bildschirm dargestellt. Gemäß dem MVC–Paradigma (vergl. Abschnitt 4.5) existiert eine logische und programmtechnische Trennung von internen Datenstrukturen und deren Visualisierung auf der Benutzeroberfläche. Auf die im MVC–Paradigma vorgesehene Trennung von View und Controller wurde verzichtet, die Objekte der visuellen Repräsentation verwalten auch Benutzereingaben.

Durch die Aufspaltung der Datenstruktur der visuellen Programmiersprache in Objekthierarchien von Model und View wird es möglich, den visuellen Editor leicht an ein anderes Interfacetoolkit oder sogar an ein völlig anderes Fenstersystem anzugepassen, wobei nur ein Teilmodul des visuellen Editors geändert werden muß. Das `vised` Modul geht nur von der Existenz von Oberflächenobjekten aus, die eine Fläche benötigen. Diese Minimalforderung ist bei allen Benutzeroberflächen erfüllt.

Kapitel 10

Beschreibung der Autorensprache

Ziel dieser Arbeit war es, einen graphischen Editor zur Spezifikation von Multimediapräsentationen zu entwickeln. Erstellte Präsentationen sollen als Quelltext einer Autorensprache gesichert werden können. Eine Autorensprache stellt andere Anforderungen als eine konventionelle Programmiersprache. Eine Autorensprache, in der Präsentationen spezifiziert werden, sollte mächtige Konstrukte anbieten, mit denen auf möglichst einfache Weise Monomedien dargestellt werden können. Sprachkonstrukte zur Manipulation von Daten sind dagegen nicht notwendig. Für einfache Präsentationen werden auch keine Kontrollstrukturen benötigt. Der graphische Editor VisEd unterstützt eine Autorensprache, die auf einem Entwurf von Bröckel ([Bröckel 1994]) basiert.

10.1 Die Autorensprache von Bröckel

Bröckel entwickelte die Autorensprache mit dem Ziel, Anwendern eine einfache Möglichkeit der Erstellung von Präsentationen zu bieten. Dabei spielen Konstrukte zur Beschreibung des Präsentationsablaufes eine zentrale Rolle. Die Autorensprache ist in Abbildung 10.1 in EBNF–ähnlicher Form dargestellt.

Bröckel nennt u.a. folgende Anforderungen an die von ihm entwickelte Sprache:

- Die Sprache muß Befehle zum Aufruf und Abbruch von Informationsdarstellungen haben. Darstellungen sollten auf verschiedenen Rechnern ablaufen können. Eine Darstellung erfolgt mit dem view–Befehl. Dieser Befehl wird in einen Aufruf eines externen Viewers übersetzt. Daher sollten beim view–Befehl Optionen für diesen Viewer übergeben werden können.

- Ein Medium sollte ausreichend lange dargestellt werden. Der Benutzer muß eine Möglichkeit haben, die Darstellung zu beenden.

- Es sollte unterschiedliche Ablaufzweige der Präsentation geben, die vom Benutzer wählbar sind. Verzweigungen sollten mehrstufig möglich sein.

```
SCRIPT_DESCRIPTION → "#script"NR {STATEMENT}

NR                 → "1.0"

STATEMENT          → VIEW_STATEMENT | CLOSE_STATEMENT |
                     WAIT_STATEMENT | CD_STATEMENT |
                     SELECTION_STATEMENT

VIEW_STATEMENT     → [REMOTE] "view" NAME {OPTIONS}

OPTIONS            → NAME [LETTER | NAME | INTEGER | FLOAT]

CLOSE_STATEMENT    → "close" NAME | "closeall"

REMOTE             → "display=" NAME |
                     "host=" NAME

CD_STATEMENT       → "cd" NAME

WAIT_STATEMENT     → "wait" [INTEGER]
SELECTION_STATEMENT→ "selection" """ TEXT """
                     {"button=" """ NAME """ ":" {STATEMENT}}
                     "end"

DIGIT              → "0" |...| "9"

INTEGER            → ("1" |...| "9") {DIGIT}

FLOAT              → INTEGER "." {DIGIT}

LETTER             → "a" |...| "z" | "A" |...| "Z" |
                     "." | "_" | "-" | "+" | ":" | "/"

NAME               → {DIGIT | LETTER}

STRING             → {NAME | "!" | "@" | "#" | "$" | "&" |
                     "*" | "(" | ")" | "=" | "{" | "}" |
                     "|" | ";" | "," | "?" | "\n"}

TEXT               → STRING {" " STRING}
```

Abbildung 10.1: Die Autorensprache von Bröckel (aus [Bröckel 1994])

Die Sprache enthält Statements und Kontrollstrukturen zum Anzeigen von Informationseinheiten und für benutzergesteuerte Verzweigungen.

Das view–Statement dient zum Anzeigen eines Mediums. Dabei ist die Angabe eines Rechners und von Optionen möglich. Mit dem close–Statement kann eine Darstellung beendet werden. Der wait–Befehl unterbricht den Ablauf der Präsentation, bis alle view–Befehle beendet sind. Mit der selection–Anweisung

sind benutzergesteuerte Verzweigungen möglich. Dazu wird eine Dialogbox aufgebaut, mit der vom Benutzer ein Ablaufpfad ausgewählt wird.

Damit lassen sich Präsentationsgraphen in Baumstruktur aufbauen. Statements für Schleifen, Sprünge und Unterprogramme enthält die Sprache nicht. Sie wurde dahingehend entwickelt, die gleichen Möglichkeiten zu bieten, über die Benutzer verfügen, wenn sie die Oberfläche eines von Bröckel entwickelten Multimediainformationssystemes benutzen. Bröckel entwickelte einen Compiler, der die Autorensprache in Unix Shell Scripte übersetzt. Die Sprache bietet daher auch nur Möglichkeiten, die auch bei Shell Scripten existieren. Dennoch ist diese Sprache mächtig genug, um Präsentationen zu spezifizieren, die im betrieblichen Umfeld anfallende Produktdaten darstellen.

In der vorliegenden Arbeit wurde ein graphischer Editor entwickelt, mit dem mittels visueller Programmierung Quellcodes dieser Autorensprache erstellt werden können. Dabei wurde auch die Sprache so erweitert, daß die oben erwähnten Einschränkungen wegfallen. Es wurden jedoch einige grundlegende Änderungen notwendig, so daß die geänderte Autorensprache nicht mehr zur von Bröckel entwickelten Sprache kompatibel ist.

10.2 Änderungen an der Autorensprache von Bröckel

Ein wichtiger Änderung erfolgte an der Semantik des view–Statements. Bei Bröckel werden view–Statements grundsätzlich asynchron, d.h. im Hintergrund ausgeführt. Diese Regel wird dann durchbrochen, wenn zwischen view–Statements wait–Statements auftreten. Dann erfolgt die Abarbeitung sequentiell. In der vorliegenden Arbeit wurde jedoch ein Sprachkonstrukt für explizite Parallelität eingeführt, da dann die Semantik offensichtlicher ist. Zusätzlich läßt sich damit Parallelität besser darstellen. Zwischen `parbegin` und `parend` geklammerte Statements lassen deren Parallelität besser erkennen als das Fehlen von Waitstatements.

Das cd–Statement wird nicht mehr unterstützt, da der Ablageort einer Informationseinheit ein Attribut des view–Statements ist. Die Kontrollstrukturen der Sprache dienen der Spezifikation des Präsentationsablaufes und nicht der Lokalisierung von Informationen.

In Abbildung 10.2 ist die veränderte Grammatik dargestellt. Bröckel gab die Sprachdefinition in EBNF Syntax an (Abb. 10.1). Der Yacc Compilergenerator verlangt jedoch die im Compilerbau übliche Sprachdefinition als kontextfreie Grammatik. Der Hauptunterschied zwischen EBNF und kontextfreien Grammatiken besteht in der Beschreibung von eventuell optionalen Wiederholungen. EBNF setzt Konstrukte, die n-fach wiederholt werden können ($n \geq 0$), in geschweifte Klammern. Bei kontextfreien Grammatiken muß dies durch Rekursion ausgedrückt werden. Da LR Parser wie die von Yacc erzeugten von links nach rechts

parsen und damit Linksrekursion effizienter als Rechtsrekursion parsen können, werden Linksrekursionen verwendet. Z.B. wird

`S ::= A{b}`

umgewandelt in

`S` $\rightarrow$ `AB`

`B` $\rightarrow$ `Bb | ` ε

Die Umwandlung erfolgt dadurch, daß für den Teil in Klammern eine eigene Regel eingeführt wird und die Wiederholung durch Linksrekursion ermöglicht wird. Der ε–Teil sorgt für den Rekursionsabbruch.

10.2.1 Weitere Sprachkonstrukte

Eine wesentliche Regel der Grammatik ist die für `statementseq`. Damit läßt sich rekursiv eine Folge von Statements ableiten.

Im Unterschied zur Grammatik von Bröckel sind jetzt noch `compound`, `repeat`, `parallel`, `execute`, `call` und `comment` von `statement` ableitbar.

Die `compound` Konstruktion gruppiert Statements zu einem Block zusammen. Dadurch wird vor allem in der graphischen Repräsentation Abstraktion von einzelnen Statements ermöglicht. Vor allem aber kann zusammen mit dem `call` Statement eine Präsentation mit Netzstruktur aufgebaut werden.

Mit `repeat` kann eine Wiederholung der Statement Sequenz abhängig von der Wiederholungsbedingung `cond` spezifiziert werden. In der vorliegenden prototypischen Implementierung kann als Bedingung nur eine Zahl angegeben werden. Eine Erweiterung auf beliebige Ausdrücke sollte leicht möglich sein.

Auch nach Einführung des `repeat` Statements können nur Präsentationen mit Baumstruktur aufgebaut werden. Zwar können jetzt Zyklen im Baum auftreten (der Graph verliert damit die Baumeigenschaft), jedoch ist noch keine allgemeine Netzstruktur möglich. Dies wird durch das `branch` Statement möglich. Jetzt können beliebige Compound Statements aufgerufen werden, die an anderer Stelle definiert wurden. Der String im `branch` Statement ist der Bezeichner des Compound Statements. Die `branch calling` Variante kehrt nach Beendigung des aufgerufenen Compound Statements wieder zurück.

Parallelität von Statements wird durch das `parbegin` ... `parend` Konstrukt angegeben. Die einzelnen parallel auszuführenden Zweige werden durch das Barzeichen | getrennt. Die hier verwendete Semantik des Parallelitätskonstruktes erfordert, daß das Parallelkonstrukt erst beendet wird, wenn sämtliche Zweige beendet sind. Soll dagegen z.B. ein Bild während der ganzen Präsentation angezeigt werden, muß dies als paralleler Zweig zum ganzen Rest der Präsentation spezifiziert werden.

Eine andere Möglichkeit wäre, Statements ein Attribut zur Ausführung im Hintergrund zu geben, so daß Viewer mit ähnlicher Semantik gestartet werden könnten, wie wenn sie von der Unix Shell aus mit Anhängen des Ampersand &

aufgerufen würden. Ein solches Attribut würde aber das Prinzip der strukturierten Programmierung und das Petrinetzmodell durchbrechen.

Das `execute` Statement ermöglicht, beliebige Unix Befehle aus der Autorensprache heraus zu starten. Damit wird deren Mächtigkeit wesentlich erhöht.

10.2.2 Kommentare bei visueller Programmierung

Üblicherweise behandeln bei der Sprachübersetzung die Scanner Kommentare wie Leerzeichen, so daß der Parser nichts von ihnen erfährt. Im vorliegenden System ist dies jedoch von Nachteil. Der Benutzer soll den Quellcode der Autorensprache sowohl mit einem Texteditor als auch über den graphischen Editor bearbeiten können. Dieser arbeitet aber nicht auf dem Quelltext, sondern auf einer aus dem Syntaxbaum erzeugten Datenstruktur. Der Syntaxbaum wird vom Parser aufgebaut und enthält nur noch Sprachkonstrukte und damit keine Kommentare mehr. Die hier verwendete Lösung für dieses Problem besteht darin, Kommentare als explizites Sprachkonstrukt einzuführen, das dann im Syntaxbaum auftaucht. Kommentare in konventionellen Programmiersprachen können überall erscheinen, wo auch Leerzeichen auftreten können. Sollte dies nachgebildet werden, müßte in der Grammatik zwischen allen Sprachkonstrukten ein Regelaufruf für das Kommentarkonstrukt stehen, z.B.

`repeat_stmt` → **repeat** `C condition C stmt_sequence C` **end**
`C` → **comment** `string` **endcomment** | ε

Die erzeugte Grammatik wird dadurch nicht übersichtlicher und einfacher zu parsen. In der vorliegenden Implementierung wurde daher die Verwendung von Kommentarstatements soweit eingeschränkt, daß sie nur noch da auftauchen können, wo auch andere Statements sein könnten. Diese Einschränkung macht sich im graphischen Editor, mit dem der Benutzer hauptsächlich arbeiten soll, jedoch nicht bemerkbar, da die einzelnen Statements sowieso als atomare grapische Objekte erscheinen und der Benutzer keine Möglichkeit hat, irgendetwas z.B. zwischen den geschweiften Klammern des `selection` Statements einzufügen. Die etwas umständlich erscheinende Klammerung der Kommentare zwischen die `comment` und `endcomment` Schlüsselwörter erleichtert das Parsen.

Der Benutzer kann mit einem Texteditor nach wie vor, wie in der Spezifikation von Bröckel vorgesehen, Kommentare im Quelltext nach # anbringen. Diese entfernt jedoch der Scanner, so daß sie nach der Bearbeitung des Quelltextes mit dem graphischen Editor nicht mehr vorhanden sind.

```
scriptdes      → #script scrptvsrn compound_stmt
scrptvsrn      → string
compound_stmt  → begin string stmt_sequence end
selection_stmt → selection string buttonspec_seq end
buttonspec_seq → buttonspec_seq button = string : stmt_sequence
                 | ε
parallel_stmt  → parbegin par_sequence parend
par_sequence   → par_sequence | stmt_sequence
                 | stmt_sequence
repeat_stmt    → repeat condition stmt_sequence end
condition      → cardinal
branch_stmt    → branch string
                 | branch calling string
comment_stmt   → comment string endcomment
stmt_sequence  → stmt_sequence statement
                 | ε
statement      → view_stmt
                 | close_stmt
                 | wait_stmt
                 | selection_stmt
                 | compound_stmt
                 | parallel_stmt
                 | repeat_stmt
                 | branch_stmt
                 | exec_stmt
                 | comment_stmt
view_stmt      → view name optionseq
optionseq      → optionseq name optionspec
                 | ε
optionspec     → optionspec name optionval
                 | ε
optionval      → letter
                 | name
                 | cardinal
                 | float
                 | ε
close_stmt     → remote close name
                 | closeall
wait_stmt      → wait cardinal
exec_stmt      → string
string         → string
remote         → device = name
                 | ε
device         → display
                 | host
name           → string
```

Abbildung 10.2: Grammatik der von VisEd unterstützten Autorensprache

Kapitel 11

Konzepte der Implementierung

11.1 Datenstrukturen für Sprachobjekte

Bei kontextfreien Grammatiken wird Wiederholung von Sprachobjekten durch Rekursion spezifiziert. Bei der Anwendung von Grammatikregeln werden linke Seiten von Regeln durch deren rechte Seiten ersetzt. Diese Folge von rekursiven Ersetzungen wird durch einen Syntaxbaum und dessen rekursive Strukturen modelliert. Schon für wenig umfangreiche Quelltexte von kontextfreien Sprachen werden die zugehörigen Syntaxbäume relativ groß (vergl. Abb. 11.10). Dies liegt daran, daß nicht nur die Blätter, die als einzige Grammatikkonstrukte ihre Entsprechung im Quelltext haben, sondern sämtliche linke Seiten von Regeln auftauchen, die beim Parsen des Quelltextes benötigt werden. Ein Syntaxbaum eignet sich damit nicht für die interaktive Manipulation durch den Benutzer. Zur Bearbeitung müßte ein Benutzer Kenntnisse über kontextfreie Grammatiken haben, um zu verstehen, wie die Ersetzung von Regeln und ihre Repräsentation im Syntaxbaum vor sich geht. Auch ist Rekursion und ihre Darstellung im Syntaxbaum ohne Informatikkenntnisse nicht verständlich.

Daher wird für die Manipulation durch den Benutzer eine andere graphische Darstellungsmethode verwendet. Es ist eine Synthese der von Nassi und Shneiderman vorgeschlagenen Boxnotation (vergl. Abschnitt 5.3) mit einer Liniennotation, wie sie in Flußdiagrammen oder im Authorware–System verwendet wird. Aus dem Syntaxbaum entsteht eine Struktur von verschachtelten Rechtecken, die als Behälter für enthaltene Listen von Sprachobjekten dienen. Deren Iteration wird nicht mehr wie in Syntaxbäumen durch Rekursion modelliert, sondern durch Verwendung von linearen Listen. Ähnlich wie in Flußdiagrammen sind innerhalb der Behälter Sprachobjekte an einer Linie angebracht. Die verschachtelten Rechtecke sind in einer Baumstruktur enthalten.

Die Knoten des Syntaxbaumes sind Objekte, die Nonterminale und Terminale der Grammatik der Autorensprache entsprechen. Die Elemente der graphischen Darstellung als Baum– und Listenstruktur dagegen sind Sprachelemente der visu-

ellen Programmiersprache. Diese kann man aber wieder wie eine konventionelle Programmiersprache definieren. Eine Spezifikation der visuellen Programmiersprache ist in Abbildung 11.1 gegeben. Hier wurde die BNF Notation verwendet, um zu verdeutlichen, daß Iterationen durch Listen repräsentiert werden. Dies drückt der Iterationsoperator der BNF Notation besser aus als die Rekursion in kontextfreien Grammatiken. Der `::=` Ersetzungsoperator ist dabei als "enthält" zu lesen.

```
veCompound       ::=   veObjectSequence
veObjectSequence ::= { veParallel | veRepeat | veSelection
                       | veCompound | veViewObject | veCloseObject
                       | veWaitObject | veExecObject |
                       | veCommentObject }
veParallel       ::= { veObjectSequence }
veRepeat         ::=   veObjectSequence
veSelection      ::= { veObjectSequence }
```

Abbildung 11.1: BNF Spezifikation der visuellen Programmiersprache

11.2 Objektmodellierung der visuellen Sprache

Die Elemente der visuellen Programmiersprache sind durch Objekte, die von der Basisklasse veObject abgeleitet werden, im Modul `vised` repräsentiert. Sie haben die Funktionalität, die Baum– und Listenstruktur aufzubauen und rekursiv die Sprachobjekte der visuellen Sprache auf dem Schirm anzuordnen. Die Objekte der veObject Hierarchie werden jedoch nicht direkt auf dem Schirm dargestellt. Jedem veObject ist ein Darstellungsobjekt auf dem Bildschirm zugeordnet, das von der Basisklasse `vvLangObj` abgeleitet ist. Objekte der Klasse `veObject` bauen die Struktur der visuellen Sprache auf. Erst ein `vvLangObj` ist ein sichtbares Fenster.

Klassennamen aus dem Modul `ptree` beginnen immer mit dem Präfix `pt`, Klassennamen aus `vised` und `visview` mit `ve` bzw. `vv`. Dies erleichtert beim Lesen des Quellcodes die Zuordnung von Klassen zu Modulen und gibt Hinweise auf die Funktion von Klassen.

11.3 Aufbau des Syntaxbaumes

`y.y` ist ein Quelltext für Yacc. Er enthält Regeln der kontextfreien Grammatik, welche die Autorensprache beschreibt. Für jedes Nonterminal der Grammatik existiert eine Klasse von syntaktischen Elementen, die im Modul `ptree` definiert sind. Häufig dienen diese Objekte nur als Behälter für Zeiger auf andere grammatische Objekte, d.h. nur als Knoten im Syntaxbaum. Eingebettet in diese Regeln sind semantische Aktionen in Form von C++ Code (vergl. Abb. 11.3).

Die semantischen Aktionen werden immer dann ausgeführt, wenn der Parsevorgang zu der Stelle gelangt ist, wo die Aktionen in die Regel eingefügt sind. Das ist genau dann der Fall, wenn die Reduktion der Nonterminals links von den semantischen Aktionen abgeschlossen ist. Im VisEd System müssen semantische Aktionen die Nonterminalen der Grammatik entsprechenden syntaktischen Objekte erzeugen, initialisieren und sie als Knoten in den Syntaxbaum einbauen. Im Kapitel über syntaxgesteuerte Übersetzung in ([Aho et al. 1986]) wird ein Verfahren skizziert, wie während des Parsevorgangs ein Syntaxbaum aufgebaut werden kann. Bottom–up Parser, wie sie von Yacc erzeugt werden, benutzen einen Stack, der Attribute der schon geparste und noch nicht für eine Reduzierung verwendete Teile des Syntaxbaumes enthält. In Abbildung 11.2 wurden die Teilbäume B und

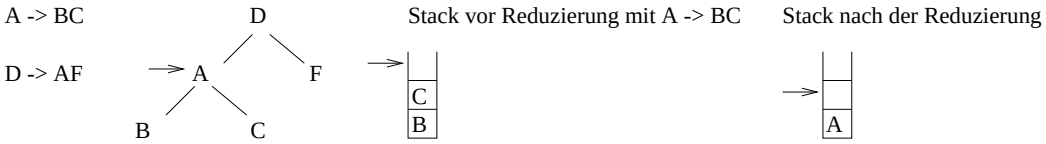

Abbildung 11.2: Stackverwendung bei Bottom-up Parsern

C schon geparst. Bevor mit der Regel `A` → `BC` reduziert wird, liegen die Attribute von B und C auf dem Stack. Bei der Reduzierung mit dieser Regel werden die Attribute von B und C vom Stack genommen und aus diesen Attributen neue Attribute für A erzeugt und auf den Stack gelegt. Realiter enthält der Stack nur Zeiger auf die Attribute. Der Stack kann recht klein sein, da bei LR Parsern und Grammatiken, die Linksrekursion enthalten, höchstens Platz für die rechte Seite der längsten Regel gebraucht wird.

Bei der vorgestellten Art von Reduzierung von kontextfreien Regeln fällt auf, daß dabei die Stackhöhe meist abnimmt oder bei Reduzierung mit Regeln der Form `X` → `Y` gleichbleibt. Es stellt sich die Frage, wann der Stack überhaupt wächst. Diese Frage wird bei [Aho et al. 1986] nicht beantwortet. Der Stack wächst, wenn beim Parsen Terminale der Grammatik angetroffen werden. Auch für ε–Produktionen müssen Attribute auf den Stack gelegt werden, sonst wird der Stack bei den Reduzierungen von Regeln nicht korrekt abgebaut.

Beim Parser von VisEd werden keine Attribute von Grammatikobjekten verwendet, sondern nur Zeiger auf die erzeugten Grammatikobjekte auf den Stack gelegt. Der Stack ist als Klasse implementiert.

```
stmt_sequence: stmt_sequence statement
                {
  ptStmtSequence *current = new ptStmtSequence;
  current->stmt = (ptStatement *)pstack.pop();
  current->seq  = (ptStmtSequence *)pstack.pop();
  pstack.push(current);
                }
                |
                {
  pstack.push(new ptEpsilon);
                }
                ;
```

Abbildung 11.3: Yacc Regel für Statementsequenz

Abbildung 11.3 enthält einen Auszug der Yacc Grammatik mit den zugehörigen semantischen Aktionen.

11.4 Umwandlung des Syntaxbaumes in Quellcode und in visuelle Darstellung

Das Modul `ptree` stellt Klassen von syntaktischen Elementen bereit, die den Symbolen der Grammatik der Autorensprache entsprechen. Objekte dieser Klassen bilden die Knoten des intern aufgebauten Syntaxbaums. Die Funktionalität dieser Klassen besteht darin, aus dem Syntaxbaum eine Baum– Listenstruktur von Oberflächenobjekten aufzubauen. Außerdem hat jedes Objekt die Funktionalität, Quelltext der Autorensprache zu erzeugen. Fügt der Benutzer auf der Oberfläche Graphikobjekte hinzu oder entfernt sie, werden die Objekte des `ptree` Moduls benachrichtigt und müssen entsprechende Änderungen im Syntaxbaum durchführen. Schließlich sind alle Objekte in der Lage, für Debugzwecke einen String mit ihrem Bezeichner auszugeben. Dadurch kann während Testläufen leicht erkannt werden, auf welches Objekt ein Zeiger verweist, indem ein Statement der Form `pointer->ident()` in den Quelltext eingefügt wird.

Die `printout` Methode schreibt auf einen Strom die Terminale aus der rechten Seite der zugehörigen Grammatikregel. Treten auch noch Nichtterminale auf, wird an die entsprechenden Grammatikobjekte die `printout` Nachricht geschickt. In Abbildung 11.4 ist der Code für die `printout` Methode des Compound Statements dargestellt. Zuerst wird das Terminal `begin` ausgegeben. Das

```
void ptCompoundStmt::printOut()
{
tabout();outstream << "begin\n";
tab_pos+=5;
seq->printOut();
tab_pos-=5;
tabout();outstream << "end\n";
}
```

Abbildung 11.4: printout Methode eines Grammatikobjektes

Klassenmember `seq` enthält einen Zeiger auf das von `compound_stmt` abhängige `statement_seq` Objekt und damit auf den davon abhängigen Rest des Syntaxbaumes. An diesen Zeiger wird jetzt die `printout` Nachricht geschickt. Rekursiv wird jetzt der von `compound_stmt` ausgehende Teil des Syntaxbaumes ausgegeben. Zuvor wird noch die Tabulatorposition erhöht, um eine gewisse Formatierung des Quelltextes zu erreichen.

11.5 Erzeugung der visuellen Darstellung

Die `createVisual` Methoden der Grammatikobjekte erzeugen die entsprechenden Objekte der visuellen Programmiersprache. Hierzu muß die Baum– und Listenstruktur aufgebaut werden, welche die Oberflächenobjekte enthält. Dieser Vorgang ist relativ komplex.

Für die Implementierung der linearen Listen wurde eine Listenklasse aus der Gnu libg++ Library ([Lea 1992]) verwendet, die Teil des Gnu C++ Entwicklungspaketes ist. Die Klasse enthält die üblichen Operationen zur Listenmanipulation. Durch die Verwendung von C++ Templates kann die Listenklasse für Listenelemente beliebigen Types instanziiert werden. Dies erwies sich als sehr nützlich, da sowohl Listen von Sprachobjekten als auch Listen dieser Listen benötigt wurden. Durch den Template Mechanismus konnten für beliebige Listenarten die gleiche Klasse benutzt werden.

Die visuelle Programmiersprache, also die Menge der auf dem Bildschirm dargestellten Sprachobjekte und ihre Verknüpfung, besteht aus atomaren Sprachobjekten, die als Icons repräsentiert werden, und Behältern. Diese sind Flächenobjekte, die eine Liste von atomaren Objekten enthalten. Container können ver-

schachtelt werden. Das wichtigste Objekt der visuellen Programmiersprache ist die Sequenz, ein unverzweigter Ablaufpfad. Die Klasse `veObjectSequence` modelliert die Sequenz. Behälter enthalten immer mindestens eine Sequenz. Das Compoundstatement und das Repeatkonstrukt enthalten eine Sequenz, das Parallel– und das Selektionskonstrukt dagegen können mehrere Pfade enthalten. Dies wird durch eine Liste von Sequenzen repräsentiert, also eine Liste von Listen von Sprachobjekten, die atomare Objekte oder Behälter sind.

Die Struktur von verketten Listen aus Objekten der visuellen Programmiersprache wird bei einem Top-Down Durchlauf durch den Syntaxbaum der Autorensprache aufgebaut. Dies geschieht in der Methode `createVisual`. Sie erzeugt Objekte der visuellen Sprache und legt sie in einer Liste ab, die den Returnwert der Methode darstellt. Da der Syntaxbaum rekursiv aufgebaut ist, erfolgen die Aufrufe von `createVisual` ebenfalls rekursiv. Äußere Inkarnationen der `createVisual` Methode fügen die von inneren Inkarnationen erhaltenen Listen zusammen. Dabei muß bei den Grammatikobjekten der Autorensprache differenziert vorgegangen werden, um die Sprachobjekte der visuellen Sprache zu erzeugen. Für Blätter des Syntaxbaumes, die elementare Statements sind, also die `view`, `close`, `wait`, `execute` und `comment` Statements, muß ein atomares Objekt der visuellen Sprache angelegt werden. Auf dem Bildschirm werden diese Statements durch Icons dargestellt. Die Methode liefert hierfür eine einelementige Liste zurück, die einen Zieger auf das erzeugte visuelle Objekt enthält. Für ε–Blätter des Syntaxbaumes wird eine leere Liste zurückgeliefert. Vom compound Statement hängt in der Grammatik der Autorensprache eine Statementsequenz ab.

```
compound_stmt → begin stmt_sequence end
stmt_sequence → stmt_sequence statement
              | ε
```

Die `stmt_sequence` Regel der Grammatik findet in der visuellen Programmiersprache keine direkte Entsprechung, da Rekursion nicht auf dem Schirm dargestellt wird. Die Sequenz wird durch ein `veObjectSequence` Objekt repräsentiert, das eine Liste der Statements enthält, die von der Statementsequenz abgeleitet werden (vergl. Abb. 11.5). Containerobjekte enthalten immer eine solche Objektsequenz. Z.B. erzeugt die `createVisual` Methode für `ptRepeat` zuerst ein veRepeat Objekt und damit auf dem Bildschirm das umgebende Rechteck. Für die Sequenz wird eine `veObjectSequence` erzeugt und in das Containerobjekt `veRepeat` angebracht. Nun muß noch die Liste der von `veObjectSequence` abhängigen Objekte erzeugt werden. Dies geschieht durch rekursive Aufrufe der `createVisual` Methode für die Statementsequenz. Dabei müssen die von inneren Inkarnationen erhaltenen Objektlisten verbunden werden und an die nächstäußere Instanz zurückgeliefert werden (vergl. Codebeispiel in Abb. 11.6).

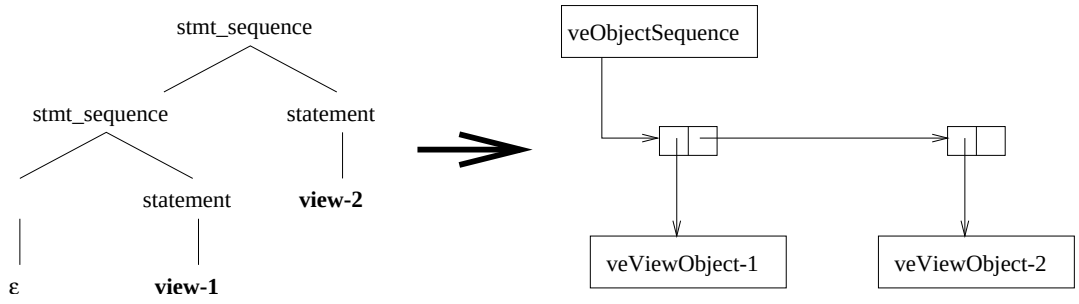

Abbildung 11.5: Umwandlung von Linksrekursion in eine Sequenz

```
veObjectDLList ptStmtSequence::createVisual(ptSyntElem* parent)
{
 veObjectDLList l1,l2;

 l1=seq->createVisual(parent);
 l2=stmt->createVisual(parent);
 l1.join(l2);
 return(l1);
}
```

Abbildung 11.6: Code für die Umwandlung einer Rekursion in die Sequenz

11.6 Layout der visuellen Sprache

Das `vised` Modul enthält die Definitionen für die Objekte der visuellen Programmiersprache. Die Methoden dieser Objektklassen haben die Aufgabe, den Flächenbedarf der Bildschirmrepräsentationen zu ermitteln und anhand dessen die Screenviews anzuordnen. Schließlich muß noch die Darstellung auf dem Bildschirm erfolgen und auf Benutzeraktionen zum Hinzufügen oder Entfernen eines Sprachobjektes reagiert werden. Die Views der Sprachobjekte, also deren Bildschirmdarstellung, sind im Modul `visview` definiert. Die Initialisierung der Sprachelemente erzeugt auch ein Viewobjekt auf dem Bildschirm, diese werden jedoch erst nach Aufruf der `showView` Methode sichtbar.

Die `needSpace` Methode liefert den Flächenbedarf eines Objektes auf dem Bildschirm. Für atomare Objekte ist dies die Größe des zugehörigen Icons, für Behälter muß dagegen rekursiv die Größe der Kinder abgefragt werden. In Abbildung 11.7 ist die `needSpace` Methode für eine Objektsequenz dargestellt. In

```
void veObjectSequence::needSpace(int& w,int& h)
{
  int count=0,maxwidth=0,sumheight=0,wc,hc;

  if (childrenList.empty()) {
    w=h=0;
    return;
  }
  for (Pix p = childrenList.first(); p != 0; childrenList.next(p))
    {
      count++;
      childrenList(p)->needSpace(wc,hc);
      maxwidth=max(maxwidth,wc);
      sumheight+=hc;
    }
  w=maxwidth;
  h=(count-1)*vdist+sumheight;
}
```

Abbildung 11.7: Ermittlung des Platzbedarfs einer Sequenz

der `for` Schleife wird die Liste der Objekte der Sequenz durchlaufen und dabei deren `needSpace` Methode aufgerufen. Die Breite der Gesamtsequenz ergibt sich aus der des größten Kindes, die Höhe aus der Summe der Höhe der Kinder.

Nachdem der Platzbedarf des Baumes von visuellen Sprachelementen festgestellt ist, werden die einzelnen Elemente von oben beginnend an ihren Platz gesetzt. Erst jetzt werden die Elemente sichtbar. Container befinden sich dabei hinter ihren Inhalten und werden von ihnen verdeckt.

11.7 Visualisierung der beschriebenen Schritte

In Abbildungen 11.8 – 11.11 werden die Schritte von der Autorensprache bis zur Bildschirmdarstellung visualisiert.

```
compound_stmt   ->   begin stmt_sequence end
stmt_sequence   ->   stmt_sequence statement |  ε
statement       ->   view_stmt | parallel_stmt | selection_stmt
view_stmt       ->   view
parallel_stmt   ->   parbegin parsequence parend
par_sequence    ->   parsequence | stmt_sequence | stmt_sequence
selection_stmt  ->   selection string buttonspec_seq end
buttonspec_seq  ->   buttonspec_seq button = string : stmt_sequence |  ε
```

```
begin
    view-1
    parbegin
        selection
            button = "Wahl 1" : view-2
            button = "Wahl 2" : view-3
        end
        |
        view-4
        view-5
    parend
end
```

Abbildung 11.8: Auszug aus der Autorensprache und Codebeispiel (Grammatik vereinfacht)

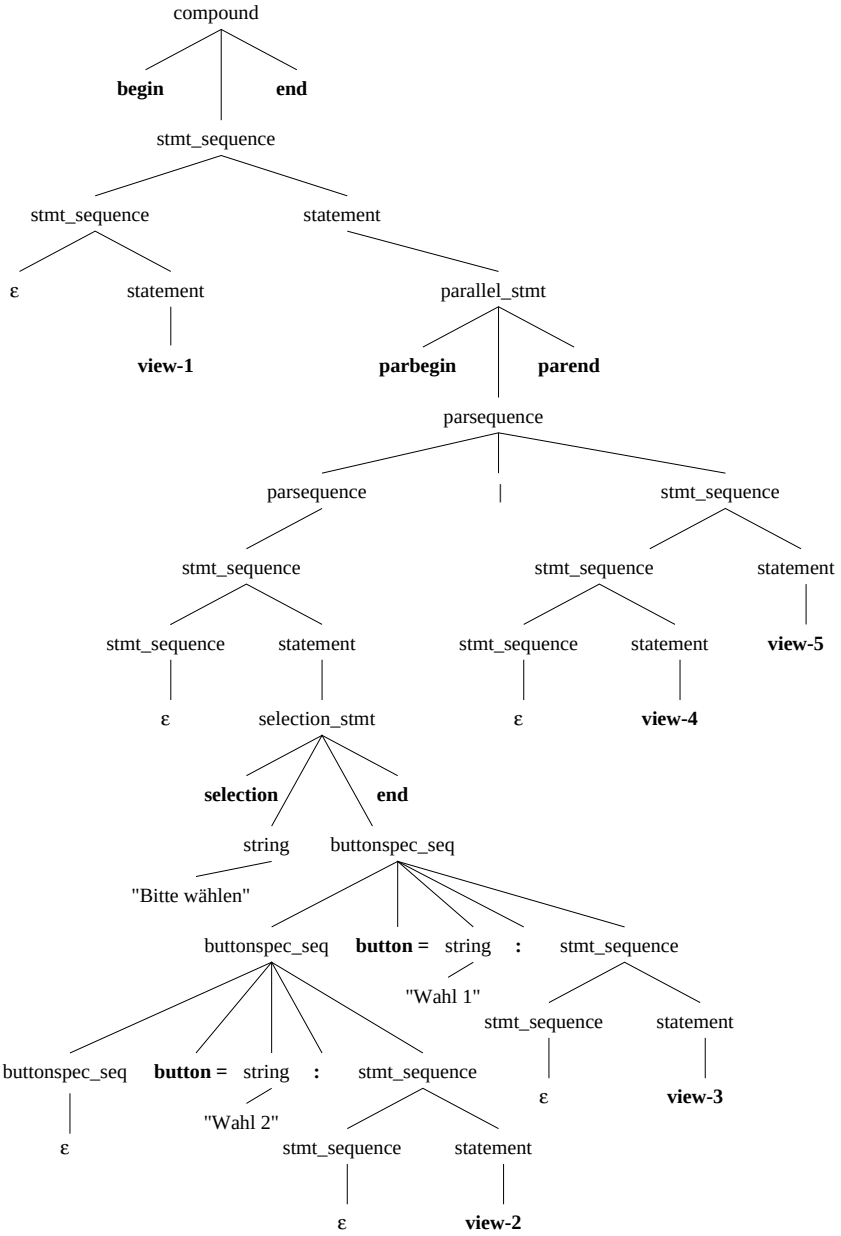

Abbildung 11.9: Repräsentation des Codebeispiels als Syntaxbaum

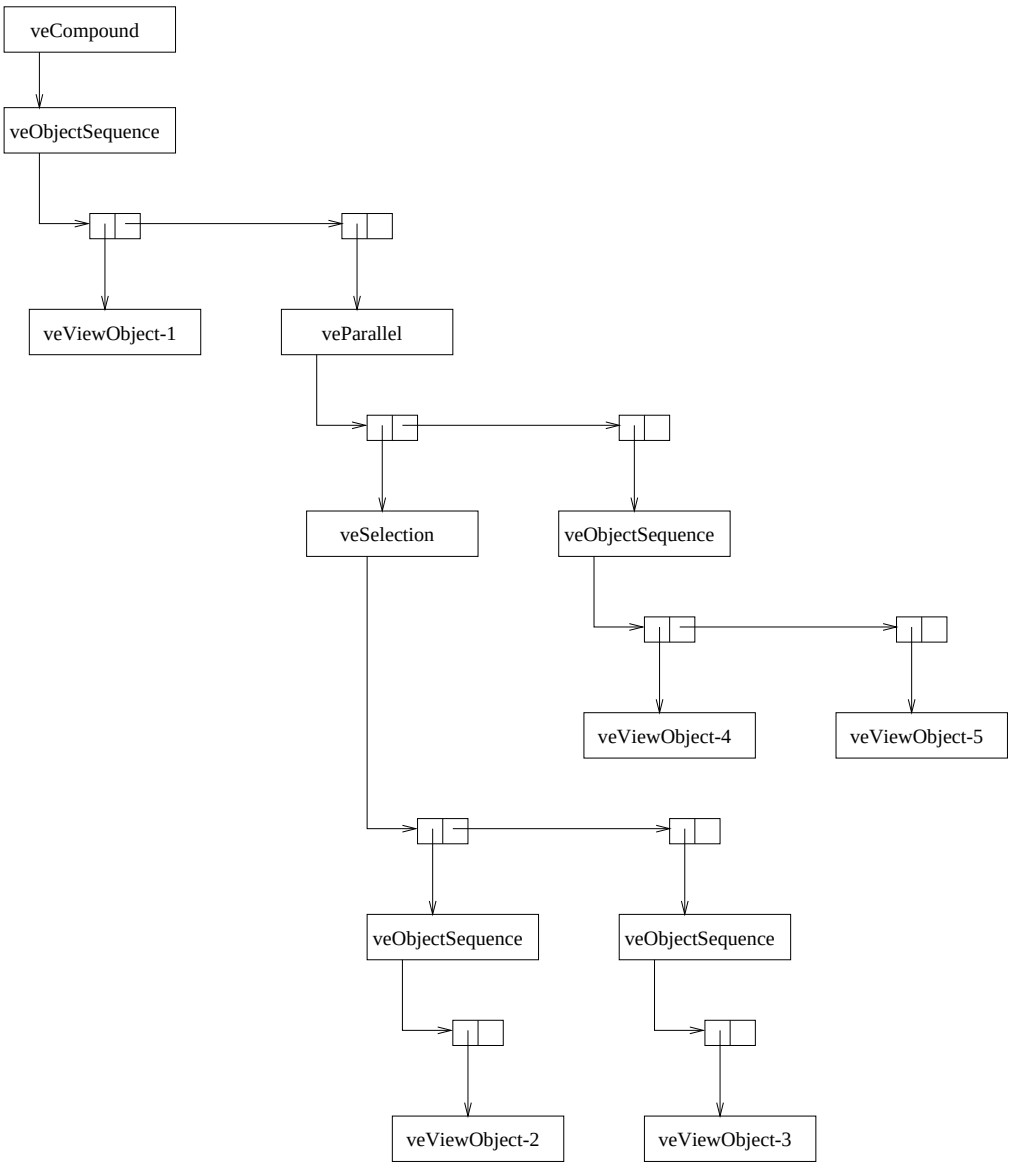

Abbildung 11.10: Repräsentation des Codebeispiels als Baum der visuellen Sprache

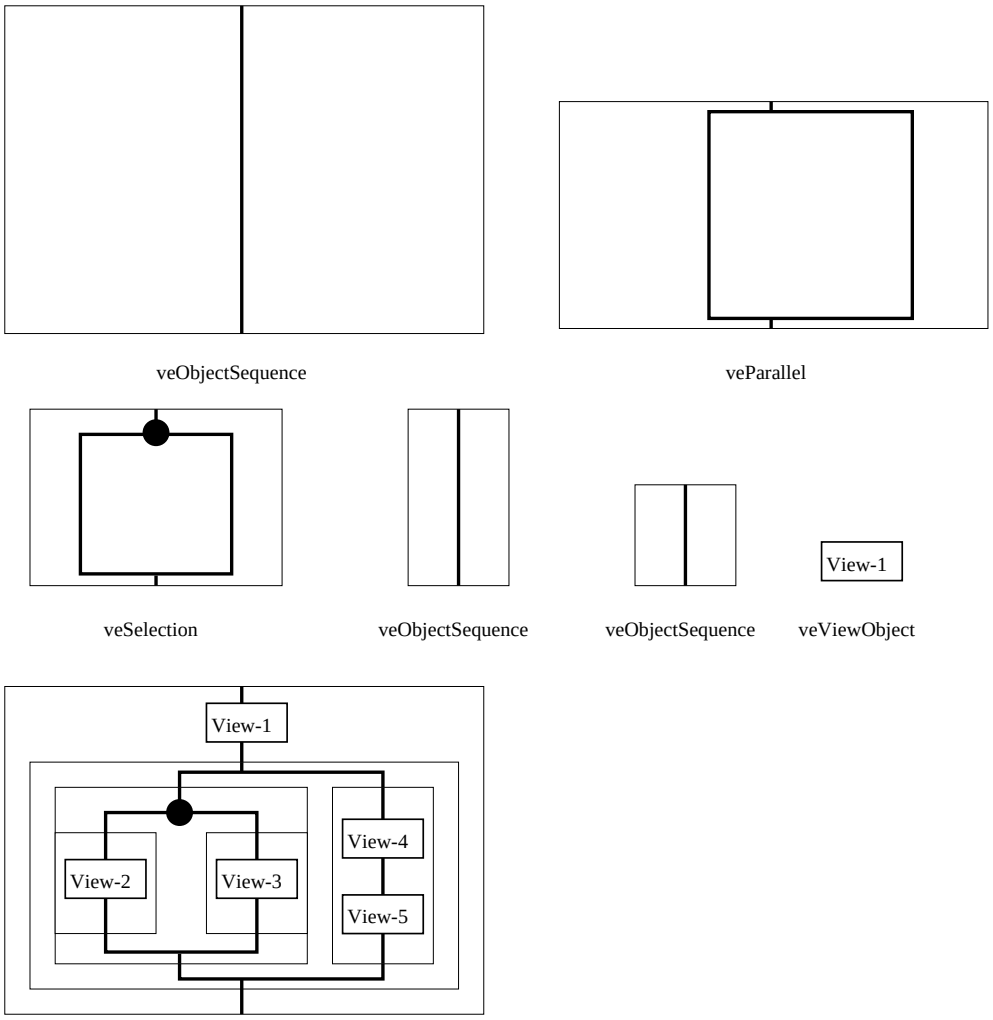

Abbildung 11.11: Darstellung der visuellen Sprache

Kapitel 12

Fazit und Ausblick

In dieser Arbeit wurde ein Konzept für einen graphischen Editor entwickelt, mit dem multimediale Präsentationen erstellt werden können. Einsatzzweck der Präsentationen ist die Informationsdarstellung im betrieblichen Umfeld. Der Editor stellt eine Menge von Sprachobjekten bereit, die von einer Palette auf die Arbeitsfläche gezogen werden können. Durch die Verwendung von Drag und Drop ist der Editor sehr einfach bedienbar und kann auch von Gelegenheitsnutzern wie Sachbearbeitern benutzt werden. Der Editor unterstützt eine strukturierte Erstellung von Präsentationen. Auch kann durch die Einführung von Compound–Statements modular gearbeitet werden.

Es zeigte sich bei der Entwicklung des graphischen Editors, daß durch mächtige Toolkits die Erstellung von graphischen Benutzeroberflächen relativ einfach möglich ist. Damit kann sich der Softwareingenieur auf die Erstellung der inneren Funktionalität von Software konzentrieren. Den größten Aufwand bei der Entwicklung des graphischen Editors VisEd forderte die Programmierung der Funktionalität zur Verwaltung eines Syntaxbaumes und dessen Umwandlung in eine geeignete graphische Repräsentation.

In dieser Arbeit wurde gemäß der Aufgabenstellung nur eine prototypische Implementierung des visuellen Editors erstellt. Bei einer möglichen Weiterentwicklung sollten noch folgende Punkte geändert bzw. ergänzt werden:

- Das Multimedia Informations System sollte darauf eingerichtet werden, über Interprozeßkommunikation Quellcode vom graphischen Editor VisEd einzulesen. Damit könnte aus dem graphischen Editor heraus eine Präsentation getestet werden. Der Editor bietet schon jetzt die Möglichkeit, auch Teile der erstellten Präsentation zu sichern. Diese Funktionalität könnte so erweitert werden, daß eine solche Teilpräsentation gleich getestet werden könnte.

- Ebenfalls sollte eine solche Interprozeßkommunikation möglich sein, um zu bestimmen, welche Daten ein view–Statement darstellen soll.

- Um die Arbeitsfläche, in der die visuellen Präsentationen erstellt werden, sollten Scrollbars gelegt werden. Damit wäre auch die Erstellung umfangreicher Präsentationen möglich, da dann der aktuell sichtbare Arbeitsbereich verschoben werden könnte.

- In der vorliegenden prototypischen Implementierung wurden nur primitive Icons zur Repräsentation der Sprachmittel erstellt. Bei einer Weiterentwicklung des Editors sollten Icons von ansprechenderem Aussehen und höherer Aussagekraft nach den in Abschnitt 5.4 vorgestellten Prinzipien des Designs von Icons entwickelt werden.

- Wünschenswert wäre auch, wenn die Icons, die eine Darstellung von Medien repräsentieren, eine optische Repräsentation des Mediums wären. Bei einem Icon, das die Darstellung einer Graphik repräsentiert, wäre das eine Verkleinerung dieser Graphik. Bei Filmen wäre es eine Verkleinerung eines Bildes aus einer typischen Scene. Natürlich ist ein solches Verfahren nur bei visuellen Medien sinnvoll.

- Ähnlich wie graphische Desktops sollte der Editor ein Papierkorb Icon enthalten. Sprachobjekte könnten dann gelöscht werden, indem sie auf den Papierkorb gezogen werden.

- Die Implementation sollte auf den Motif++ Toolkit (vergl. Abschnitt 7.1.5) umgestellt werden, da zweifelhaft ist, ob der in dieser Arbeit verwendete Xm++ Toolkit noch weiterentwickelt und gepflegt wird.

Literaturverzeichnis

[Aho et al. 1986] Aho, A.V., Sethi, R., Ullman, J.D., Compilers: Principles, Techniques and Tools, Reading 1986

[Beil 1993] Beil, M., Einführung in die Programmierung mit OSF/Motif In: iX 1/1993 und iX 2/1993

[Belady et al. 1980] Belady, L.A., Evangelisti, C.J., Power, L.R. Greenprint: A graphic representation of structured programs. In: IBM Systems Journal, Vol. 19, Number 4, 1980, wiederveröffentlicht in [Glinert 1990a]

[Bücker et al. 1993] Bücker, M. C., Geidel, J., Lachmann, M. F., Objectworks
Smalltalk, Berlin, Heidelberg 1993

[Bröckel 1994] Bröckel, S., Multimediale Informationsaufbereitung, Diplomarbeit, Universität Stuttgart, Fakultät Informatik, 1994

[Drapeau 1991] Drapeau, G. D., MAEstro – A Distributed Multimedia Environment. In: Proceedings of the 1991 Summer USENIX Conference, Nashville 1991

[Drapeau 1993] Drapeau, G. D., Synchronization in the MAEstro Multimedia Authoring Environment, In: ACM Multimedia 6/93

[Eirund, Hofmann 1993] Eirund, H., Hofmann, M., Designing Multimedia Presentations, In: Procedings der Internationalen Hypermedia '93 Konferenz, Belin, Heidelberg 1993

[Fisher 1994] Fisher, S., Multimedia Authoring: Building and Developing Documents, Boston 1994

[Gaines, Shaw 1993] Gaines, B. R., Shaw, M. L. G., Open Architecture Multimedia Documentes, In: ACM Multimedia 6/93

[Glinert, Tanimoto 1984] Glinert, E. P., Tanimoto, S. L., Pict: An Interactive Graphical Programming Environment, Computer, Nov. 1984, wiederveröffentlicht in [Glinert 1990a]

[Glinert 1990a] Glinert, E. P., Visual Programming Einvironments: Paradigms and Systems, Los Alamitos 1990

[Glinert 1990b] Glinert, E. P., Visual Programming Einvironments: Applications and Issues, Los Alamitos 1990

[Gloor 1990] Gloor, P. A., Hypermedia–Anwendungsentwicklung, Stuttgart 1990

[Götze 1994] Götze, R.: Dialogmodellierung für multimediale Benutzerschnittstellen. - Dissertation, Universität Oldenburg, Fachbereich Informatik, 1994

[Gottheil et al. 1992] Gottheil, K., Kaufmann, H.–J., Kern, T., Zhao, R., X und Motif, Einführung in die Programmierung des Motif–Toolkits und des X–Window–Systems, Berlin, Heidelberg 1992

[Heller 1990] Heller, D., XView Programming Manual: An OPEN LOOK Toolkit for X11, Sebastopol 1990

[Herczeg et al. 1992] Herczeg, J., Hohl, H., Ressel, M., Progress in Building User Interface Toolkits: The World According to XIT, In: Proceedings of the ACM Symposium on User Interface Software and Technology, 1992

[Herczeg 1994] Herczeg, M., Software-Ergonomie: Grundlagen der Mensch–Computer–Kommunikation, Bonn 1994

[Hirakawa, Ichikawa 1994] Hirakawa, M., Ichikawa, T., Visual Language Studies – A Perspective, In: Software–Concepts and Tools, 15, 1994

[Hühne 1994] Hühne, M., Gnu–Software auf System V, Release 4, In: iX 11/1994

[Inwood et al. 1994] Inwood, B., Connell, I., Scane, R., Collyer, J., Christie, B., Ashcroft, V., Brown, I., Knight, J., The information content and user driven aspects of the ACT–IT multimedia authoring system, In: Information and Software Technology, Vol. 36, Nr. 5, 1994

[Kaehler 1988] Kaehler, C., HyperCard Power: Techniques and Scripts, Reading 1988

[Krasner, Pope 1988] Krasner, G. E., Pope, S. T., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, In: Journal of Object-Oriented Programming, Vol. 1, Nr. 3, 1988

[Krienke 1994] Krienke, R., C++ kurzgefaßt: eine Einführung in C++, Mannheim 1994

[Lea 1992] Lea, D., User's Guide to the Gnu C++ Library, Free Software Foundation 1992

[van Loon 1993] Looking at Motif in an Object–Oriented Way: Motif++, In: 3D Computervision, Utrecht 1993

[Mason, Brown 1990] Mason, T., Brown, D., lex & yacc, Sebastapol 1990

[Microsoft Press 1991] Microsoft Press, Microsoft Windows Multimedia Programming Workbook, Redmond 1991

[Myers 1986] Myers, B. A., Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In: Conference Proceedings, CHI '86: Human Factors in Computing Systems, ACM 1986, wiederveröffentlicht in [Glinert 1990a]

[Nassi, Shneiderman 1973] Nassi, I. und Shneiderman, B., Flowchart Techniques for Structured Programming, In: SIGPLAN NOTICES 8,8, 1973, wiederveröffentlicht in [Glinert 1990a]

[Paxson 1993] Paxson, V., Flexdoc: Manpages zu Gnu Flex, Free Software Foundation 1993

[Pong, Ng 1983] Pong, M.C., Ng, N., PIGS–A System for Programming with Interactive Graphical Support, In: Software – Practice and Experience, Vol. 13, Number 9, 1983, wiederveröffentlicht in [Glinert 1990a]

[Pountain 1994] Pountain, D., Starting with a Clean Sheet, In: Byte 11/1994

[Roller et al. 1995] Roller, D., Bihler, M., Stolpmann, M., Adaptive hypermediale Informationsaufbereitung in betrieblichen Informationssystemen, Institut für Informatik, Universität Stuttgart 1995

[Ruhland 1990] Ruhland, R., Beschreibungsmethoden für graphische Benutzeroberflächen am Beispiel X-Windows, Studienarbeit, Universität Stuttgart, Fakultät Informatik, 1990

[Scheifler et al. 1988] Scheifler, R. W., Gettys, J., Newman, R., X-Window System, C Library and Protocol Reference, Digital Press, 1988

[Schreyjak 1994] Schreyjak, S., Analyse von Multimedia Autorensystemen anhand des Beispiels MAEstro, Studienarbeit, Universität Stuttgart, Fakultät Informatik, 1994

[Sedgewick 1988] Sedgewick, R., Algorithms, Reading 1988

[Shneiderman 1983] Shneiderman, B., Direct Manipulation: A Step Beyond Programming Languages, In: Computer, Aug. 1983, wiederveröffentlicht in [Glinert 1990b]

[Shneiderman 1992] Shneiderman, B., Designing the User Interface, Reading 1992

[Shu 1988] Shu, N. C., Visual Programming, New York 1988

[Stallman 1993] Stallman, R. M., Using and Porting Gnu CC, Free Software Foundation 1993

[Steinbrink 1993] Steinbrink, B., Multimedia–Regisseure: Autorensysteme und –werkzeuge im Vergleich, In: c't 10/1993

[Steinmetz 1993] Steinmetz, R., Multimedia–Technologie: Einführung und Grundlagen, Berlin, Heidelberg 1993

[Strassl, Binder 1994a] Strassl, B., Binder, S., The Xm++ User's Guide, Institut für angewandte Informatik und Informationssysteme, Universität Wien 1994

[Strassl, Binder 1994b] Strassl, B., Binder, S., Xm++ Class Reference Manual, Institut für angewandte Informatik und Informationssysteme, Universität Wien 1994

[Tripp 1988] Tripp, L. L., A Survey of Graphical Notations for Program Design: An Update, In: ACM SIGSOFT Software Engineering Notes, Vol. 13, Number 4, 1988, wiederveröffentlicht in [Glinert 1990a]

[Wood, Wood 1987] Wood, T. W., Wood, S. K., Icons in everyday life, In: Human-Computer-Interaction – INTERACT '87, wiederveröffentlicht in [Glinert 1990b]