Ein Generator zur Entwicklung visueller Sprachen

Dissertation

Schriftliche Arbeit zur Erlangung des akademischen Grades
„Doktor der Naturwissenschaften"
im Fachbereich Mathematik-Informatik der Universität Paderborn

vorgelegt von

Matthias Jung

Paderborn, November 2000

Vorwort

Viele liebe Menschen haben es mir durch Ihre Unterstützung ermöglicht, diese Arbeit erfolgreich abzuschliessen. Bei allen bekanke ich mich, auch wenn ich nur einige wenige hier namentlich nennen kann.

An erster Stelle bedanke ich mich bei meiner Frau Hana und bei meiner Tochter Elena Madeleine. Beide haben mir durch Ihre Gegenwart und Liebe die doch manchmal frustrierende Forschungsarbeit erst ermöglicht. Hana hat mir darüberhinaus in den schwierigen Zeiten beigestanden, unter anderem indem sie nie die Hoffnung auf einen erfolgreichen Abschluß der Arbeit aufgegeben und meinen für sie oft nur schwer verständlichen Erläuterungen zugehört hat.

Uwe Kastens hat mich vor neun Jahren ins Eli-Team, und vor fünf Jahren in seine Arbeitsgruppe aufgenommen. Er hat den Anstoß zur Beschäftigung mit der Implementierung visueller Sprachen gegeben. Meine Dankbarkeit gilt dem produktiven Arbeitsklima, seiner Kritik, seinen Anregungen und seinen Korrekturvorschlägen. Dadurch hat er geholfen, diese Dissertation in Form und Inhalt zu verbessern, auch wenn dies nicht immer angenehm war.

Zusammen mit ihm hat Gerd Szwillus diese Arbeit begutachtet. Auch Gerd hat wertvolle Tips zur Verbesserung der Arbeit geliefert. Die Gespräche mit ihm haben mir außerdem Zuversicht gegeben, wenn ich ihrer bedurfte.

Die Mitglieder der Arbeitsgruppe, insbesondere Carsten Schmidt und Michael Thies haben mir im weiteren durch interessante Diskussionen zum Thema der Arbeit in wichtigen Augenblicken den notwendigen Dialog ermöglicht. Die Gespräche haben mir geholfen, die oft komplexe Materie zu durchdringen. Sie waren für mich oft nicht nur klärend, sondern auch entspannend und ermunternd.

Durch ihren unermüdlichen Einsatz bei der systematischen Implementierung der Spezifikationsmodule haben Carsten Schmidt und Christian Schindler die praktische Umsetzung des VLEli-Systems in besonderer Weise unterstützt. Die Zusammenarbeit mit den beiden war in jeder Phase interessant und beflügelnd.

Matthias Jung

Inhaltsverzeichnis

1. Einführung

Visuelle Sprachen sind Sprachen, die visuelle Notationen benutzen, um Programmkonstrukte und die Beziehungen zwischen Programmkonstrukten darzustellen. In der Softwareerstellung werden sie erfolgreich unter anderem für die Programmierung (Prograph), den Softwareentwurf (UML) und für die Spezifikation verteilter Systeme (SDL) verwendet. Weitere Einsatzgebiete sind etwa die Beschreibung von 3D-Modellen und ihre Animation (SAM) und die Abfrage von Datenbanken (QBE).

Visuelle Sprachen haben Vorzüge gegenüber textuellen Sprachen. Zum einen liefern sie zusätzliche Möglichkeiten, um die inneren Beziehungen und Strukturen eines Anwendungsbereichs darzustellen. Informationen können so prägnanter dargestellt werden, als dies mit textuellen Notationen möglich ist. Whitley [1997] belegt mit Messungen, daß dadurch das Verständnis eines Anwenders, insbesondere bei komplexen Fragestellungen verbessert werden kann. Zum anderen können in einer visuellen Sprache anwendungsspezifische Symbole und Strukturierungstechniken verwendet werden. Visuelle Notationen eines Anwendungsbereichs können so für eine visuelle Sprache genutzt und eine größere Vertrautheit der Anwender mit der Sprache erzielt werden. Die von Nardi [1993, S. 37-39] angeführten Untersuchungen zeigen, daß damit die Motivation eines Anwenders und sein Verständnis der Sprachkonstrukte wesentlich verbessert werden.

Um eine visuelle Sprache zu entwerfen und zu implementieren, sind umfangreiche Kenntnisse zahlreicher Gebiete erforderlich. Beim Sprachentwurf muß eine *visuelle Notation* konzipiert werden. Vor allem von der Notation einer visuellen Sprache hängt die Verständlichkeit der damit implementierten Programme und die Tauglichkeit der Sprache für Projekte realistischer Größe ab. Weiteres Know-How ist erforderlich, um einen *spezialisierten Editor* für eine visuelle Sprache zu erstellen. Vor allem die Editieroperationen und das Verfahren, mit dem das Layout der Sprachkonstrukte realisiert wird, bestimmen in hohem Maße die Benutzbarkeit der Sprachimplementierung [Green und Petre 1996]. Um Werkzeuge zu erstellen, mit denen visuelle Programme analysiert und weiterverarbeitet werden können, sind ferner Kenntnisse aus dem Gebiet der *Konstruktion von Übersetzern* erforderlich.

Der Beitrag dieser Arbeit zur Forschung im Bereich der Informatik ist eine neuartige Methode, mit der Entwicklungsumgebungen für visuelle Sprachen generiert werden können [Jung u.a. 2000]. Die Methode wurde durch das im Rahmen dieser Arbeit erstellte Werkzeugsystem VLEli umgesetzt und vereinigt drei wichtige Eigenschaften: Sie *kapselt das Know-How*, das zum Entwurf und zur Implementierung visueller Sprachen benötigt wird, eignet sich für eine *große Klasse visueller Sprachen* und stellt *allgemeine Techniken der Sprachimplementierung* zur Verfügung, mit denen Werkzeuge für die Analyse und Weiterverarbeitung visueller Programme generiert werden können. Durch die Kapselung der Entwurfs- und Implementierungskenntnisse können visuelle Sprachen mit ausdrucksstarken Spezifikationen niedriger Komplexität beschrieben werden. Das ermöglicht es einerseits auch Nichtspezialisten, visuelle Sprachen zu implementieren. Andererseits können so Designexperimente und Sprachänderungen mit angemessen niedrigem Aufwand durchgeführt werden. Darüberhinaus bietet die Methode ein breites Spektrum einsetzbarer Sprachelemente und -konstrukte, verschiedene Verfahren der Layouterzeugung und weitgehenden Einfluß auf die Editieroperationen, so daß sich benutzerfreundliche Editoren für ein weites Spektrum visueller Sprachen implementieren lassen.

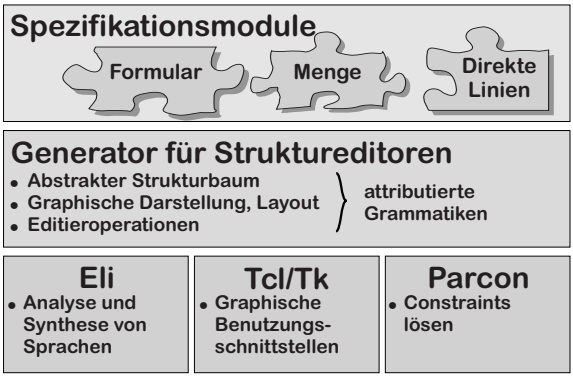


Abbildung 1.1.: Die drei Ebenen in VLEli

Das Werkzeugsystem VLEli besteht aus drei Ebenen. Eine Ebene benutzt jeweils die Funktionalität der darunterliegenden Ebene und erhöht das Abstraktionsniveau der Ausdrucksmittel, die verwendet werden können, um visuelle Sprachen zu spezifizieren. Auf der untersten Ebene befinden sich einzelne Werkzeuge, mit denen jeweils Teile einer visuellen Sprache implementiert werden können. Die darauf aufbauende Ebene integriert diese Werkzeuge und kann Entwicklungsumgebungen für visuelle Sprachen aus Spezifikationen generieren. Darauf baut eine

dritte Ebene auf, in der Module angewendet werden können, die Spezifikationen für wichtige Elemente und Konstrukte visueller Sprachen kapseln. Die Architektur von VLEli habe ich in Abbildung 1.1 skizziert.

Um den Entwurf visueller Sprachen zu vereinfachen, werden in dieser Arbeit *visuelle Muster* eingeführt. Analog zu den Entwurfsmustern von Gamma u.a. [1996] beschreiben visuelle Muster das wiederverwendbare Entwurfs-Know-How einer visuellen Sprache, indem sie wesentliche Eigenschaften eines Sprachkonstrukts und seiner graphischen Darstellung definieren. Die visuellen Muster wurden im Rahmen einer Untersuchung acht visueller Sprachen erarbeitet. Sie lassen sich vom Sprachentwickler anhand der Struktur einer Sprache identifizieren und umfassen wichtige, vielfach eingesetzte Verfahren für die Visualisierung von und Interaktion mit Informationen. Die visuellen Muster sind ein methodischer Fortschritt für den Entwurf visueller Sprachen.

Für die Implementierung visueller Sprachen wurden in dieser Arbeit *Spezifikationsmodule* entwickelt. Spezifikationsmodule sind flexibel einsetzbare, miteinander kombinierbare Module wiederverwendbarer Spezifikationen, die Implementierungsalternativen für visuelle Muster bereitstellen. Eine visuelle Sprache läßt sich mit ihnen beschreiben, indem für die visuellen Muster jeweils passende Spezifikationsmodule ausgewählt werden. Das VLEli-System kann daraus automatisch eine auf die Sprache spezialisierte Entwicklungsumgebung generieren. Die Spezifikationsmodule kapseln das zur Sprachimplementierung erforderliche Know-How, lassen sich aber auch weitgehend an die Erfordernisse einer visuellen Sprache anpassen.

Die durch VLEli generierten Entwicklungsumgebungen bestehen aus einem auf die Sprache spezialisierten Strukturediteor sowie Werkzeugen, mit denen visuelle Programme analysiert und weiterverarbeitet werden können. In den Entwicklungsumgebungen werden die visuellen Programme in einer abstrakten Form gespeichert, die die Grundlage zur Durchführung wichtiger sprachspezifischer Aufgaben ist. Diese Aufgaben werden durch *attributierte Grammatiken* spezifiziert. Ein Generator stellt daraus Attributauswerter für die Lösung dieser Aufgaben her.

In dieser Arbeit werden Attributauswerter in einem großen Umfang für die Aufgaben der Sprachimplementierung verwendet. Sie erzeugen etwa die für einen Benutzer bestimmte graphische Darstellung eines visuellen Programms, wozu unter anderem Layoutberechnungen durchgeführt werden müssen. Sie bestimmen weiterhin die Editieroperationen, wobei auch semantische Informationen über das editierte Programm extrahiert und verwendet werden können, um komfortable Editieroperationen oder strukturelle Transformationen vorzusehen. Schließlich werden sie auch eingesetzt, um Analysatoren und Transformatoren zu erzeugen, mit denen visuelle Programme weiterverarbeitet werden können.

Ein wichtiges neues Ergebnis der vorliegenden Arbeit besteht darin, daß die Techniken von Kastens und Waite [1994] auf das Gebiet der Implementierung visu-

eller Sprachen ausgeweitet wurden. Diese Modularisierungstechniken für attributierte Grammatiken werden in dieser Arbeit verwendet, um die Spezifikationsmodule zu implementieren. Dabei werden einige ihrer charakteristischen Merkmale genutzt, um Sprachunabhängigkeit, Kombinierbarkeit und einfache Anwendung der Spezifikationsmodule zu erzielen.

Um die Modularisierungstechniken anwenden zu können, müssen die visuellen Programme durch Strukturbäume repräsentiert werden. In dieser Arbeit werden dazu erstmalig Strukturbäume durch Definitionstabellen ergänzt. So entsteht eine Programmrepräsentation, mit der einerseits ähnlich wie bei Graphen auch allgemeine Beziehungen direkt dargestellt werden können. Andererseits können durch diese Repräsentation auch allgemeine Werkzeuge für die Verarbeitung von Sprachen, z.B. Generatoren für Attributauswerter für die Implementierung visueller Sprachen wiederverwendet werden, ohne das Konzept dieser Werkzeuge zu verändern oder die Beziehungen visueller Sprachen auf hierarchische zu beschränken. Ein weiterer Vorteil dieser Repräsentation resultiert für Sprachen, die eine visuelle und eine äquivalente textuelle Notation aufweisen, etwa SDL. Für solche Sprachen können weiterverarbeitende Werkzeuge für beide Notationen aus einer gemeinsamen Spezifikation generiert werden.

Ein weiteres neues Ergebnis dieser Arbeit ist die Methode der Layoutberechnung. Um das Layout eines visuellen Programms zu berechnen, können in VLEli verschiedene Verfahren benutzt werden: die Anordnung der Sprachkonstrukte kann direkt durch Attributberechnungen ermittelt oder auch durch ein Constraint-Netzwerk beschrieben werden, das durch einen Attributauswerter generiert wird. Die Verfahren wirken sich unterschiedlich auf das Verhalten der damit angeordneten Sprachkonstrukte und so auf die Benutzbarkeit einer Sprachimplementierung aus. Beide Verfahren werden einzeln bereits für die Implementierung visueller Sprachen verwendet. Neuartig bei VLEli ist, daß beide Verfahren unterstützt und für verschiedene Sprachkonstrukte miteinander kombiniert werden können. So kann für die Spezifikationsmodule jeweils dasjenige Verfahren benutzt werden, das zu einfacher benutzbaren Editoren führt, ohne dabei die Kombinierbarkeit der Spezifikationsmodule aufzugeben.

Überblick über die folgenden Kapitel

In Kapitel 2 fasse ich zunächst den Stand der Forschung auf dem Gebiet der visuellen Sprachen und ihrer Implementierung zusammen, soweit es für diese Arbeit relevant ist. Dazu gebe ich einen Überblick über das Arbeitsgebiet, gehe auf Grundlagen der Sprachimplementierung ein und beschreibe wesentliche Konzepte verwandter Ansätze. In Kapitel 3 entwickle ich die Idee, wiederverwendbare Entwurfskonzepte visueller Sprachen durch visuelle Muster zu dokumentieren. Dabei umreiße ich die Methode der Sprachimplementierung durch Spezifikations-

module und beschreibe die Arbeiten, die zur Identifizierung der visuellen Muster geführt haben.

Im daran anschließenden Kapitel diskutiere ich die Implementierung visueller Sprachen. Hier begründe ich den Einsatz des Liga-Systems, entwickle die der Sprachimplementierung zugrundeliegende Programmrepräsentation und beschreibe die Anwendung der Attributauswerter, etwa zur Berechnung des Layouts und zur Bestimmung der Editieroperationen. In Kapitel 5 schildere ich die Implementierung der Spezifikationsmodule und evaluiere den Ansatz der Implementierung visueller Sprachen durch Module. Die Ergebnisse der Arbeit fasse ich schließlich in Kapitel 6 zusammen. Die Bezüge meiner Arbeit zu Arbeiten anderer beschreibe ich jeweils am Ende der zentralen Kapitel.

Hinweise für die Leser

Zitate

Anstelle der weitverbreiteten Form von Literaturreferenzen, bei der sowohl Autor als auch Erscheinungsjahr in eckige Klammern eingefaßt sind (z.B. [Kastens und Waite 1994]), werden in dieser Arbeit alternativ auch nur Erscheinungsjahr und Seitenangaben in eckige Klammern gesetzt und die Namen der Autoren in den Text integriert. Diese Form kann manchmal den Ausdruck und Lesefluß verbessern. Bis zu zwei Autoren werden hierbei direkt angeführt, ab drei Autoren wird nur der erste Autor mit dem Zusatz “u.a.” genannt.

Personenbezeichnungen

In dieser Arbeit wird die maskuline Form auch dann verwendet, wenn das natürliche Geschlecht unwichtig ist oder sowohl männliche als auch weibliche Personen gemeint sind. In diesem Sinne sind “der Anwender” sowie “der Sprachentwickler” ausdrücklich geschlechtsneutral gemeint und sollen auch weibliche Personen ansprechen.

Visuelle Programme

Mit einem “visuellen Programm” ist allgemein ein Satz einer visuellen Sprache gemeint. Dies soll nicht den weiten Bereich der Anwendungen ausschließen, in denen visuelle Sprachen benutzt werden, die nicht der Beschreibung von Software dienen.

2. Implementieren visueller Sprachen

In diesem Kapitel fasse ich den Stand der Forschung im Bereich visueller Sprachen und ihrer generatorgestützten Implementierung zusammen, soweit er für diese Arbeit relevant ist.

Im ersten Abschnitt motiviere ich den Einsatz visueller Sprachen. Ausgehend von einem Überblick über das Arbeitsgebiet liefere ich Beispiele für visuelle Programmier- und Softwarebeschreibungssprachen und referiere Arbeiten, die Hinweise für den Entwurf einer guten Notation für eine (neue) visuelle Sprache geben.

Im zweiten Teil dieses Kapitels erläutere ich Methoden und Werkzeuge, die die Grundlage für die Konzeption und Implementierung des VLEli-Systems bilden. In Abschnitt 2.3 gehe ich schließlich auf Verfahren ein, mit denen Entwicklungsumgebungen für visuelle Sprachen erzeugt werden können. Dabei diskutiere ich auch einige Generatoren und universelle Editoren.

2.1. Visuelle Sprachen und visuelle Programmierung

In der Anfangszeit visueller Sprachen wurden Thesen aufgestellt, warum visuelle Sprachen Vorteile gegenüber textuellen Sprachen haben. So wurde beispielsweise verschiedentlich behauptet, daß visuelle Sprachen die Kommunikation erleichtern, weil ihre Symbole den realen Gegenständen in weitaus größerem Maße ähneln, als das dafür gebräuchliche Wort [Staufer 1987, S. 53] oder daß visuelle Programme leichter verständlich für ungeübte Beobachter sind [Glinert 1990a, S. 173f].

Inzwischen sind empirische Untersuchungen visueller und textueller Notationen durchgeführt worden. Pandey und Burnett [1993] haben etwa die Einfachheit der Programmierung mit einer visuellen und zwei textuellen Sprachen durch die Anzahl der Fehler gemessen, die von Probanden bei der Erstellung einfacher Matrix-Manipulationsprogramme gemacht wurden. Die Autoren haben dabei

festgestellt, daß Programme, die in der visuellen Sprache Forms/3 erstellt wurden, am häufigsten vollständig korrekt waren. Es gibt aber auch ernüchternde Ergebnisse: Green und Petre [1992] bescheinigen der visuellen Sprache LabView größere Probleme in Bezug auf die Verständlichkeit ihrer Programme.

Um die Frage nach den Vorteilen visueller Sprachen und damit nach den Zielen ihrer Benutzung zu beantworten, zitiere ich im folgenden die Ergebnisse von Whitley. Whitley [1997] hat zahlreiche empirische Untersuchungen visueller und textueller Notationen zusammengetragen. Er hat dabei zwischen visuellen Programmiersprachen und anderen visuellen Notationen unterschieden und kommt zu folgenden Schlußfolgerungen:

> *"...compared to textual notations, visual notations can provide better organization and can make information explicit. Moreover, properly used visuals result in quantifiable performance benefits."*

und

> *"...the benefit of using visual representations grows as the size or complexity of the problem grows."*

Sinnvoll eingesetzt, kann man in visuellen Sprachen also grobe Strukturen und Zusammenhänge besser veranschaulichen als mit textuellen Sprachen. Das bewirkt, daß es Menschen meßbar leichter fällt, Verständnisfragen zu beantworten. Dadurch sind Vorteile insbesondere bei komplexen Problemstellungen zu erzielen. Ein Beispiel dafür ist der Entwurf objektorientierter Softwaresysteme.

2.1.1. Überblick

Im folgenden werde ich einen Überblick über das Arbeitsgebiet geben und einige Ansätze zusammenfassen, die visuelle Sprachen klassifizieren.

Eine allgemein anerkannte Gliederung des Arbeitsgebietes gibt Shu [1988] unter dem Oberbegriff "visuelle Programmierung" an. Nach ihrer Definition ist "visuelle Programmierung" gekennzeichnet durch "die Benutzung einer aussagekräftigen graphischen Repräsentation im Verlauf der Programmierung"[1]. Sie gliedert "Visuelle Programmierung" in die Gebiete "visuelle Umgebungen" und "visuelle Sprachen", siehe Abbildung 2.1.

Visuelle Umgebungen verwenden eine integrierte graphische Benutzungsschnittstelle. Aus der Vielzahl der Anwendungen sind für den Bereich "Visuelle Programmierung" insbesondere zwei Anwendungsgebiete interessant, die Visualisierungssysteme und die Systeme zum "visuellen Instruieren" [2]. Die Visualisie-

[1] engl. Original: "the use of meaningful graphic representations in the process of programming" [Shu 1988, S. 9]

[2] engl. Original: "visual coaching"

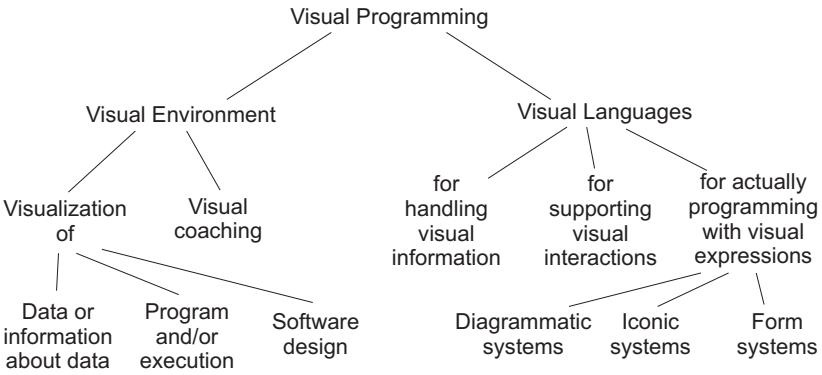


Abbildung 2.1.: Gliederung des Arbeitsgebiets “Visuelle Programmierung” nach Shu [1988]

rung hat die Aufgabe, durch visuelle Darstellungen eine Verbesserung des Verständnisses in ihrem Anwendungsbereich zu erzielen. Im Bereich der Visualisierungssysteme identifiziert Shu Systeme zur Visualisierung von Daten oder Informationen über Daten, Systeme zur Visualisierung der Programmausführung (auch textueller Sprachen) und Systeme zur Visualisierung des Softwaredesigns. Price, Baecker und Small [1993] geben einen Überblick über zwölf Visualisierungssysteme.

Unter “visuellem Instruieren” versteht Shu den Anwendungsbereich der Programmierung durch Beispiel. Ein Endbenutzer erstellt dabei Programme, indem er Aktionen vorführt, die vom Computer später entweder mehrfach ausgeführt werden oder aus denen ein Programm abgeleitet wird. Cypher [1994] und Lieberman [2000] beschreiben Systeme und Methoden für die Programmierung durch Beispiel. Beachtenswert ist hierbei, daß Shu den Bereich der Programmierung durch Beispiel nicht unter “visuelle Sprachen” faßt, was von einigen Autoren anders gesehen wird.

Der Begriff “visuelle Sprache” wird von verschiedenen Autoren unterschiedlich abgegrenzt. Shu [1988] unterscheidet drei Bereiche, die “Sprachen zur Unterstützung visueller Interaktion”, die “Sprachen für die Verarbeitung visueller Informationen” und die “Sprachen für die Programmierung mit visuellen Ausdrücken” (visuelle Programmiersprachen). Chang [1987] fügt eine weitere Kategorie hinzu, die “Piktogrammsprachen zur Verarbeitung visueller Informationen”.

In seinem Kapitel zur Begriffsbestimmung kritisiert Schiffer [1998], daß universelle textuelle Sprachen auch zur Unterstützung der visuellen Interaktion und zur Verarbeitung visueller Informationen herangezogen werden können. Aus der

> *"**Visuell** ist die Bezeichnung für jene Eigenschaft eines Objekts, durch das mindestens eine Information über das Objekt, die für das Erreichen eines Handlungsziels unverzichtbar ist, nur durch das visuelle Wahrnehmungssystem des Menschen gewonnen werden kann."*
>
> *"Eine **visuelle Sprache** ist eine formale Sprache mit visueller Syntax oder visueller Semantik und dynamischer oder statischer Zeichengebung."*
>
> *"Eine **visuelle Programmiersprache** ist eine visuelle Sprache zur vollständigen Beschreibung der Eigenschaften von Software. (...)"*
>
> *"Eine **visuelle Softwarebeschreibungssprache** ist eine visuelle Sprache zur Beschreibung bestimmter Aspekte von Software. Aus den damit erstellten Beschreibungen ist Programmcode maschinell generierbar."*
>
> *"**Visuelle Programmierung** ist die Erstellung von Software mit visuellen Programmiersprachen und visuellen Softwarebeschreibungssprachen. (...)"*

Abbildung 2.2.: Definitionen von Schiffer [1998, Kap. 2] zum Arbeitsgebiet "Visuelle Sprachen"

Kategorie von Shu sieht er deshalb lediglich den Bereich der visuellen Programmierung als kennzeichnend für den Begriff "visuelle Sprachen" an. Unter dem Oberbegriff "visuelle Sprachen" versteht er aber nicht nur Programmiersprachen, sondern identifiziert beispielsweise den Entwurf von Software als weiteres Einsatzgebiet. Für diese Kategorie prägt er den Begriff "visuelle Softwarebeschreibungssprachen". In Abbildung 2.2 habe ich die Definitionen aus seinem Buch wiedergegeben.

Um das Gebiet "visueller Sprachen" weiter zu unterteilen, schlägt Shu die Aufteilung in Diagramm-, Piktogramm- und Formularsprachen vor. Sie betrachtet dabei drei Aspekte visueller Programmiersprachen: Neben dem Sprachniveau und dem Anwendungsbereich betrachtet sie noch den Visualisierungsgrad einer visuellen Sprache. Der Nutzen einer solchen Unterteilung ist jedoch fraglich, weil bis jetzt noch nicht gezeigt werden konnte, daß Sprachen mit hohem Visualisierungsgrad Vorteile gegenüber solchen mit niedrigem haben.

Eine ähnliche Unterteilung wird auch in der Taxonomie von Singh und Chignell [1992] benutzt. Die visuellen Programmiersprachen werden auch hier unter "Visueller Programmierung" einsortiert und weiter in "Flußdiagramme", "Icons", "Tabellen/Formulare" und "Andere" unterteilt. Bei "Flußdiagrammen" wird weiter zwischen "Datenfluß" und "Kontrollfluß" unterschieden.

Eine andersartige Klassifikation liefert Myers [1994, 1990]. In einer ersten Taxonomie werden visuelle Sprachen zusammen mit ihrer Implementierung betrachtet und in acht Kategorien aufgeteilt. Zur Unterscheidung wird zwischen Program-

mierung durch Beispiel oder durch Programme, zwischen visuellen und textuellen Sprachen sowie zwischen Übersetzern und Interpretern unterschieden. Als weiteren Aspekt für die Unterteilung schlägt Myers vor, wesentliche Charakteristika der (graphischen) Repräsentation der Programme zu benutzen.[3]

In ähnlicher Weise werden auch durch Costagliola u.a. [1997a] wesentliche Aspekte der Notation zur Unterteilung visueller Sprachen benutzt. Hier werden die Kriterien aber in eine Hierarchie eingebettet. Auf oberster Ebene wird etwa zwischen der Darstellung von Beziehungen eher durch Verbindungen oder durch geometrische Aspekte unterschieden. Die sich ergebende Hierarchie wird auch zur generatorgestützten Implementierung visueller Sprachen verwendet [Costagliola u.a. 1999].

Nach der Ansicht von Schiffer [1998, S. 116] verdeckt "die Gliederung visueller Programmiersprachen nach dem Erscheinungsbild der Sprachkonstrukte (...) einen weitaus interessanteren Gesichtspunkt, nämlich das konzeptionelle Modell der Programmiersprache". Zur weiteren Unterteilung der visuellen Sprachen benutzt er die Klassifizierung, die Burnett und Baker [1994] zur Erfassung der Informatik-Literatur für visuelle Sprachen eingeführt haben. Er argumentiert, daß ein Klassifikationsschema, das zur Unterteilung von Veröffentlichungen über Sprachen benutzt werden kann, sicherlich auch zur Unterteilung der Sprachen selbst geeignet ist. Das Klassifikationsschema habe ich in Abbildung 2.3 aufgeführt. Ich habe aber nicht die Version aus der Veröffentlichung, sondern die aktualisierte Fassung aus dem WWW [Burnett 2000] verwendet. Das Klassifikationsschema liefert auch eine Grundlage zur Beschreibung einer visuellen Sprache, etwa für Eigenschaften der Sprache (VPL-III) oder für ihren Zweck (VPL-V).

2.1.2. Einige visuelle Sprachen

Um die praktische Relevanz dieses Arbeitsgebietes zu dokumentieren und um eine Grundlage für das Verständnis der folgenden Kapitel zu geben, werde ich im folgenden einige visuelle Sprachen beschreiben. Die Sprachen wurden nach ihrem Erfolg und Bekanntheitsgrad ausgewählt, z.B. UML, LabView und Prograph. Einige weniger bekannte Sprachen sind aufgeführt, weil sie in Untersuchungen oder Beispielen der folgenden Kapitel eine Rolle spielen.

Zur Strukturierung wurde keine der eingeführten Klassifikationen benutzt, weil Vollständigkeit in Bezug auf Klassifikationsschemata in diesem Abschnitt von eher untergeordneter Bedeutung ist. Im ersten Unterabschnitt werde ich einen kurzen Überblick über die Notationen der Unified Modeling Language geben und drei für diese Arbeit wichtige Notationen vorstellen. Die nächsten beiden Abschnitte stellen zwei kommerziell erfolgreiche Datenflußsprachen, LabVIEW und

[3] Der Autor bezeichnet das Unterscheidungskriterium als "Spezifikationstechnik", beschreibt es aber mit "...what kind of representation they use for the code"

VPL: Visual Programming Languages

VPL-I. Environments and Tools for VPLs

VPL-II. Language Classifications

A. Paradigms
   1. Concurrent languages
   2. Constraint-based languages
   3. Data-flow languages
   4. Form-based and spreadsheet-based languages
   5. Functional languages
   6. Imperative languages
   7. Logic languages
   8. Multi-paradigm languages
   9. Object Oriented languages
   10. Programming-by-demonstration languages
   11. Rule-based languages

B. Visual Representations
   1. Diagrammatic languages
   2. Iconic languages
   3. Languages based on static pictorial sequences

VPL-III. Language Features

A. Abstraction
   1. Data abstraction
   2. Procedural abstraction

B. Control flow

C. Data types and structures

D. Documentation

E. Event handling

F. Exception handling

VPL-IV. Language Implementation Issues

A. Computational approaches (e.g. demand-driven, data-driven)

B. Efficiency

C. Parsing

D. Translators (interpreters and compilers)

VPL-V. Language Purpose

A. General-purpose languages

B. Database languages

C. Image-processing languages

D. Scientific visualization languages

E. User-interface generation languages

F. Languages for programming web-based applications

VPL-VI. Theory of VPLs

A. Formal definition of VPLs

B. Icon theory

C. VPL design issues
   1. (Cognitive and user-interface design issues moved to section VPL-VI.D)
   2. Effective use of screen real estate
   3. Liveness
   4. Scope
   5. Type checking and type theory
   6. Visual representation issues (e.g. static representation, animation)

D. Human-oriented issues
   1. Usability studies
   2. Cognitive and user-interface design issues

VPL-VII. Software Engineering Issues for VPLs

A. Reusing visual code

B. Testing visual code

C. Debugging visual code

Abbildung 2.3.: Klassifikationsschema für visuelle Programmiersprachen von Burnett und Baker [1994], hier nach Burnett [2000]

Prograph vor. Im Anschluß daran beschreibe ich die Sprache Streets, die im Rahmen einer Projektgruppe an der Universität Paderborn entstanden ist. Schließlich gehe ich auf die Nassi-Shneiderman-Diagramme ein.

Weitere Kurzbeschreibungen wichtiger visueller Sprachen können den Arbeiten von Schiffer [1998] und Poswig [1996] entnommen werden. Eine Reihe objektorientierter visueller Sprachen beschreiben Burnett u.a. [1995]. Schließlich liefern Glinert [1990b] und Shu [1988] einen Überblick über ältere Arbeiten.

2.1.2.1. Unified Modeling Language

Die 'Unified Modeling Language' ist eine visuelle Softwarebeschreibungssprache zum Entwurf objektorientierter Software. Sie wurde verschiedentlich implementiert, z.B. durch Rational Rose [Quatrani 1998] und durch Argo/UML [Robbins u.a. 1998]. UML ist aus Arbeiten von Booch [1991], Rumbaugh u.a. [1991] und Jacobson u.a. [1992] hervorgegangen. Unter der Federführung der Object Management Group begann 1996 die Standardisierung der Sprache, an der zahlreiche Unternehmen beteiligt waren. Die derzeit aktuelle Version 1.3 ist im Juni 1999

Main Area	*Diagrams*	*Main Concepts*
structural	class diagram	class, association, generalization, dependency, realization, interface
	use case diagram	use case, actor, association, extend, include, use case generalization
	component diagram	component, interface, dependency, realization
	deployment diagram	node, component, dependency, location
dynamic	statechart diagram	state, event, transition, action
	activity diagram	state, activity, completion transition, fork, join
	sequence diagram	interaction, objekt, message, activation
	collaboration diagram	collaboration, interaction, collaboration role, message
model management	class diagram	package, subsystem, model

Tabelle 2.1.: Diagrammarten der Unified Modeling Language nach Rumbaugh u.a. [1999]

publiziert worden [Rational Software Corporation u.a. 1999]. UML ist lediglich die bekannteste visuelle Sprache zum objektorientierten Entwurf. Sie ist aus der Integration zahlreicher anderer Sprachen und Notationen hervorgegangen. Einen Überblick über zahlreiche graphische Notationen zum Entwurf objektorientierter Software liefert Wieringa [1998].

Der hohe Bekanntheitsgrad von UML ist der breiten Akzeptanz zuzurechnen, für die sicherlich auch die Mitwirkung zahlreicher Unternehmen die Ursache ist. Die Unified Modeling Language besteht aus acht verschiedenen Diagrammarten. Eine Übersicht über die Notationen und die zugrundeliegenden Konzepte liefert Tabelle 2.1. Die Notationen sind dabei in zwei Gruppen unterteilt. Die erste Gruppe umfaßt Notationen, die statische Eigenschaften eines objektorientierten Softwaresystems beschreiben. Die zweite Gruppe solche, die dessen dynamisches Verhalten spezifizieren. Die Spalte “main concepts” gibt die wesentlichen Entwurfskonzepte an, die durch die Notation visualisiert werden.

Für diese Arbeit sind die Klassendiagramme, die Ablaufdiagramme und die Zustandsdiagramme von größerer Bedeutung. Mit Hilfe der Klassendiagramme wird ein Teil der Klassenhierarchie eines objektorientierten Softwaresystems entworfen. Die Klassen werden dabei durch Rechtecke dargestellt, die maximal dreifach unterteilt sein können, um den Namen, die Attribute und die Methoden der Klasse aufzunehmen. In den Diagrammen werden Vererbungsbeziehungen, Assoziationen und Aggregationen unterschieden und durch Linien mit verschiedenartigen Pfeilspitzen dargestellt. Interessante Merkmale der Klassendiagramme sind die verschiedenartigen Beschriftungen der Linien und die Tatsache, daß auch Linien wiederum verbunden werden können. Ein Beispiel für ein Klassendiagramm ist in Abbildung 2.4 abgebildet. Im Beispiel wird die Klassenhierarchie für ein Flugreservierungssystem spezifiziert.

Mit Hilfe der Ablaufdiagramme wird das zeitliche Verhalten der Objekte eines Software-Systems beschrieben. Die Lebenszeit eines Objekts wird durch vertikal verlaufende Lebenslinien visualisiert, zwischen denen Kommunikation durch horizontal verlaufende Pfeile spezifiziert werden. Die Art der Visualisierung ist damit dieselbe, die bei den Message-Sequence-Charts [Mauw 1996] benutzt wird, um die Kommunikation zwischen kooperierenden Prozessen zu spezifizieren.

Mit Ablaufdiagrammen kann man beispielsweise den Ablauf einer Transaktion zwischen den Objekten des Softwaresystems spezifizieren. Das Beispiel aus Abbildung 2.5 visualisiert den Ablauf einer Sitzreservierung im o.a. Flugreservierungssystem. Das mit Kiosk beschriftete Objekt realisiert die Benutzungsschnittstelle, die anderen Objekte steuern den Vorgang und führen den Bezahlvorgang per Kreditkarte durch.

Mit Hilfe der Zustandsdiagramme wird der Zustand eines Objekts des Softwaresystems oder der Ablauf einer Methode spezifiziert. Das Zustandsdiagramm enthält benannte Zustände, die durch abgerundete Rechtecke dargestellt werden. Die Pfeile zwischen den Zuständen sind Übergänge. Sie werden als Reaktion des

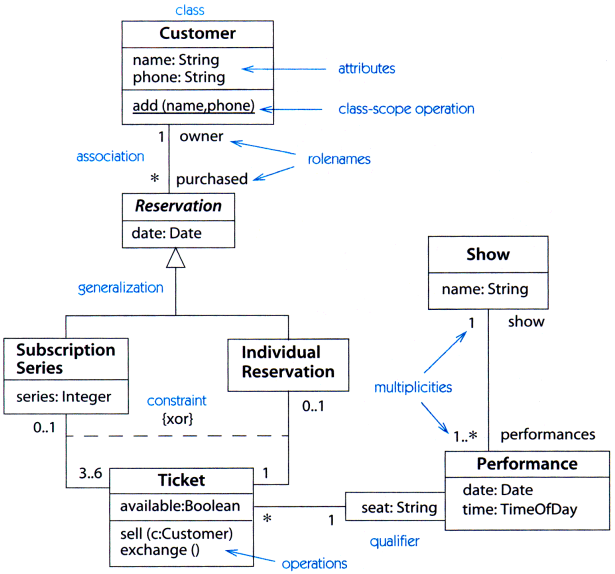


Abbildung 2.4.: UML-Klassendiagramm nach Rumbaugh u.a. [1999]

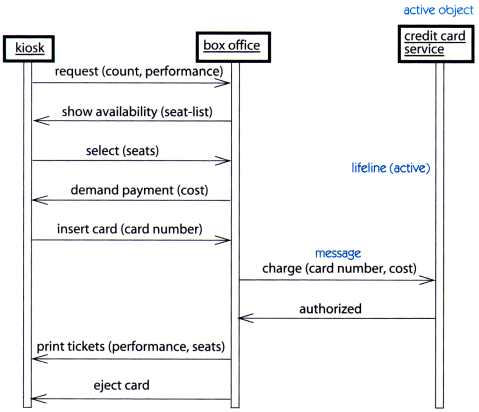
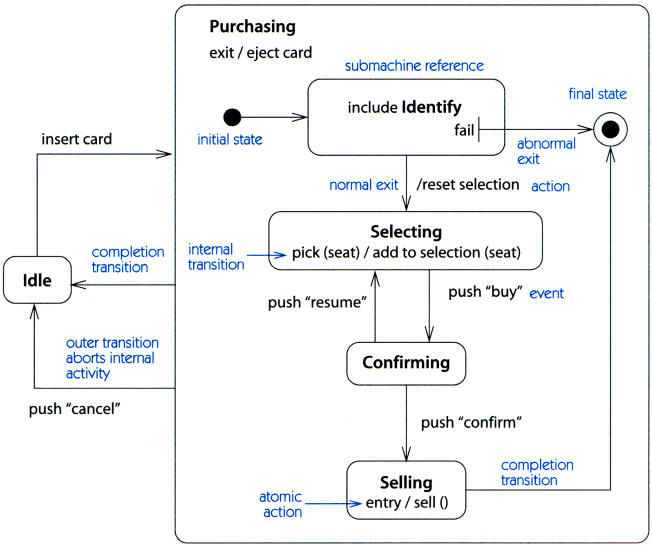
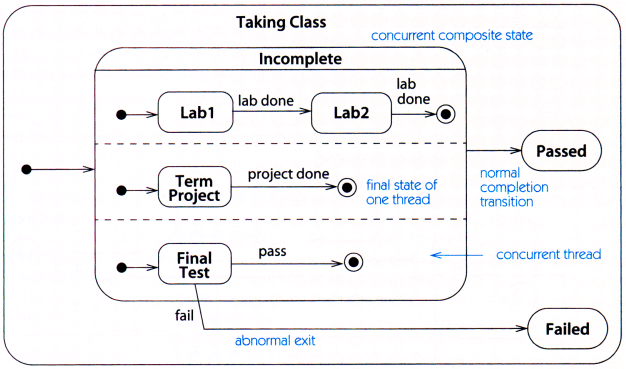


Abbildung 2.5.: UML-Ablaufdiagramm nach Rumbaugh u.a. [1999]



(a)



(b)

Abbildung 2.6.: UML-Zustandsdiagramme nach Rumbaugh u.a. [1999]

Objekts auf äußere Ereignisse angestoßen und können auch das Ausführen von Code beinhalten.

Das Zustandsdiagramm aus Abbildung 2.6(a) enthält wiederum ein Beispiel aus dem Flugreservierungssystem. Der rechte Zustand "Purchasing" ist dabei in vier weitere Zustände unterteilt. Diese Art der Gruppierung wird XOR-Superstate genannt, weil genau einer der inneren Zustände "aktiv" ist. Eine andere Art der Gruppierung heißt AND-Superstate, siehe dazu Abbildung 2.6(b). Ist der Zustand "Incomplete" aktiv, so ist in jedem der drei Unterbereiche ein Zustand aktiv. Diese Art der Hierarchisierung ist ein wesentlicher Vorteil der Zustandsdiagramme im Vergleich zu endlichen Automaten.

Für die weitere Betrachtung der Zustandsdiagramme wird in dieser Arbeit die Beschreibung von Harel [1988] zugrundegelegt. Die Unterschiede zum UML-Standard bestehen neben kleineren Änderungen der graphischen Darstellung darin, daß keine Konstrukte zur Ausführung von Code vorgesehen sind.

2.1.2.2. LabView

Die visuelle Programmiersprache LabView[4] ist eine auch kommerziell erfolgreiche Datenflußsprache. Mit Hilfe des LabView-Systems [Jamal und Wenzel 1995, 1997] können Meß- und Steuerungssysteme softwaretechnisch realisiert werden. Das System eignet sich u.a. für die Überwachung, Auswertung und Visualisierung von Meßdaten sowie die Simulation elektronischer Steuerungen. Die erste Version des Systems ist 1986 für den Macintosh-Computer umgesetzt worden. Die derzeit aktuelle Version 6i wird von National Instruments [2000] vertrieben und läuft auf fast allen gängigen Betriebssystemen.

Neben dem Prinzip der datenflußgesteuerten Programmierung ist in LabView-Programmen die strikte Trennung zwischen Schnittstelle und Implementierung und die Modularisierung von Bedeutung. Die Programmbausteine in LabView werden "virtuelle Instrumente" genannt. Sie ähneln in Aussehen und Funktionsweise einem realen elektrischen Instrument. Sie bestehen aus einer Frontplatte, die die Benutzungsschnittstelle eines virtuellen Instruments darstellt und einem Blockschaltbild, das in der Art eines Schaltplans die Programmlogik des virtuellen Elements enthält. Die Frontplatte kann Ein- und Ausgabekomponenten wie Dreh- und Schieberegler oder analoge Anzeigen enthalten, die nach ihrem Einfügen in die Frontplatte im Blockschaltbild automatisch als Datenquellen oder -senken auftauchen.

Abbildung 2.7 zeigt ein virtuelles Instrument, das aus Frontplatte (1) und Blockschaltbild (2) besteht, ein Fenster mit Werkzeugen zur Bearbeitung virtueller Instrumente (3), sowie einige wichtige Konstrukte für die Programmlogik (4). Mit

[4] "LabView" ist eigentlich der Name für die Sprachimplementierung. Die visuelle Sprache des LabView-Systems heißt "G". Trotzdem werde ich im folgenden von der visuellen Sprache "LabView" schreiben, da es sich bei LabView um die einzige Implementierung von "G" handelt.

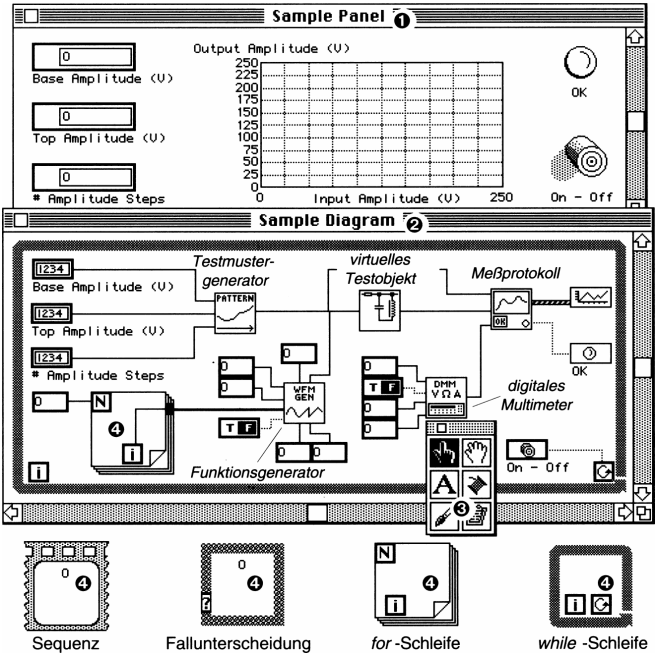


Abbildung 2.7.: Frontplatte, Blockschaltbild und Sprachkonstrukte von LabVIEW nach Poswig [1996, Abb. 3.18]

Hilfe des gezeigten LabView-Programms kann das Ein-/Ausgabeverhalten eines virtuellen Testobjekts gemessen und in einem Diagramm angezeigt werden. Poswig [1996] wendet es an, um das Verhalten einer Frequenzweiche zu visualisieren, die ebenfalls als virtuelles Instrument realisiert ist.

Wie in Abbildung 2.7 in der Mitte anhand der Beschriftungen zu sehen ist, können bei der Implementierung eines virtuellen Instruments andere Instrumente genutzt werden. Deren Datenquellen und -senken erscheinen hier als Schnittstellen, die mit anderen Funktionselementen des Blockschaltbildes verbunden werden können. Durch diese Technik der Modularisierung bleiben virtuelle Instrumente vom Umfang her überschaubar. Dadurch sind LabView-Programme im allgemeinen relativ übersichtlich, die zur Verfügung stehende Bildschirmfläche wird gut genutzt. Wie Green und Petre [1992] in einer Untersuchung zeigen, wird das Verständnis der LabView-Programme aber durch die für Fallunterscheidungen verwendete Notation erschwert.

2.1.2.3. Prograph

Auch Prograph ist eine kommerziell erfolgreiche Datenflußsprache. Im Gegensatz zu LabView ist Prograph als objektorientierte Universalprogrammiersprache konzipiert. Eine erste Idee zu Prograph wurde 1982 an der Acadia Universität im Rahmen eines Seminars hervorgebracht. Die weitere Entwicklung der Sprache ist in den Arbeiten von Pietrzykowski u.a. [1983], Matwin und Pietrzykowski [1985], Cox und Pietrzykowsky [1988], Cox u.a. [1989] und Steinman und Carver [1996] dokumentiert. Inzwischen wird Prograph von Pictorius, Inc. vertrieben. Die neueste, mir bekannte Publikation zu Prograph diskutiert Spracherweiterungen für die Implementierung paralleler Programme [Cox u.a. 1998].

Auf oberster Ebene bestehen Prograph-Programme aus einer Klassenhierarchie. Jede Klasse besteht neben den Attributen auch aus Methoden, die durch eine Folge von Fällen (engl. *cases*) implementiert werden. Jeder Fall ist wiederum durch einen Datenflußgraphen realisiert, der Datenflußoperationen und Prüfungen enthält. Schlägt bei der Programmausführung eine Prüfung in einem Datenflußgraph fehl, so wird die Ausführung des aktuell bearbeiteten Falles abgebrochen und beim Datenflußgraph für den nächsten Fall aufgesetzt.

Abbildung 2.8 zeigt drei Methoden aus der Implementierung des Quicksort-Algorithmus in Prograph. Die links angeordnete Methode “Quick Sort” führt die Interaktion mit dem Benutzer durch und ruft die Methode “quickSort” auf. Diese

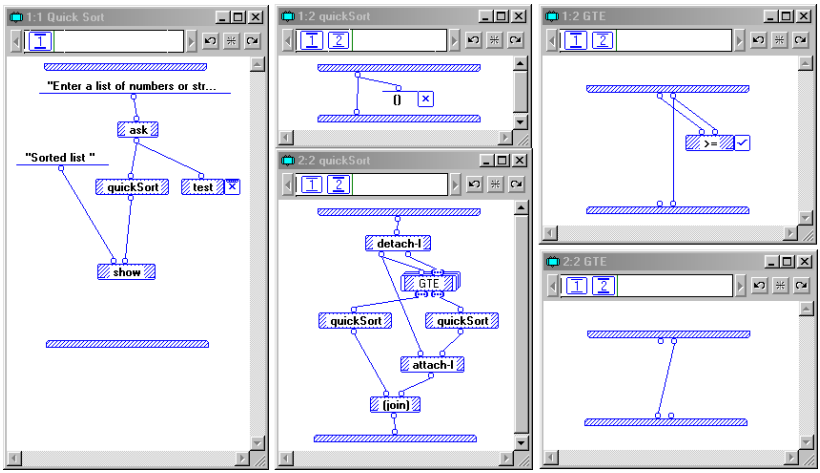


Abbildung 2.8.: Drei Methoden aus der Quicksort-Implementierung in Prograph

ist durch zwei Fälle implementiert. Der erste Fall prüft, ob die Liste leer ist und schlägt fehl, wenn dies nicht der Fall ist. Das führt dann zur Ausführung des unteren Datenflußgraphen für "quickSort". Die Methode "GTE" wird einmal für jedes Element der zu sortierenden Liste aufgerufen. Dadurch wird die Liste in zwei Teillisten aufgeteilt. Mit weiteren Konstrukten können auch iterative Berechnungen formuliert werden.

Interessant an Prograph ist insbesondere der Gebrauch der Fenster. Obwohl mit Prograph umfangreiche Programme konstruiert werden können, bleibt der Umfang der einzelnen Fenster doch meist kompakt und übersichtlich.

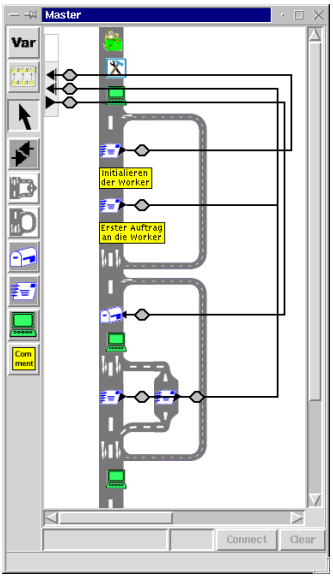
2.1.2.4. Streets

Streets ist eine visuelle Programmiersprache zur Implementierung paralleler Programme. Sie ist 1998 im Rahmen einer Projektgruppe an der Universität Paderborn entstanden. Die Entwicklungsumgebung für Streets-Programme umfaßt auch Codegenerierung für C++ mit Aufrufen von PVM [Geist und Sunderam 1992]. Andere Sprachen, mit denen parallele Programme implementiert oder modelliert werden können, sind beispielsweise Tracs [Bartoli u.a. 1995], Trapper [Kraemer-Fuhrmann u.a. 1993], Klieg [Toyoda u.a. 1997; Shizuki u.a. 1999], Visper [Stankovic und Zhang 1997], Meander [Wirtz 1994, 1995], VPE [Tan und Cai 1995], HeNCE [Beguelin und Dongarra 1991] und Code [Newton und Browne 1992].

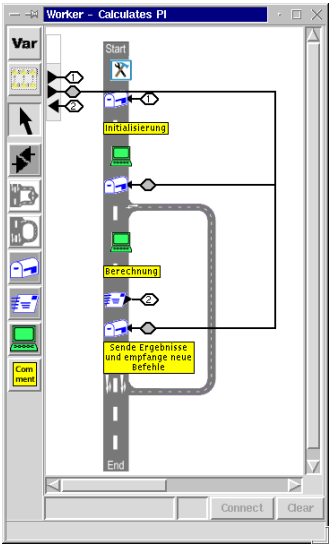
In Streets wird zwischen sequentiellen und parallelen Anteilen eines parallelen Programms unterschieden. Das sequentielle Verhalten wird textuell, die parallelen Teile visuell programmiert. Der Kontrollfluß eines parallelen Prozesses wird, soweit er für das parallele Verhalten des Prozesses von Bedeutung ist, durch eine visuelle Notation spezifiziert, die an Flußdiagramme erinnert, aber die Metapher "Straße" benutzt. Das hat auch später zum Namen der visuellen Sprache geführt.

Auf der Straße befinden sich unterschiedliche Icons, die Softwarebausteine visualisieren. Das Icon "Computer" erlaubt die Eingabe beliebiger sequentieller Codeteile, die Icons "Brief" und "Briefkasten" stehen für Nachrichtenweitergabe an andere Prozesse, die durch das Message-Passing-Prinzip realisiert ist. Der Kommunikationsparter kann dabei auf graphischem Wege durch Linien angegeben werden. Die Erzeugung von Prozessen wird durch das "Stadt"-Symbol visualisiert.

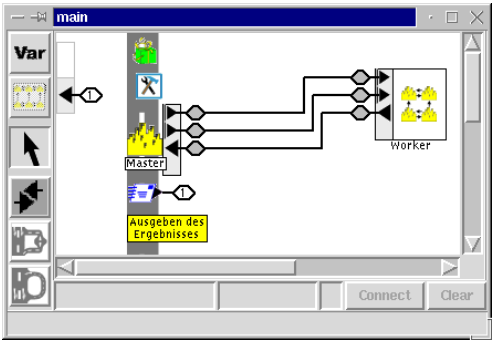
In Abbildung 2.9 ist ein einfaches Streets-Programm abgebildet, das die Zahl π mit Hilfe eines Näherungsverfahrens berechnet: Für zufällig erzeugte Punkte aus dem Inneren eines Quadrats wird ermittelt, ob diese auch innerhalb des Kreises liegen, der dem Quadrat einbeschrieben ist. Das Verhältnis der Anzahl der Punkte im Kreis zur gesamten Anzahl der Punkte ist ein Maß für $\frac{\pi}{4}$. Die Berechnung wird mit Hilfe eines Master-Worker-Programms ausgeführt, dessen visuelle Teile Abbildung 2.9 zeigt.



(a) Programm des Master-Prozesses



(b) Programm der Worker-Prozesse



(c) Hauptprogramm

Abbildung 2.9.: Streets-Programm zur Berechnung von π

2.1.2.5. Fluß- und Nassi-Shneiderman-Diagramme

Nassi und Shneiderman [1973] haben eine visuelle Programmiersprache eingeführt, mit der strukturierte, imperative Programme erstellt werden können. Die Nassi-Shneiderman-Diagramme (NSD) und die Flußdiagramme [Bohm und Jacopini 1966] sind die bekanntesten visuellen kontrollflußbasierten Sprachen. Einen Überblick über eine Reihe ähnlicher Notationen gibt Tripp [1988].

Die NSD sind mehrfach, z.B. durch Systeme wie PIGS [Pong und Ng 1983] oder GRASE [Albizuri-Romero 1984] implementiert worden. Die Sprache enthält Konstrukte für Schleifen, Fallunterscheidungen und parallele Abarbeitung, die i.W. aus einfachen Linien aufgebaut sind. Anweisungen und Ausdrücke werden textuell angegeben und in die Zeichnungen eingefügt. Beispiele für Nassi-Shneiderman-Diagramme treten in den folgenden Kapiteln mehrfach auf, z.B. in Abbildung 5.17(b) auf Seite 196.

Nach Nassi und Shneiderman [1973] sind die Vorteile der NSD, daß die bedingten Anweisungen übersichtlich und selbst bei mehrfach geschachtelten Bedingungen noch leicht verständlich sind. Im weiteren lassen sich NSD relativ kompakt zeichnen.

Visuelle Notationen, die den Kontrollfluß eines Programms visualisieren, werden in verschiedenen neueren Sprachen verwendet, z.B. in Streets [Kastens und Jung 1998] und SDL [Belina u.a. 1991]. Untersuchungen bei Programmieranfängern haben gezeigt, daß Verständnisfragen für visuelle kontrollflußorientierte Programme schneller beantwortet wurden, als für visuelle datenflußorientierte [Good 1999].

2.1.3. Entwurf visueller Sprachen

Ein wesentlicher Unterschied zwischen dem Entwurf visueller und textueller Sprachen ist die Konzeption der Notation für die Sprache. Für die visuelle Sprache umfaßt dies die Auswahl einer geeigneten visuellen Notation und ggf. noch die Planung der Benutzerinteraktion mit dieser Notation.

Obwohl in der Vergangenheit viele Thesen über potentielle Vorteile visueller Notationen publiziert wurden – ich habe am Anfang dieses Kapitels Beispiele genannt – gibt es bislang nur wenige Arbeiten, die, durch empirische Untersuchungen untermauert, Hinweise auf Auswahlkriterien für die Notation einer neuen visuellen Sprache geben. In den folgenden Unterabschnitten werde ich einige wichtige Arbeiten aus diesem Bereich diskutieren.

Eine visuelle Sprache und ihre Implementierung kann im weiteren auch als Mensch-Maschine-Schnittstelle für ein Informationssystem angesehen werden. Entwurfsrichtlinien existieren auch für diese Bereiche [Mayhew 1992; Lin u.a. 1997]. Sie können auf den Entwurf visueller Sprachen übertragen werden [Bottoni u.a. 1999].

2.1.3.1. "Cognitive Dimensions"

Die "Cognitive Dimensions" wurden von Green [1989] als Alternative zu GOMS (Effizienzuntersuchung einzelner einfacher Aufgaben) [Card u.a. 1983] eingeführt, um es Sprachentwicklern und Nichtpsychologen zu ermöglichen, sowohl visuelle, als auch textuelle Sprachen und ihre Implementierung zu untersuchen. "Cognitive Dimensions" sind Eigenschaften der Notation – oder genauer der strukturellen Aspekte der Notation einer Sprache. Sie sind als orthogonal zueinander stehende Einflußfaktoren für die kognitiven Prozesse herausgearbeitet worden, die einen Anwender einer Sprache in seiner Fähigkeit beeinflussen, ein Programm zu entwickeln, zu lesen oder zu ändern.

Die "Cognitive Dimensions" gründen auf aktuellen Ansichten über die Vorgänge beim Programmieren, die die Mensch-Maschine-Schnittstelle und die relevanten psychologischen Aspekte betreffen [Green und Petre 1996]. Sie werden in zahlreichen Arbeiten für die Untersuchung visueller Sprachen und ihrer Implementierung eingesetzt. Die Arbeiten von Green und Petre [1996] untersuchen z.B. die Sprachen Prograph, LabView und Basic, Modugno, Green und Byers [1994] untersuchen die visuelle Sprache Pursuit, Hendry und Green [1994] Tabellenkalkulationen.

In Tabelle 2.2 habe ich eine Kurzzusammenfassung der "Cognitive Dimensions" zitiert, die Green und Petre [1996] in ihrer Arbeit benutzen. Einige der Faktoren sind sicherlich in dieser Kurzform noch nicht verständlich. Eine ausführliche Beschreibung mit prägnanten Beispielen ist in [Green und Petre 1996] enthalten und soll hier nicht wiederholt werden. An dieser Stelle will ich lediglich Beispiele für die Faktoren aufführen, die als Qualitätskriterien für die *Implementierung* einer Sprache angewendet werden können, das sind die sekundäre Notation und die Viskosität.

Abbildung 2.10 zeigt die Beispiele, die Green und Petre dafür benutzen. In Abb. 2.10(a) ist ein Teil eines Basic-Programms abgebildet, das Möglichkeiten für sekundäre Notationen illustriert. Die Gemeinsamkeiten und Unterschiede der beiden Anweisungsblöcke werden hier durch die Reihenfolge der Anweisungen und durch den Leerraum hervorgehoben. Unter Möglichkeiten der sekundären Notation fassen Green und Petre auch Kommentare, betrachten diese jedoch als nur wenig nützlich, wenn nicht größere Programmbereiche mit ihnen erläutert werden können. In diesem Zusammenhang sehen die Autoren Probleme, wenn visuelle Programme wie Prograph mit Hilfe sehr vieler Fenster editiert werden. Abb. 4.8(a) auf Seite 114 zeigt beispielsweise einen Teil eines typischen Prograph-Programms.

Die kognitive Dimension Viskosität ist ein Begriff, der aus der Physik übernommen wurde. Übertragen auf Programmiersprachen und ihre Implementierung

Cognitive Dimension	*Description*
Abstraction gradient	What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
Closeness of mapping	What 'programming games' need to be learned?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Diffuseness	How many symbols or graphic entities are required to express a meaning ?
Error-proneness	Does the design of the notation induce 'careless mistakes'?
Hard mental operations	Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening?
Hidden dependencies	Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
Premature commitment	Do programmers have to make decisions before they have the information they need?
Progressive evaluation	Can a partially-complete program be executed to obtain feedback on 'How am I doing'?
Role expressiveness	Can the reader see how each component of a program relates to the whole?
Secondary notation	Can programmers use layout, colour, other cues to convey extra meaning, above and beyond the 'official' semantics of the language?
Viscosity	How much effort is required to perform a single change?
Visibility	Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

Tabelle 2.2.: Kurzbeschreibungen der "Cognitive Dimensions" nach Green und Petre [1996]

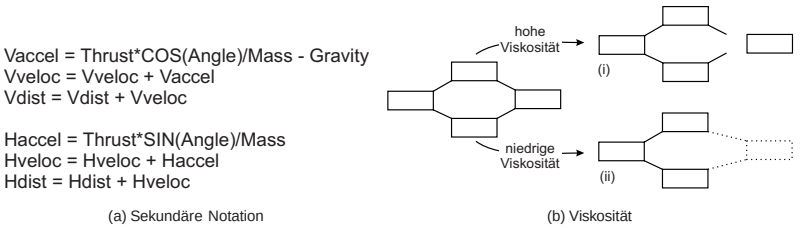


Abbildung 2.10.: Beispiele für einige "Cognitive Dimensions" (vgl. [Green und Petre 1996, Fig. 13 u. 14])

mißt sie den "Aufwand, um eine kleine Änderung zu realisieren".[5] Abb. 2.10(b) visualisiert zwei Alternativen für eine Editieroperation in einem visuellen Editor. Reagiert der Editor auf Verschiebung eines Rechtecks wie in Teilbild (i) illustriert, so muß ein Benutzer alle Verbindungen von Hand nacheditieren, was als extrem hohe Viskosität angesehen wird. Intelligentere Editoren "behandeln Verbindungen wie Gummi-Bänder" (übersetzt), wie in (ii) dargestellt.

Die Bedeutung der "Cognitive Dimensions" für den Sprachentwurf im allgemeinen und den visueller Sprachen im besonderen kann durch zahlreiche Veröffentlichungen belegt werden. Whitley [1997] betont beispielsweise, daß diese "scharfsinnigen Beobachtungen die Basis für eine Theorie (visueller Programmiersprachen) bilden" und daß sie bereits "in einem frühen Stadium des Sprachentwurfs für die Evaluierung nützlich sind" (übersetzt).

Einen direkteren Einfluß auf den Entwurf einer visuellen Sprache haben die Gütekriterien, die Yang, Burnett, DeKoven und Zloof [1997] aus den "Cognitive Dimensions" entwickelt haben. Mit den Gütekriterien kann der Entwurf einer visuellen Notation anhand der Qualität der sog. "statischen Repräsentation" und anhand der geplanten Interaktionstechniken für Programme der Sprache evaluiert werden. Unter "statischer Repräsentation" verstehen die Autoren dabei "das Erscheinungsbild eines visuellen Programms 'im Ruhezustand' " (übersetzt). Dabei zählen nur die Teile der Notation eines Programms zur statischen Repräsentation, die sich gleichzeitig auf einem unendlich großen Bildschirm darstellen ließen.

Die Gütekriterien, die Yang u.a. definieren, verfolgen drei Ziele. Der erste Satz von Gütekriterien bewertet die Verständlichkeit, die durch die statische Repräsentation einer visuellen Sprache erzielt wird. Die statische Repräsentation kann das Verständnis fördern, wenn alle Abhängigkeiten zwischen den Programmkon-

[5] engl. Original: "how much work the user has to put in to effect a small change" [Green und Petre 1996, S. 161]

strukten, der Programmstruktur und -logik explizit visualisiert werden. Im weiteren ist es für das Verständnis nützlich, wenn das Ergebnis der Programmausführung mit dem Quelltext zusammen betrachtet werden kann und wenn Möglichkeiten für "sekundäre Notationen" vorhanden sind.

Das zweite Ziel, das mit der statischen Repräsentation verfolgt wird, ist die Skalierbarkeit der Sprache. Die Skalierbarkeit einer visuellen Sprache und ihrer Implementierung bestimmt, in welchem Maße auch umfangreichere Programme damit erstellt werden können. Wichtig sind hier mehrere Aspekte. Zum ersten sollen möglichst viele Details der Darstellung optional auch unterdrückt werden können. Das hilft dem Anwender, einen größeren Überblick über ein visuelles Programm zu erlangen. Im weiteren müssen möglichst alle in Beziehung stehenden Informationen gleichzeitig sichtbar gemacht werden können, was z.B. durch den Einsatz mehrerer Fenster erreicht werden kann. Schließlich ist es notwendig, die vorhandene Bildschirmfläche möglichst gut auszunutzen. Burnett u.a. [1995] identifizieren als weiteren Aspekt die Dokumentierbarkeit der Programme. Die Autoren schlagen Maßnahmen vor, um die zur Verfügung stehende Bildschirmfläche durch Kommentare nicht einzuschränken.

Als drittes muß die statische Repräsentation nach Yang u.a. [1997] möglichst große Ähnlichkeit mit der Notation aufweisen, die vom "Zielpublikum" der Sprache benutzt wird, um die Probleme des Anwendungsbereichs der Sprache zu lösen. Diese Erkenntnis ist eine Schlußfolgerung aus der kognitiven Dimension "Closeness of Mapping". Bei dieser fordern Green und Petre [1996], daß ein Benutzer einer Sprache seine Probleme möglichst leicht auf die Ausdrucksmittel der Sprache abbilden können sollte. Yang u.a. meinen, daß dazu eingeführte visuelle Notationen des Anwendungsbereichs genutzt werden können. Diese sind den Anwendern der Sprache bereits vertraut. Die von Nardi [1993, S. 37-38] angeführten Untersuchungen belegen, daß sich das günstig auf die Effektivität und die Motivation zur Benutzung der visuellen Sprache auswirkt. Auch die von Green und Petre [1993] durchgeführten Untersuchungen weisen nach, daß die Fähigkeit, eine visuelle Notation in der richtigen Weise zu lesen, von der Erfahrung des Betrachters abhängt.

2.1.3.2. Match-Mismatch-Hypothese

Eine weitere wichtige Erkenntnis, die beim Entwurf der Notation einer visuellen oder textuellen Sprache berücksichtigt werden sollte, ist die sog. "Match-Mismatch-Hypothese", die Gilmore und Green [1984] aufgestellt haben. Sie besagt, daß "die Fähigkeit zur Lösung von Problemen davon abhängt, ob die Struktur des Problems zur Struktur der Notation paßt" (übersetzt). Um die Match-Mismatch-Hypothese zu bestätigen, haben Gilmore und Green zunächst eine Untersuchung alternativer textueller Notationen für bedingte Anweisungen durchge-

Inderal — 1 tablet 3 times a day
Lanoxin — 1 tablet every a.m.
Carafate — 1 tablet before meals and at bedtime
Zantac — 1 tablet every 12 hours (twice a day)
Quinaglute — 1 tablet 4 times a day
Coumadin — 1 tablet a day

(a) Listendarstellung

	Breakfast	Lunch	Dinner	Bedtime
Lanoxin	✓			
Inderal	✓	✓	✓	
Quinaglute	✓	✓	✓	✓
Carafate	✓	✓	✓	✓
Zantac		✓		✓
Coumadin				✓

(b) Matrixdarstellung

Abbildung 2.11.: Verschiedene Darstellungen von Zeitplänen zur Einnahme von Medikamenten nach Day [1988] (zitiert nach Whitley [1997])

führt. Green und Petre [1992] haben die Hypothese auf visuelle Notationen übertragen, indem Sie die Sprache LabVIEW untersuchten.

Diese und weitere Untersuchen hat Whitley [1997] aufgearbeitet, um zu gesicherten Aussagen über die Vor- und Nachteile visueller Notationen gegenüber textueller Notationen zu gelangen. Whitley hat dazu zwischen "Beweisen" für und gegen visuelle Sprachen, sowie zwischen Untersuchungen von Programmiersprachen und anderen Untersuchungen unterschieden. Das Ergebnis, zu dem Whitley aufgrund dieser Analyse kommt, habe ich bereits am Anfang dieses Kapitels zusammengefaßt. In diesem Abschnitt will ich zwei interessante Untersuchungen aus dem Nicht-Programmiersprachen-Bereich beschreiben. Diese Untersuchungen sind lediglich zwei Beispiele für eine Reihe von Untersuchungen, die Entscheidungshilfen dafür liefern, unter welchen Umständen bestimmte Arten von Notationen gewählt werden sollten.

Day [1988] hat zwei gleichartige Untersuchungen in unterschiedlichen Anwendungsbereichen durchgeführt. In der ersten hat sie verschiedenartige Notationen eines Zeitplans für die Einnahme von Medikamenten miteinander verglichen. Bei der ersten Notation handelte es sich um eine nach Medikament sortierte Liste, bei der in einer textuellen Form die Einnahmezeitpunkte beschrieben sind, siehe Abbildung 2.11(a). Die zweite Notation enthielt dieselben Informationen, die hier jedoch graphisch in einer Matrix aufbereitet sind, siehe Abbildung 2.11(b). Die Untersuchung wurde mit Hilfe von Verständnisfragen durchgeführt, deren korrekte Beantwortung verglichen wurde. Die Testgruppen mußten jeweils eine der No-

tationen auswendig lernen und beantworten, welche und wieviele Medikamente zu einer bestimmten Tageszeit eingenommen werden sollen. Die Testgruppen, die die Fragen anhand der Matrixdarstellung beantworteten, haben wesentlich öfter korrekte Antworten geliefert. Ihre Antworten waren zu 78% korrekt, während die Testgruppe, die die Listendarstellung benutzte, lediglich zu 56% richtige Antworten lieferte.

Bei der zweiten Untersuchung hat Day [1988] die Effizienz der Benutzung eines Texteditors gemessen. Zwei Gruppen von Probanden wurde die Wirkungsweise von Tastaturbefehlen nach Art des Editors `vi` durch verschiedene Notationen, einer Liste und einer räumlichen Karte (siehe Abb. 2.12) beschrieben. Auch hier mußten die Tastenbefehle gelernt werden. Die Karte führte schon für kleine Editieraufgaben im Durchschnitt zu einer geringeren Anzahl von Tastaturbefehlen. Der Vorteil der Karte war sogar größer als der Nutzen der Vorerfahrung der Probanden.

Day's und die Untersuchungen weiterer Autoren belegen, daß bestimmte Notationen sich für gewisse Fragestellungen besser eignen als andere. Die so erzielbaren Vorteile gelten jedoch nur für die jeweils betrachtete Fragestellung, wie die Ergebnisse von McGuinness zeigen.

McGuinness [1986] hat eine Baum- mit einer Matrixdarstellung anhand mehrerer unterschiedlicher Fragestellungen verglichen. In der Untersuchung visualisieren beide Darstellungen die Verwandschaftsbeziehungen vierer Geschwister, die jeweils vier Kinder haben. Während die Baumstruktur lediglich die familiären Beziehungen visualisiert, stellt die Matrix zusätzlich gleichaltrige Kinder mehrerer Familien miteinander in Beziehung, siehe Abbildung 2.13. Mehrere Gruppen von Probanden sollen auch hier Fragen aus dem Gedächnis beantworten. Die erste Klasse von Fragen kann besonders gut mit den Informationen beantwortet wer-

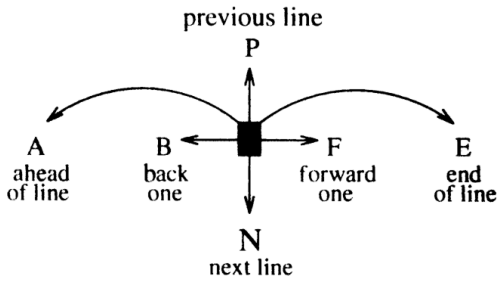
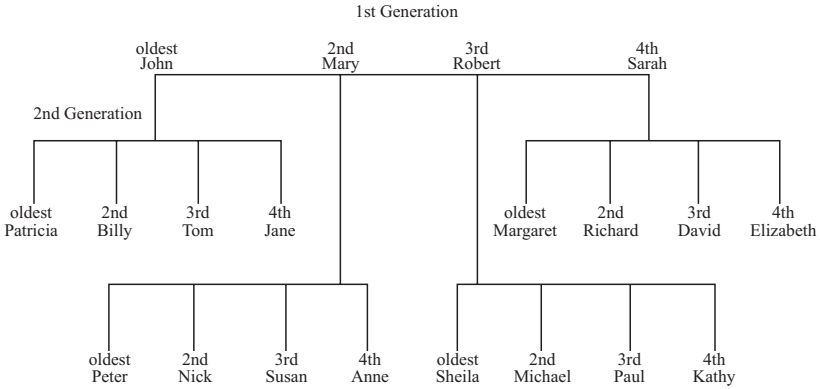


Abbildung 2.12.: 'Räumliche Karte' zur Beschreibung der Cursorbewegungen eines Texteditors nach Day [1988] (zitiert nach Whitley [1997])



(a) Baumdarstellung

1st Generation

2nd Generation	oldest John	2nd Mary	3rd Robert	4th Sarah
oldest	Patricia	Peter	Sheila	Margaret
2nd	Billy	Nick	Michael	Richard
3rd	Tom	Susan	Paul	David
4th	Jane	Anne	Kathy	Elizabeth

(b) Matrixdarstellung

Abbildung 2.13.: Verschiedenartige Darstellungen für Verwandschaftsbeziehungen nach McGuinness [1986] (zitiert nach Whitley [1997])

den, die in der Matrix hervorgehoben werden ('Wer kann mit wem in Urlaub fahren?'), die zweite Klasse von Fragen basiert auf den familiären Beziehungen ('Wer erbt wessen Haus?'). Beide Fragen wurden durch einen Satz von Regeln und Ausnahmen erschwert.

Das Ergebnis dieser Untersuchungen war, daß eine der Fragestellungen den Vorteil der Matrix belegt hat, während die andere Frage keinen Unterschied zwischen Baum und Matrix zeigen konnte. Bei der ersten Frage hat sich weiterhin gezeigt, das der relative Vorteil der Matrix in hohem Maße mit der Anzahl der Ausnahmen zunimmt.

Für die Wahl der Notation für eine visuelle Sprache sieht Whitley [1997] zwei Konsequenzen. Zum einen kann man die Match-Mismatch-Hypothese so interpretieren, daß keine 'optimale' Notation für eine visuelle Sprache gefunden werden kann. Jede Notation hebt bestimmte Informationen auf Kosten anderer hervor und wird sich deshalb als ungeeignet für bestimmte Aufgaben erweisen. Andersherum interpretiert muß also die Art der gestellten Aufgaben bei der Entwicklung einer visuellen Notation mitberücksichtigt werden.

Im weiteren sieht Whitley einen Vorteil für visuelle Notationen. Er besteht darin, daß sie sich besser als textuelle Notationen dazu eignen, Informationen konsistent und strukturiert zu visualisieren, sowie Abhängigkeiten klar darzustellen. Bei geeigneter Wahl der Notation ermöglicht dies einem Benutzer, Probleme effizienter mit der visuellen Sprache zu lösen. Der dadurch erzielbare Vorteil nimmt mit der Komplexität der zu lösenden Probleme zu.

2.2. Grundlegende Methoden und Werkzeuge

Um VLEli zu entwerfen und zu implementieren, benutze ich einige Werkzeuge und Methoden, die von anderen mit unterschiedlichen Zielen entwickelt wurden. In den folgenden Unterabschnitten werde ich einen Überblick über die Arbeiten geben, auf denen ich mit dem VLEli-System aufbaue. Die Diskussionen bleiben dabei bewußt allgemein – eine tiefergehendere Behandlung der zugrundeliegenden Konzepte erfolgt jeweils im Kontext ihrer Benutzung in den folgenden Kapiteln.

Der erste Unterabschnitt hat die Bezeichnung "Muster", womit z.B. Entwurfsmuster für objektorientierte Sprachen gemeint sind. Gamma u.a. [1996] führen damit eine Methode ein, mit der das Entwurfs-Know-How objektorientierter Sprachen dokumentiert werden kann. Die Methode läßt sich, wie ich in Kapitel 3 zeige, auch für visuelle Sprachen verwenden und erleichtert den Entwurf neuer visueller Sprachen.

Im Anschluß daran beschreibe ich attributierte Grammatiken. Mit diesen wird hier nicht nur die Analyse und Weiterverarbeitung visueller Programme implementiert. Sie werden darüberhinaus auch benutzt, um die graphische Darstellung

eines visuellen Programms aus der internen Repräsentation des Programms zu erzeugen. Darauf gehe ich im Detail in Kapitel 4 ein.

Die aus den attributierten Grammatiken generierten Attributauswerter lösen in VLEli für ein visuelles Programm unter anderem die Aufgabe, die Anordnung der graphischen Darstellung zu berechnen. Ein andere Sorte von Werkzeug, das in VLEli hierzu auch verwendet werden kann, ist ein Constraint-Solver. In Abschnitt 2.2.3 gebe ich einen Überblick über das Arbeitsgebiet der constraintbasierten Computergraphik, Details enthält ebenfalls Kapitel 4.

2.2.1. Muster

Die folgenden Absätze definieren den Begriff "Muster" und beschreiben die Anwendung für das Arbeitsgebiet des Entwurfs objektorientierter Programme. Im Anschluß daran gehe ich auf die Anwendung von Mustern beim Entwurf visueller Sprachen ein.

Um den Begriff des "Musters" zu erklären, wird oft die Definition von Christopher Alexander zitiert: "Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so daß Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen." [Alexander u.a. 1977, zitiert nach Gamma u.a. 1996, Seite 2].

Um bedeutende und erfolgreiche Entwurfstechniken für objektorientierte Software wiederverwendbar zu machen, identifizieren und dokumentieren Gamma, Helm, Johnson und Vlissides [1996] Muster, die Experten bei der Entwicklung objektorientierter Softwareentwürfe anwenden. Sie halten diese Techniken in Form von Entwurfsmustern fest, die in ähnlicher Form auch von Coplien [1995] und Pree [1995] erwähnt werden. Ihre Beschreibung besteht aus vier charakteristischen Grundelementen: der Name benennt ein Entwurfsproblem, sowie dessen Lösung mit ein oder zwei Worten. Der Problemabschnitt beschreibt das adressierte Problem und seinen Kontext. Der Lösungsabschnitt beschreibt die Elemente, aus denen die Lösung besteht und ihre Beziehungen untereinander. Der Konsequenzenabschnitt beschreibt schließlich die Vor- und Nachteile der Musteranwendung für den resultierenden Entwurf.

Entwurfsmuster scheinen sich gut zu eignen, um wichtige und wiederkehrende Entwurfstechniken zu dokumentieren. Das zeigt die Anwendung dieser Methode in anderen Arbeitsgebieten, beispielsweise für den Entwurf von Programmierdialekten [Coplien 1992] oder für die Entwicklung von Petri-Netzen [Nædele und Janneck 1998]. Die Anwendung ist auch im Arbeitsgebiet visueller Sprachen erfolgsversprechend: Auch hier existiert Expertenwissen für den Sprachentwurf, das bislang nicht systematisch dokumentiert wurde. Hinweise auf solches Expertenwissen gibt beispielsweise Myers [1994, 1990], der visuelle Sprachen anhand

wesentlicher Charakteristika ihrer Notation gegliedert hat. Er hat damit implizit auch einige Alternativen für verschiedene Arten visueller Notationen genannt, z.B. “data flow graphs”, “spreadsheets” und “jigsaw puzzle pieces”. Auch Whitley [1997] benennt solches Expertenwissen: bei der Aufarbeitung empirischer Untersuchungen verwendet Whitley die Begriffe wie Matrix, Baum und Liste im visuellen Sinne. Die von ihm zusammengestellten Untersuchungen helfen, Kriterien für die sinnvolle Anwendung dieser Notationen zu identifizieren.

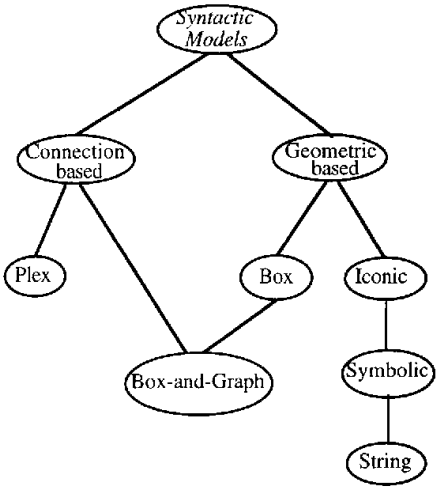


Abbildung 2.14.: Hierarchie syntaktischer Modelle nach Costagliola u.a. [1997a]

Etwas weiter gehen die syntaktischen Modelle von Costagliola u.a. [1997a]. Die Autoren betrachten visuelle Sprachen als aus graphischen Objekten bestehend, zwischen denen Beziehungen durch unterschiedliche Methoden visualisiert werden. Mit Hilfe der syntaktischen Modelle unterteilen sie visuelle Sprachen nach der Methode, die primär zur Darstellung der Relationen verwendet wird. Auf oberster Ebene wird zwischen der Darstellung der Relationen durch Berührung einerseits und durch geometrische Eigenschaften der Darstellung andererseits unterschieden. Nach unten hin verfeinert sich diese Einteilung immer weiter, bis schließlich als Spezialfall textuelle Sprachen in das Framework integriert werden. Abbildung 2.14 gibt einen Überblick über die syntaktischen Kategorien.

Die syntaktischen Modelle liefern eine Übersicht über Techniken, die für die

Visualisierung von Beziehungen beim Entwurf einer Sprache angewendet werden können. Auch wenn noch wichtige Bestandteile, die die Beschreibung der Entwurfsmuster ausmachen, bei den syntaktischen Modellen fehlen, so können sie dennoch als Basis für den Entwurf einer visuellen Sprache nützlich sein.

Mit Hilfe der spezialisierten Editoren, die Costagliola u.a. [1999] entwickelt haben, kann diese Entwurfstechnik auch für die Sprachimplementierung benutzt werden. In den Editoren werden die Beziehungen zwischen den Sprachkonstrukten abstrahiert, indem die Eigenschaften eines syntaktischen Modells ausgenutzt werden. Um eine visuelle Sprache zu spezifizieren, gibt ein Sprachentwerfer eine positionelle Grammatik [Costagliola u.a. 1997b] an, die auf dieser Abstraktion der graphischen Darstellung aufbaut. Aus dieser Grammatik wird ein Parser für die visuelle Sprache konstruiert.

Unbefriedigend ist hierbei jedoch, daß zum Einsatz der Editoren eine Sprache durch genau ein syntaktisches Modell beschrieben werden muß. Für Sprachen, bei denen unterschiedliche Teile der Notation gut durch verschiedene syntaktische Modelle beschrieben werden können, haben die Autoren hybride syntaktische Modelle eingeführt. Bei diesen leidet jedoch die Einfachheit der Spezifikation, was den Vorteilen der Anwendung der syntaktischen Modelle entgegenwirkt.

Glinert und Gonczarowski [1987] untersuchen Eigenschaften visueller Objekte. Sie bemerken, daß Menschen bestimmte Strukturen grundsätzlich als ähnlich empfinden und umschreiben diese Strukturen mit dem Begriff “Klassen”. Ein Beispiel ist die Klasse Rechteck: “Ein spezielles Objekt der Klasse ‘Rechteck’ kann durchgezogen oder gestrichelt, mit dicker oder dünner Linie gezeichnet werden; das Bild könnte auch gefärbt oder um $45°$ gedreht sein usw.” (übersetzt) Die Autoren beschreiben visuelle Objekte, indem sie ihre Klasse nennen und klassenspezifische Attribute angeben. Klassen werden später im System Vampire [McIntyre und Glinert 1992] benutzt, um Aussehen und Verhalten der Piktogramme iconbasierter visueller Sprachen zu beschreiben.

Im weiteren gibt es Ansätze, bei denen Bibliotheken die Implementierung gewisser Aspekte einer visuellen Sprache vereinfachen. Im VIVID-System [Dangberg und Müller 1999] wird etwa für die Spezifikation des Layouts Bezug auf sog. Layoutmanager genommen. Diese kapseln die Aufgabe der Anordnung der visuellen Objekte eines visuellen Programms und ermöglichen es einem Sprachentwerfer, diese wiederkehrenden Aufgaben der Sprachimplementierung durch Bezug auf eine Bibliothek zu lösen. In ähnlicher Weise stellt auch das GenGEd-System [Bardohl 1998] erweiterbare Bibliotheken von “High-Level-Constraints” zur Verfügung, mit denen die Gültigkeit des Layouts der graphischen Objekte beschrieben werden kann. Schließlich werden zur Kapselung wiederverwendbarer Lösungen in Frameworks wie Escalante [McWhirter und Nutt 1994], DV-Centro [Bronwn 1997] oder Unidraw [Vlissides und Linton 1990] die Generalisierungsmechanismen objektorientierter Programmiersprachen verwendet.

Auch die Komponenten einer Bibliothek, etwa die Layoutmanager des VIVID-Systems oder die High-Level-Constraints in GenGEd erleichtern die Sprachimplementierung und in gewisser Weise auch Aspekte des Entwurfs einer visuellen Sprache. Diese Ansätze decken jedoch nur einen geringen Teil der Entwurfsaspekte einer visuellen Sprache ab (VIVID, GenGEd) oder bieten eine relativ niedrige Abstraktionsebene (Escalante, DV-Centro, Unidraw). Dadurch ist der Nutzen dieser Ansätze für den Sprachentwurf begrenzt. Weiterhin beschreibt keiner dieser Ansätze die Vor- und Nachteile der Anwendung der Beschreibungsmittel der jeweiligen Bibliothek.

Im Aufgabenbereich der Konstruktion von Benutzungsschnittstellen wird das Konzept der Muster, beispielsweise unter dem Namen “Widgets” schon länger verwendet. Ein Widget kapselt einen Teil des Aussehens einer graphischen Benutzungsschnittstelle und definiert, welche Editieroperationen anwendbar sind. Im weiteren ist bei Widgets oft auch der Bezug zu einer abstrakten Datenstruktur festgelegt. Bekannte Widgets sind u.a. Eingabefelder (Entry-Widget), Knöpfe (Button-Widget), Listen (Listbox-Widget) oder umschaltbare Knöpfe (Radiobutton-Widget). Sie sind in vielen Programmierumgebungen verfügbar, beispielsweise in Motif [McMinds 1993], Java [Chan und Lee 1998], Visual C++ [Kruglinski 1994] oder Tcl/Tk [Welch 1997]. Damit können Benutzungsschnittstellen auf einer hohen Abstraktionsebene komponiert werden. Teilweise kommen dabei auch visuelle Sprachen zur Anwendung.

Johnson, Nardi, Zarmer und Miller [1993] haben das Konzept der Widgets zu *visuellen Formalismen* weiterentwickelt. Visuelle Formalismen sind “graphische Darstellungen mit wohldefinierter Semantik, die Beziehungen auszudrücken” [Nardi und Zarmer 1993, S. 20, übersetzt]. Johnson u.a. benutzen sie, um Endbenutzer bei der Konstruktion von Anwendungsprogrammen mit graphischer Benutzungsschnittstelle zu unterstützen. Eine wichtige Eigenschaft der visuellen Formalismen ist ihre Spezialisierbarkeit, die auch für das weite Einsatzgebiet verantwortlich ist. Johnson u.a. haben sechs visuelle Formalismen identifiziert und im ACE-System implementiert. Nardi und Zarmer [1993] betonen die Bedeutung der visuellen Formalismen für den Entwurf graphischer Benutzungsschnittstellen.

Die Widgets und die visuellen Formalismen haben das Ziel, Programmierer bei der Implementierung einer graphischen Benutzungsschnittstelle zu unterstützen. Strukturediторn für visuelle Sprachen verfügen auch über Benutzungsschnittstellen, dies ist jedoch nur ein Aspekt der Implementierung einer visuellen Sprache.

2.2.2. Attributierte Grammatiken

Attributierte Grammatiken sind eine formale Methode, um Berechnungen für Informationen zu spezifizieren, die baumartig strukturiert sind. Sie werden in vielen Anwendungsbereichen benutzt, beispielsweise um Analyse- und Synthesephasen

bei der Übersetzung textueller Programme zu spezifizieren [Kastens 1991a], um Datenstrukturen zu visualisieren [Grosch 1999], um graphische Oberflächen zu erzeugen [Kuiper und Saraiva 1998] und um Teile interaktiver Entwicklungsumgebungen, sowohl für textuelle [Reps und Teitelbaum 1989a] als auch für visuelle Sprachen [Backlund u.a. 1990; Chabrier u.a. 1988] zu implementieren.

Vorbedingung für den Einsatz attributierter Grammatiken ist, daß die Bäume, über die Berechnungen ausgeführt werden sollen, durch kontextfreie Grammatiken spezifiziert sind. Eine attributierte Grammatik ordnet den Symbolen einer Grammatik Attribute und den Regeln der Grammatik Berechnungen zu, die die Werte der Attribute bestimmen. Dazu dürfen auch andere Attribute benutzt werden, so daß die Berechnungen Abhängigkeiten zwischen den Attributen spezifizieren. Ein Attributauswerter ist ein Softwaremodul, das die Berechnungen einer attributierten Grammatik für jeden nach der Grammatik beschreibbaren Baum in einer legalen Reihenfolge ausführt.

Die Verfahren zur Auswertung attributierter Grammatiken werden durch zwei wesentliche Faktoren bestimmt. Dies sind zum einen die Mächtigkeit der akzeptierten Grammatikklasse und zum zweiten der für Attributwerte benötigte Speicherplatz. Diese Faktoren sind in hohem Maße voneinander abhängig und beeinflussen beide den Einsatz attributierter Grammatiken für realistische Anwendungen. Eine für den praktischen Einsatz relevante Grammatikklasse ist die Klasse OAG (für *ordered attribute grammar*), die Kastens [1980] eingeführt hat. Diese Klasse ist mächtig genug für den praktischen Einsatz. Weiterhin läßt sich effizient überprüfen, ob eine attributierte Grammatik in der Klasse OAG enthalten ist. Schließlich lassen sich effektive Maßnahmen zur Reduktion des für Attributwerte benötigten Speicherplatzes anwenden [Kastens 1987; Waite 1986].

Einen Überblick über Ansätze zur Auswertung attributierter Grammatiken sowie über Verfahren zur Reduktion des für Attributwerte benötigten Speicherplatzes liefern Deransart, Jourdan und Lorho [1988]. Die Autoren geben auch einen Überblick über Systeme zur Implementierung von Attributauswertern. Ferner ist eine umfassende kommentierte Bibliographie beigefügt. Die Tagungsbände der International Summer School SAGA [Alblas und Melichar 1991] sowie der Workshops zu attributierten Grammatiken [Deransart und Jourdan 1990; Parigot und Mernik 1999] enthalten Informationen über neuere Methoden und Systeme. Schließlich wird eine Bibliographie zu attributierten Grammatiken im WWW geführt und beständig aktualisiert [Parigot 2000].

In diesem Abschnitt gehe ich näher auf die Themenbereiche ein, die für diese Arbeit von größerem Interesse sind. Das sind zum einen die inkrementelle Auswertung attributierter Grammatiken und zum anderen die Sprachmittel, die zur Modularisierung attributierter Grammatiken mit dem Ziel der Wiederverwendung in den Spezifikationssprachen der Generatoren für Attributauswerter umgesetzt wurden.

Attributierte Grammatiken werden in dieser Arbeit auch eingesetzt, um visuelle Programme zu analysieren und weiterzuverarbeiten. Hierzu können Werkzeuge und Module angewendet werden, die auch die Analyse und Übersetzung textueller Sprachen ermöglichen. Sie speichern etwa Eigenschaften der Programmobjekte eines Programms [Waite und Kadhim 1995] oder unterstützen dessen Übersetzung durch Musterbasierte Beschreibung von Zieltext [Ptg 2000]. Solche Werkzeuge und Module sind im Eli-System [Kastens u.a. 1998] integriert. Ein anderes System mit vergleichbarer Werkzeugunterstützung ist PCCTS [Parr 1997].

2.2.2.1. Inkrementelle Attributauswerter

Inkrementelle Attributauswerter werden in Anwendungsbereichen eingesetzt, in denen die Berechnungen attributierter Grammatiken für Bäume ausgewertet werden müssen, die schrittweise geringfügig transformiert werden. Bei inkrementellen Attributauswertern werden dabei nicht alle Berechnungen erneut ausgeführt, sondern nur diejenigen, die von geänderten Baumteilen direkt oder indirekt abhängen. Das Ziel ist es, die Zeit zu verkürzen, die für die Aktualisierung der Attributwerte benötigt wird.

Inkrementelle Attributauswerter werden durch einige Systeme generiert, z.B. von OPTRAN [Wilhelm 1991], vom Synthesizer Generator [Reps und Teitelbaum 1989a], von FNC-2 [Jourdan und Parigot 1991] und von APPLAB [Hedin 1999]. Ein wichtiger Anwendungsbereich inkrementeller Attributauswerter ist die semantische Analyse und Codegenerierung in Entwicklungsumgebungen für Sprachen, insbesondere wenn diese Strukturediteren einsetzen [Reps 1982]. Die Strukturbäume werden dabei durch die Editieroperationen eines Benutzers schrittweise geändert. Attributauswerter berechnen hier semantische Informationen, aus denen etwa eine Liste mit Programmfehlern erzeugt wird. Diese soll meist nach einer Editieroperation aktualisiert werden.

Ein weiterer Anwendungsbereich ist die Implementierung von Optimierungsphasen in Übersetzern, die mit attributierten Grammatiken spezifiziert werden [Wilhelm 1974]. Hierbei ändert jeder Optimierungsschritt das Programm und invalidiert die Analyseinformationen, die zur Planung des nächsten Optimierungsschritts aktualisiert werden müssen.

Eine naives Verfahren zur inkrementellen Attributauswertung könnte zunächst die geänderten Attributwerte ermitteln – sie liegen i.a. an der Grenze zwischen geänderten und nicht geänderten Teilen des Strukturbaums – und anschließend jeweils die Attributewerte neu berechnen, die von geänderten Attributen abhängen. Ein wesentliches Problem des Verfahrens besteht darin, daß dabei einige Attribute u.U. mehrfach berechnet werden müssen.

Die unterschiedlichen Verfahren zur inkrementellen Auswertung unterscheiden sich u.a. darin, wie dieses Problem vermieden wird. Reps, Teitelbaum und

Demers [1983] zeigen, daß inkrementelle Attributauswerter effizient implementierbar sind und ein weites Anwendungsgebiet haben. In diesem Ansatz wird ein "Scheduling-Graph" benutzt, der die direkten und indirekten Abhängigkeiten zwischen den Attributen repräsentiert. Die Attribute werden damit in einer Reihenfolge berechnet, bei der jeder berechnete Attributwert notwendigerweise das Endergebnis ist. Ein Nachteil des Verfahrens ist der zur Speicherung des Graphen zusätzlich benötigte Speicherplatz.

Bei anderen Verfahren kann ein Algorithmus zur nichtinkrementellen Auswertung der Berechnungen einer attributierten Grammatik für die inkrementelle Attributauswertung ausgenutzt werden. Yeh [1983] führt den Algorithmus vor, den Yeh und Kastens [1988] verbessern. Reps und Teitelbaum [1989a] übernehmen ihn später für den Synthesizer-Generator und realisieren weitere Verbesserungen. Die Kernidee ist, daß ein nichtinkrementeller Attributauswerter ausgeführt wird, jedoch ohne dabei tatsächlich Attributberechnungen auszuführen. Erst wenn Attribute von geänderten Attributen abhängen (oder noch nicht berechnet sind) werden Attributberechnungen ausgeführt.

Es gibt einige weitere Probleme bei inkrementeller Attributauswertung. Ein erstes Problem ist, daß alle Attribute bis zum nächsten Lauf gespeichert werden müssen. Im Vergleich zur nichtinkrementellen Attributauswertung verzichtet man also auf den Einsatz wirkungsvoller Verfahren, z.B. Gruppierung von Attributen, die zur Reduktion des Attributspeichers die teilweise disjunkten Lebenszeiten der Attributwerte ausnutzen. Methoden zur Abhilfe sind z.B. die Benutzung eines "Function-Cache" [Pugh 1988; Saraiva u.a. 1996].

Ein weiteres Problem inkrementeller Attributauswerter sind komplexwertige Attribute, vor allem im Zusammenhang mit weitreichenden Abhängigkeiten. Dabei treten Attribute auf, die von sehr vielen Attributen abhängen und ihrerseits und bei vielen Berechnungen benötigt werden. Bei solchen, in der Praxis häufigen Abhängigkeitskonstellationen müssen bei Änderungen oft eine beträchtliche Anzahl von Attributen neu berechnet werden, was den Nutzen der nichtinkrementellen Auswertung schmälert.

Komplexwertige Attribute und weitreichende Abhängigkeiten führen ferner zu einem hohen Ressourcenbedarf für Attributwerte und zu schwer wartbaren Spezifikationen. Um diese Probleme zu vermeiden, läßt sich bei nichtinkrementellen Attributauswertern eine effektive Methode einsetzen. Die zu transportierenden Informationen werden dabei in einer separaten Datenstruktur, etwa einer Definitionstabelle gespeichert [Waite und Kadhim 1995]. Zur Änderung dieser Datenstruktur werden Attributberechnungen mit Seiteneffekten benutzt. Hierbei muß sichergestellt werden, daß der Zugriff auf die Daten in einer richtigen Reihenfolge stattfindet, daß Informationen etwa erst dann gelesen werden, wenn sie zuvor geschrieben wurden.

Hierfür können Attributberechnungen verwendet werden, die lediglich Ab-

hängigkeiten spezifizieren aber keinerlei Informationen transportieren. Derartige Berechnungen lassen sich in einfacher Weise durch Konstrukte wie CONSTITUENTS und INCLUDING [Kastens 1976] spezifizieren und effizient, d.h. ohne Ausführung von Code im Attributauswerter umsetzen. Vorteile dieses Verfahrens sind u.a. eine Vereinfachung der attributierten Grammatik, insbesondere wenn mehrere voneinander unabhängige Informationen transportiert werden müssen. Weiterhin vermeidet diese Methode Attribut-Kopieroperationen. Sie vermindert den Bedarf an komplexwertigen Attributen und reduziert dadurch auch den für Attributwerte benötigten Speicherplatz drastisch.

Inkrementelle Attributauswerter können dieses Verfahren jedoch nicht einsetzen. Der Grund dafür ist, daß Attributberechnungen mit Seiteneffekten nicht angewendet werden können, da man bei inkrementeller Attributauswertung von einer funktionalen, seiteneffekt-freien Semantik der attributierten Grammatik ausgeht. Als Alternative werden bei inkrementellen Attributauswertern etwa Zugriffe auf entfernte Attribute im Strukturbaum eingeführt [Hedin 1991, 1992, 1999]. Hierbei werden zunächst Referenzen auf Strukturbaumknoten in einen Zielkontext transportiert. Anschließend kann über die Referenz direkt auf Attribute des entfernten Strukturbaumknotens zugegriffen werden. Diese Technik hat jedoch den Nachteil, daß sich die Zyklenfreiheit der attributierten Grammatik nicht mehr statisch prüfen läßt und daß die Strategie zur Attributauswertung erst zur Laufzeit bestimmt werden kann.

2.2.2.2. Modularisierung und Wiederverwendung

Beim Entwurf einer Sprache oft zahlreiche bewährte Konzepte existierender Sprachen wiederverwendet und höchstens für einige Teilbereiche neue Lösungen erarbeitet. Im Bereich textueller Sprachen wird beim Sprachentwurf etwa Bezug auf die Algol- oder die C-Verdeckungsregel genommen, um zu definieren, wann zwei identische Namen im Programm dasselbe Programmobjekt identifizieren. Solche Standardtechniken können auch bei der Sprachimplementierung wiederverwendet werden. Hierzu können Module eingesetzt werden, die Bausteine für eine attributierte Grammatik enthalten. Derartige Module können benutzt werden, um eine (neue) Sprache zu spezifizieren. Ich nenne sie im folgenden *Spezifikationsmodule*.

Beim Entwurf einer Sprache werden oft nicht für alle Aspekte der Sprache neue Lösungen erfunden. Vielmehr werden oft zahlreiche bewährte Konzepte existierender Sprachen wiederverwendet und höchstens für einige Teilbereiche neue Lösungen erarbeitet. Im Bereich textueller Sprachen wird beim Sprachentwurf etwa oft Bezug auf die Algol- oder die C-Verdeckungsregel genommen, um zu definieren, wann zwei identische Namen im Programm dasselbe Programmobjekt identifizieren. Solche Standardtechniken können auch bei der Sprachimplementierung wiederverwendet werden. Hierzu können Module eingesetzt werden, die Baustei-

ne für eine attributierte Grammatik enthalten. Derartige Module können benutzt werden, um eine (neue) Sprache zu spezifizieren. Ich nenne sie im folgenden *Spezifikationsmodule*.

Die Grundlage für Spezifikationsmodule bilden Techniken, die der eingesetzte Generator für die Modularisierung und Wiederverwendung von Teilen attributierter Grammatiken bereithält. Sehr gut geeignet für die Bildung wiederverwendbarer Module sind die Techniken des Liga-Systems, die auf der Symbolattributierung und der Vererbung beruhen [Kastens und Waite 1994]. Auf diese Konzepte gehe ich im folgenden detaillierter ein. Im Anschluß gebe ich einen Überblick über zwei weitere Ansätze, die sich zur Implementierung von Spezifikationsmodulen eignen, das sind die "Modular Attribute Grammars" [Dueck und Cormack 1990], die in FNC-2 umgesetzt sind [Jourdan u.a. 1990], sowie die "Composable Attribute Grammars" [Farrow u.a. 1992].

Auf zahlreiche weitere Ansätze zur Modularisierung attributierter Grammatiken gehe ich hier nicht weiter ein, weil diese nicht das Ziel der Wiederverwendung haben. Die Ansätze von Ganzinger und Giegerich [1984], Saraiva und Swierstra [1999] und Roussel u.a. [1994] haben etwa das Ziel der separaten Übersetzung von Modulen. Mernik u.a. [1999] unterstützen mit Modulen die inkrementelle Entwicklung attributierter Grammatiken. Mernik u.a. [1999] wollen schließlich das Verständnis durch Module vereinfachen.

Modularisierung durch Symbolattributierung und Vererbung

In einer attributierten Grammatik werden normalerweise die Attributberechnungen den Grammatikproduktionen zugeordnet. Auf diese Weise können Informationen zwischen den Symbolen einer Produktion ausgetauscht und so auch über größere Distanzen im Strukturbaum transportiert werden. Bei der *Symbolattributierung* des Liga-Systems können Attributberechnungen auch den Symbolen der Grammatik zugeordnet werden. Diese Berechnungen werden im Kontext der Produktionen ausgeführt, die das betreffende Symbol enthalten. Sie bewirken, daß die Berechnungen der attributierten Grammatik in hohem Maße unabhängig gegenüber Änderungen der zugrundeliegenden Grammatik werden. Um weitreichende Zusammenhänge zu spezifizieren, ohne dazu Produktionen anzugeben, können in den Symbolberechnungen die Konstrukte INCLUDING, CONSTITUENTS und CHAIN für entfernte Attributzugriffe eingesetzt werden [Kastens 1976; Lorho 1977]. Reguläre, d.h. Produktionen zugeordnete Attributberechnungen sind so nur noch selten erforderlich.

Das Vererbungskonzept des Liga-Systems beruht auf der Vererbung von Symbolberechnungen. Hierbei werden Symbolberechnungen für Symbole definiert, die in der Grammatik nicht vorkommen. Solche Symbole nennen Kastens und Waite *Rollen*. Wird eine Rolle an ein Grammatiksymbol vererbt, so werden einerseits die Attribute und die Berechnungen der Rolle auch dem Grammatiksymbol zugeordnet. Weiterhin werden aber auch alle Konstrukte für den entfernten At-

tributzugriff ausgeweitet, so daß sich die Abhängigkeiten, die zwischen den Attributen der Rollen definiert sind, auf die Attribute der Grammatiksymbole übertragen. Schließlich lassen sich die ererbten Attributberechnungen durch Symbol-Attributberechnungen und diese wiederum durch Attributberechnungen, die den Produktionen zugeordnet sind, überschreiben.

Abbildung 2.15 zeigt zwei Spezifikationsmodule, die mit Hilfe dieser Techniken realisiert wurden, sowie deren Anwendung für eine Beispielsprache. Die Spezifikationsmodule sind sprachunabhängig und leicht verständlich, da die Berechnungsrollen ('Programm', 'Gültigkeitsbereich', 'definierte' und 'angewandte' Bezeichner) unmittelbar Konzepten des gelösten Problems entsprechen. Darüberhinaus lassen sich die Spezifikationsmodule auch mehrfach anwenden und ein Grammatiksymbol kann Attributberechnungen mehrerer Rollen erben. Das macht die Module flexibel einsetzbar. Im Eli-System [Kastens u.a. 1998] werden Spezifikationsmodule für eine breite Palette von Aufgaben angeboten. Ihr Einsatz hat die Sprachimplementierung in einer Reihe von Anwendungen erheblich vereinfacht.

Außer im Liga-System wird die Vererbung von Attributberechnungen in einer Reihe weiterer Systeme unterstützt, etwa in den Generatoren von Hedin [1989, 1999] und von Koskimies [1991]. Das Vererbungskonzept dieser Generatoren basiert jedoch auf Attributberechnungen, die den Produktionen der Grammatik zugeordnet sind. Die Spezifikationsmodule dieser Systeme sind dadurch in einem höheren Maße Grammatik-abhängig. Auch die Vererbung von Berechnungen mehrerer Module an einen Grammatikkontext ist bei diesen Ansätzen nicht möglich [Hedin 1989].

Modularisierung durch 'Pattern-Matching'

Die Spezifikationsmodule für modulare attributierte Grammatiken [Dueck und Cormack 1990; Jourdan u.a. 1990] (MAG-Module) bestehen aus Regeln. Jede Regel enthält ein Muster für eine Produktion (engl. *production pattern*) und Attributberechnungen. Das Produktions-Muster bestimmt, auf welche Produktionen einer Sprache die Regel angewendet wird. Für alle passenden Produktionen werden die Attributberechnungen der entsprechenden Regel der Sprache zugeordnet. Dabei werden mehrere Definitionen für ein Attribut durch die Reihenfolge der Regeln unterschieden. Außerdem werden bei der Anwendung eines Moduls alle Attributberechnungen ignoriert, die nicht zur Berechnung von Attributen des Wurzel-Kontextes beitragen oder die von undefinierten Attributen abhängen.

Abbildung 2.16 zeigt ein MAG-Modul, das dieselbe Aufgabe wie das Modul aus Abbildung 2.15(b) löst.[6] Das Modul besteht aus vier Regeln. Die erste Regel paßt auf alle Produktionen, die das Symbol `IdUse` auf der linken Seite haben. Die anderen Regeln passen auf Produktionen, die kein, eines oder zwei Symbole auf der rechten Seite aufweisen. Diese Regeln werden benötigt, damit die Attributbe-

[6] Die Attributberechnungen sind in LIDO-Notation notiert, die Typdefinitionen weggelassen.

```
CLASS SYMBOL Root: Env: Environment, GotAllKeys: VOID;
CLASS SYMBOL Range: Env: Environment,
                   GotLocalKeys, GotAllKeys: VOID;
CLASS SYMBOL IdDef, IdUse: Sym: Symbol, Key: DefTableKey;

CLASS SYMBOL Root COMPUTE
  SYNT.Env = NewEnv();
  SYNT.GotAllKeys = "yes";
END;

CLASS SYMBOL Range COMPUTE
  INH.Env = NewScope (INCLUDING (Range.Env, Root.Env));
  SYNT.GotLocalKeys = CONSTITUENTS IdDef.Key;
  INH.GotAllKeys = SYNT.GotLocalKeys
    <- INCLUDING (Range.GotAllKeys, Root.GotAllKeys);
END;

CLASS SYMBOL IdDef COMPUTE
  SYNT.Key = DefineIdn(INCLUDING Range.Env, THIS.Sym);
END;

CLASS SYMBOL IdUse COMPUTE
  SYNT.Key = KeyInEnv(INCLUDING Range.Env, THIS.Sym)
    <- INCLUDING Range.GotAllKeys;
END;
```

(a) Modul für die Namensanalyse nach der Algol-Verdeckungsregel

```
CLASS SYMBOL NoKeyMsg: Key: DefTableKey;

CLASS SYMBOL NoKeyMsg COMPUTE
  IF (EQ(THIS.Key, NoKey),
    message(ERROR, "identifier not defined", 0, COORDREF));
END;
```

(b) Modul für die Meldung undefinierter Bezeichner

```
TREE SYMBOL prog INHERITS Root END;
TREE SYMBOL block INHERITS Range END;
TREE SYMBOL procrange INHERITS Range END;
TREE SYMBOL defid INHERITS IdDef END;
TREE SYMBOL useid INHERITS IdUse, NoKeyMsg END;
```

(c) Anwendung der Module für eine einfache Sprache

Abbildung 2.15.: LIDO-Spezifikationsmodule für die Namensanalyse (vgl. [Kastens und Waite 1994])

```
module CheckDefined
  'IdUse -> ...
      IdUse.Checked = IF(EQ(THIS.Key, NoKey),
            message(ERROR, "identifier not defined", 0, COORDREF));
  A ->
        A.Checked = "yes";
  A -> B
        A.Checked = B.Checked;
  A -> B C
        A.Checked = B.Checked <- C.Checked;
```

Abbildung 2.16.: Spezifikationsmodul für modulare attributierte Grammatiken nach Kastens und Waite [1994]

rechnung der ersten Regel nicht ignoriert wird. Sie bilden das CONSTITUENTS-Konstrukt nach, daß in dieser Spezifikationssprache nicht zur Verfügung steht.

Ein wesentlicher Nachteil dieses Ansatzes ist, daß nicht zwischen Symbolen eines MAG-Moduls und Symbolen der Sprache unterschieden wird. Das kompliziert die Anwendung der MAG-Module, für die etwa zusätzliche Attribute oder Kettenregeln in der Sprache eingeführt werden müssen. Weiterhin fehlt, wie oben erwähnt, ein CONSTITUENTS-Konstrukt. Das hat zur Folge, daß die Module nicht alle Anwendungssituationen abdecken und so manchmal vom Anwender erweitert werden müssen [Kastens und Waite 1994].

Modularisierung durch 'Glue'-Grammatiken

Das zentrale Konzept der 'Composable Attribute Grammars' [Farrow u.a. 1992] (CAG) ist das Zusammenfügen attributierter Grammatiken. Ein CAG-Modul ist prinzipiell eine eigenständige attributierte Grammatik, der eine modulspezifische Grammatik zugrundeliegt. Den Produktionen dieser Grammatik sind Konstruktoren zugeordnet. Ein CAG-Modul wird benutzt, indem konzeptionell durch Aufruf der Konstruktoren in einer sog. Glue-Grammatik ein Strukturbaum nach der Modulgrammatik konstruiert wird. Der Glue-Grammatik liegt die Grammatik für den Anwendungskontext eines Moduls zugrunde, das ist i.a. die Grammatik für eine Sprache. Ist dieser Strukturbaum vollständig erzeugt, so können die dafür definierten Attributberechnungen ausgeführt werden.

Dieses Verfahren ist sehr flexibel einsetzbar. Ein wesentlicher Nachteil ist, daß keinerlei Konstrukte für den entfernten Attributzugriff vorhanden sind. Das macht insbesondere die Anwendung eines Moduls recht komplex, da u.U. sehr vielen Kontexten der Sprachgrammatik teilweise komplexe Attributberechnungen mit Aufrufen der Konstruktoren zugeordnet werden müssen.

2.2.3. Constraints

"Constraint" heißt übersetzt in etwa "Zwang" [Griebel 1996, S. 7] oder "Einschränkung". Angewandt im Bereich der constraintbasierten Computergraphik legen "Constraints" Zwänge für legale Konfigurationen einer graphischen Zeichnung fest, etwa indem sie fordern, daß zwei Linien gleich lang sein müssen oder indem sie fordern, daß zwei graphische Objekte einander berühren müssen. Daraus ergeben sich Vorteile für die Änderung dieser Graphik, da inkonsistente Zustände vermieden werden können.

Im Bereich der Constraint-basierten Graphik werden Constraints für eine graphische Zeichnung durch ein numerisches Formelsystem beschrieben, dessen Lösungen jeweils erlaubte Konfigurationen der Zeichnung definieren. Das dabei auftretende Formel-System heißt auch *Constraint-Netzwerk*. Es wird von einem Werkzeug bearbeitet, das *Constraint-Solver* heißt. Dieses verarbeitet das Constraint-Netzwerk nach einer Änderung und überführt eine u.U. ungültig gewordene Anordnung der graphischen Objekte in eine gültige.

Im SKETCHPAD-System wurden von Sutherland [1963] erstmals Constraints benutzt, um erforderliche Relationen zwischen Punkten, Linien und Kreisen zu spezifizieren und diese Relationen automatisch auch nach Änderungen der Graphik aufrecht zu erhalten. Weitere Constraint-Solver sind u.a. Thinglab [Borning 1979, 1981], später weiterentwickelt zu DeltaBlue [Freeman-Benson u.a. 1990], Garnet [Myers u.a. 1990], Multi-Garnet [Sannella und Borning 1992] und Parcon [Griebel 1996]. Constraints werden auch in anderen Bereichen eingesetzt: Marriott und Stuckey [1998] liefern einen Überblick mit Schwerpunkt bei der logischen Programmierung.

Für den im Bereich der Computergraphik eingesetzten Constraint-Solver formuliert Griebel [1996] einige Anforderungen, die ich hier kurz wiedergeben will. Die wichtigste dieser Forderungen ist dabei sicherlich die Forderung des "least astonishment". Sie bedeutet, daß bei unterspezifizierten Constraint-Netzwerken (solche mit mehr als einer Lösung) jeweils die Änderung durchzuführen ist, die einen Benutzer am wenigsten überrascht. Das wird hier so ausgelegt, daß das Layout sich einerseits insgesamt möglichst wenig ändert und daß zum zweiten eine Änderung des Benutzers möglichst nicht rückgängig gemacht wird. Weitere Forderungen betreffen Eigenschaften des Constraint-Solvers, die zur effizienten Beschreibung legaler Anordnungsalternativen benötigt werden: *ungerichtete Constraints* führen zu deklarativen Beschreibungen (z.B. Linie a und Linie b sind gleich lang), aber oft auch zu zyklischen Abhängigkeiten. Im weiteren kommt man mit einem gewissen Grundrepertoire an "Basis-Constraints", z.B. Addition und Subtraktion" aus, benötigt aber Ungleichungen, Disjunktionen sowie eine Funktion zur Formulierung von Constraints über den euklidischen Abstand.

Wichtig ist schließlich auch die Geschwindigkeit des Constraint-Solvers. Griebel [1996, S. 3] fordert, daß "in einem interaktiven, constraintbasierten System (...)

eine Lösung in Echtzeit ermittelt werden (muß)", wobei eine Obergrenze für Echtzeitfähigkeit bei Antwortzeiten von einer Sekunde angesehen wird.

2.3. Entwicklungsumgebungen für visuelle Sprachen

Um diagrammbasierte visuelle Sprachen durch Generatoren oder von Hand zu implementieren, werden im Wesentlichen zwei Ansätze benutzt. Die eine Variante verwendet einen angepaßten Universaleditor und setzt einen visuelle Parser ein, die zweite Variante implementiert einen Strukturedítor. Beide Ansätze sind auch für textuelle Sprachen gebräuchlich.

In den folgenden Unterabschnitten diskutiere ich diese Ansätze. Da mit VLEli ebenfalls Strukturedítoren implementiert werden, gehe ich ausführlicher auf Strukturedítoren ein. Dazu diskutiere ich in Abschnitt 2.3.3 ihre Grundlagen und beschreibe anschließend einige Systeme, die Editoren für visuelle Sprachen erzeugen können.

Auf die Implementierung von Piktogrammsprachen gehe ich dagegen nicht ausführlicher ein. Piktogrammsprachen (oder Piktogrammsätze) bezeichnen "visuelle Ausdrücke mit Piktogrammen, bei denen die Positionen der Piktogramme eine wesentliche Rolle spielen" [Myers 1994, zitiert nach Schiffer 1998, S. 126]. Der Leser sei hier auf die Arbeiten von McIntyre und Glinert [1992], Crimi u.a. [1990], Hirakawa u.a. [1990] und Chang u.a. [1989] verwiesen.

2.3.1. Universaleditor und visuelle Parser

Analog zu einer weitverbreiteten Methode der Implementierung textueller Sprachen kann man versuchen, auch für visuelle Sprachen einen sprachunabhängigen Universaleditor zu benutzen. Um die visuellen Programme zu prüfen und weiterzuverarbeiten, kann ebenfalls ein Parser, in diesem Fall ein visueller Parser benutzt werden.

Die Übertragung dieser Implementierungsstrategie von textuellen auf visuelle Sprachen führt zu zusätzlichen Problemen. Zum ersten gibt es kein allgemein anerkanntes sprachunabhängiges Dateiformat für visuelle Sprachen, das zum Austausch von Informationen zwischen einem Universaleditor und einem Parser aber benötigt wird. Zum zweiten ist ein komfortabler visueller Editor in weit größerem Maße, als dies bei textuellen Sprachen der Fall ist, sprachabhängig: um beispielsweise Platz für ein neu eingefügtes Sprachkonstrukt zu schaffen, müssen etwa vorhandene Sprachkonstrukte in einer komplexen Weise neu angeordnet werden. Das kann durch spezialisierte sprachabhängige Editieroperationen extrem vereinfacht werden.

Frühe Ansätze zur generatorgestützten Implementierung visueller Sprachen lösen diese Probleme, indem sie die unterstützte Klasse visueller Sprachen einschränken, z.B. auf Graphen. Ein Beispiel ist Visual Programmers Workbench [Golin u.a. 1989]. Der sprachunabhängige graphische Editor unterstützt Rechtecke, Kreise, Linien, Pfeile und Text und speichert visuelle Programme in einer Textdatei.

In neueren Ansätzen werden Entwicklungsumgebungen erzeugt, die Editor und Analyse / Weiterverarbeitung integrieren sowie die Editoren auf die zu implementierende visuelle Sprache spezialisieren. Dazu können oft spezielle visuelle Objekte aus einem gewissen Grundrepertoire zusammengesetzt werden. Weiterhin werden meist inkrementelle visuelle Parser benutzt, um Benutzer unverzüglich über die Korrektheit der Editieroperationen informieren zu können.

Beispiele für derartige Generatoren sind VisPro [Zhang und Zhang 1998], VLCC [Costagliola u.a. 1999] und DiaGen [Minas 1998; Minas und Viehstaedt 1995]. VisPro setzt einen generischen Editor ein, der durch spezielle visuelle Objekte und durch sprachabhängige Editieroperationen an eine visuelle Sprache angepaßt werden kann. VLCC beruht auf einer hierarchischen Unterteilung wesentlicher Aspekte der graphischen Notation einer Sprache[Costagliola u.a. 1997a]. Für jeden Eintrag der Hierarchie existiert ein spezialisierter graphischer Editor sowie eine angepaßte Art der Spezifikation des visuellen Parsers.

Das Penguins-System [Chok und Marriott 1998; Chok u.a. 1999] realisiert eine weitere Verbesserung: Durch den inkrementellen Parser wird ein Constraint-Netzwerk generiert, mit dessen Hilfe bereits erkannte Bildstrukturen gemeinsam bearbeitet werden können. Diese Art der Sprachimplementierung wird von den Autoren “Intelligent Diagram Editor” genannt.

2.3.1.1. Visuelle Parser

Visuelle Parser überprüfen, ob eine Zeichnung ein Programm einer bestimmten visuellen Sprache darstellt. Ist dies der Fall, so kann er eine strukturelle Abstraktion des Programms erzeugen. Diese stellt die Basis für die Analyse und Weiterverarbeitung des Programms dar. Ein visueller Parser ist also ein sinnvolles Werkzeug, um einen Universaleditor für eine visuelle Sprache zu ergänzen. Um visuelle Parser zu erzeugen, werden in zahlreichen Ansätzen verschiedene Spezifikationstechniken eingeführt.

Marriott, Meyer und Wittenbourg [1998] geben einen Überblick über die Ansätze, die sie in drei Gruppierungen aufteilen. In der ersten und umfangreichsten Gruppe finden sich Ansätze, die auf Grammatiken basieren und somit auf den Beschreibungstechniken aufbauen, die für textuelle Sprachen bereits erfolgreich verwendet werden. Da bei visuellen Sprachen die Reihenfolge der Symbole auf der rechten Seite einer Produktion aber meist keine Semantik trägt, sprechen Marriott u.a. hier von Multimengen-Grammatiken (engl. *multiset-grammars*). In den

anderen Gruppen finden sich Spezifikationstechniken, die auf logischen bzw. algebraischen Formalismen aufbauen.

Einige der Spezifikationstechniken werden von Marriott und Meyer [1998] in eine Hierarchie einsortiert. Marriott und Meyer erwähnen dabei, daß selbst die ausdrucksstärksten Spezifikationstechniken nicht in der Lage sind, einige interessante visuelle Sprachen zu beschreiben. Das Analyseproblem ist dabei bereits für eingeschränkte Sprachklassen sehr aufwendig. Dies ist wohl auch die Ursache für die Vielzahl der Ansätze: Man versucht Techniken zu erarbeiten, die einerseits mächtig genug sind, um einen Großteil der interessierenden visuellen Sprachen damit zu spezifizieren und für die sich andererseits effiziente visuelle Parser konstruieren lassen.

2.3.2. Strukturediteren

Bei jeder Art der Implementierung visueller Sprachen treten in irgendeiner Form zwei verschiedene Repräsentationen der visuellen Programme auf. Die eine Repräsentation ist die graphische Darstellung des Programms, die andere eine abstrakte Repräsentation, die die Programmstrukturen reflektiert und die sich besser als die graphische Darstellung eignet, um semantische Eigenschaften des visuellen Programms zu prüfen und das Programm weiterzuverarbeiten [Rekers und Schürr 1996].

Bei dem oben beschriebenen Ansatz erzeugt ein Benutzer im Editor die graphische Darstellung für ein visuelles Programm. Ein visueller Parser transformiert diese Darstellung in die abstrakte Struktur. Bei einem syntaxgesteuerten Editor oder Strukturediteur editiert der Benutzer dagegen die abstrakte Struktur eines visuellen Programms. Das bedeutet zum einen, daß das editierte Programm jederzeit syntaktisch korrekt ist. Zum zweiten kann das visuelle Programm jederzeit analysiert und weiterverarbeitet werden, ohne zunächst mit einem visuellen Parser die abstrakte Struktur zu ermitteln. Das wirkt sich auch auf die Benutzung des Strukturediteurs aus, z.B. weil durch Analyseinformationen auch die Editieroperationen der generierten Editoren verbessert werden können. [Reps und Teitelbaum 1989a].

Der Benutzer eines Strukturediteurs sieht eine Visualisierung der abstrakten Struktur, die nach Änderungen jeweils aktualisiert wird. Die Kontexte der abstrakten Struktur, die noch nicht zu terminalen Strukturen verfeinert sind, werden dabei meist durch Platzhalter dargestellt. An diesen Platzhaltern kann das visuelle Programm verfeinert werden, z.B. indem Produktionen einer Baum- oder Graphgrammatik angewendet werden.

Diese Methode der Implementierung visueller Sprachen hat einige Vorteile. Zum einen lassen sich anhand der abstrakten Struktur relativ leicht strukturelle Transformationen realisieren. Diese haben bei visuellen Sprachen eine wesentlich

größere Bedeutung als bei textuellen Sprachen: um strukturelle Transformationen "von Hand" durchzuführen, muß ein Benutzer eine Reihe von Layoutänderungen vornehmen, die bei visuellen Darstellungen viel komplexer als bei textuellen Programmen sind. Zum zweiten sind visuelle Parser sehr aufwendig. Sie lassen sich effizient meist nur für eine eingeschränkte Klasse visueller Sprachen erzeugen [Marriott und Meyer 1998]. Dies sind wahrscheinlich die Gründe dafür, daß kommerziell erfolgreich eingesetzte visuelle Sprachen meistens durch Strukturediteren implementiert sind.

Strukturediteren werden auch für die Implementierung textueller Sprachen benutzt, haben sich dort aber nicht allgemein durchgesetzt. Die von Minör [1992] dafür angeführten Gründe werden teilweise auch gegen den Einsatz von Strukturediteren für visuelle Sprachen benutzt. Die Argumente sind etwa, daß die Art der Interaktion einen Benutzer in der Reihenfolge seiner Arbeit beschränke oder daß sich einige strukturelle Änderungen durch einen Universaleditor einfacher durchführen lassen.

Die Argumente betreffen also Aspekte der Benutzbarkeit der visuellen Strukturediteren. Sie werden aber meines Wissens nach bislang nicht durch empirische Untersuchungen untermauert, eher das Gegenteil ist der Fall. Der Generator VPE [Grant 1998] beruht etwa auf Prinzipien, die Minör [1992] für die Implementierung benutzerfreundlicher Strukturediteren erarbeitet hat. Grant mißt die Geschwindigkeit der Programmeingabe für die Sprache VML [Cardelli 1983] gegen die Eingabe äquivalenter textueller Programme mit einem Texteditor. Er kommt zu dem Ergebnis, daß die visuelle Programmierumgebung im Durchschnitt lediglich um den Faktor 1,4 bis 2,7 langsamer ist.

Dies ist ein sehr gutes Ergebnis. Es entkräftet die Argumente gegen Strukturediteren zwar noch nicht, zeigt aber, daß sich damit effizient benutzbare Sprachimplementierungen erzeugen lassen. Obwohl von einigen Autoren Effizienzmessungen durchgeführt werden [Chok und Marriott 1998; Minas und Viehstaedt 1995], existieren Untersuchungen der Benutzbarkeit meines Wissens nach bislang nicht für Sprachen, die durch universelle visuelle Editoren implementiert wurden. Entsprechend können auch die Techniken der Implementierung visueller Sprachen noch nicht anhand der Benutzbarkeit miteinander verglichen werden.

2.3.3. Grundlagen von Strukturediteren

In diesem Abschnitt gehe ich auf wesentliche Aspekte von Strukturediteren für visuelle Sprachen ein und verdeutliche einige wichtige Unterschiede gegenüber der Konstruktion von Strukturediteren für textuelle Sprachen. Dabei soll die Betrachtung auf die Grundlagen beschränkt werden, die für den Schwerpunkt dieser Arbeit von Bedeutung sind. Einige wichtige Aspekte, z.B. Aspekte der Mensch-Maschine-Schnittstelle von Strukturediteren, bleiben deshalb hier unerwähnt.

Als abkürzende Notationen sollen im folgenden Strukturediteren für visuelle Sprachen “visuelle Strukturediteren” genannt werden. Mit “textuellen Strukturediteren” meine ich entsprechend Strukturediteren für textuelle Sprachen.

2.3.3.1. Repräsentation der Programme

Ein wesentlicher Aspekt von Strukturediteren ist der Aufbau der Programmrepräsentation, die dem Editor zugrundeliegt. Strukturediteren für textuelle Programmiersprachen basieren meistens auf Strukturbäumen [Hood 1985]. Beispiele dafür sind Generatoren wie der Synthesizer Generator [Reps und Teitelbaum 1989a], PSG [Bahlke und Snelting 1992] oder der universelle Strukturedit or Mjøllner ORM [Minör und Magnusson 1996]. Es gibt aber auch Systeme, die andere Repräsentationen, z.B. Graphen benutzen [Engels u.a. 1986]. Diese eignen sich oft auch für die Anwendung in anderen Bereichen, z.B. für strukturierte Textdokumente.

Werden Strukturbäume benutzt, so wird eine textuelle Sprache oft durch eine kontextfreie Grammatik spezifiziert. Die Grammatiken sind manchmal um spezielle Konstrukte erweitert, beim Synthesizer Generator sind etwa Erweiterungen für optionale Grammatiksymbole und für Listen vorhanden [Reps und Teitelbaum 1989b, S. 19]. Durch die Produktionen wird aber nicht nur die Repräsentation der Programme festgelegt. Auch viele andere Teile sprachabhängige Teile des Editors wie z.B. die Editieroperationen werden durch sie bestimmt. Schließlich liefern die Strukturbäume, die durch die Produktionen beschrieben werden, einen natürlichen Geltungsbereich für Interaktionsoperationen wie “Ausschneiden” und ”Einfügen” (cut & paste).

Einige Generatoren für visuelle Strukturediteren verwenden ebenfalls Strukturbäume, um visuelle Programme zu repräsentieren, z.B. VPE [Grant 1998] und Gigas [Chabrier u.a. 1988; Lextrait und Zarli 1990]. Dies wird jedoch im allgemeinen als nicht ausreichend erachtet. Ein Grund dafür ist, daß bei visuellen Sprachen sehr viel mehr Möglichkeiten existieren, um die Beziehungen in einem Programm zu visualisieren. So ist es durchaus gebräuchlich, hierzu Linien zu benutzen, die zwischen den in Beziehung stehenden Programmstellen gezeichnet werden. Das hat zwei Auswirkungen. Einerseits muß die Editieroperation ‘Einfügen einer Linie zwischen zwei Programmkonstrukten’ spezifiziert werden, was nicht durch eine kontextfreie Produktion erreicht werden kann. Andererseits muß bei der Visualisierung des Programms der Verlauf der Linie an ein evtl. geändertes Layout angepaßt werden. Dazu müssen meistens Informationen zwischen den an Beziehungen beteiligten Programmstellen ausgetauscht werden.

Das motiviert den Wunsch, alle Arten von Beziehungen explizit in einer Grammatik spezifizieren und in der Repräsentation der Programme speichern zu können. Ein Beispiel dafür ist der Generator LOGGIE [Backlund u.a. 1990], bei dem Strukturbäume um Ausdrucksmittel für weitere Beziehungen ergänzt wer-

den. Sie werden dort "*Garlands*" genannt. Auch das Konzept für den Generator GEGS [Szwillus 1996] sieht vor, Strukturbäume um zusätzliche, hier *Nichtstrukturkanten* genannte Kanten zu erweitern. Bei diesen Ansätzen werden die Editieroperationen nicht mehr allein durch die Produktionen einer kontextfreien Grammatik spezifiziert, da zusätzliche Ausdrucksmittel vorhanden sind, um Garlands (bei LOGGIE) oder Nichtstrukturkanten (bei GEGS) zu erzeugen. Dadurch wird die kontextfreie Grammatik im Prinzip zu einer Grammatik für Graphen erweitert, wobei die 'Baumkanten' oft von 'anderen Kanten' unterschieden werden.

In anderen Ansätzen werden Graphgrammatiken benutzt, um die Programmrepräsentation und die Editieroperationen einheitlich zu beschreiben. Beispiele sind GenGEd [Bardohl 1998], der im folgenden PROGRESS genannte, aber anonyme Ansatz von Rekers und Schürr [1996], sowie die Ansätze von Arefi u.a. [1990] und Göttler [1992]. Die Grammatiken, auf denen diese Ansätze beruhen, sind im Detail recht verschieden, beruhen jedoch meist auf kontextsensitiven Produktionen. Weiterhin lassen sich auch oft in irgendeiner Weise Vorbedingungen angeben, deren Erfüllung die Anwendbarkeit einer Produktion bestimmt. Arefi u.a. [1990] formulieren so Graphgrammatik-Produktionen, die komplexe Transformationen wie die If-To-While-Transformation beschreiben.

Ein Problem der Benutzung von Graphen ist, daß diese nicht in einfacher Weise existierende Werkzeuge für die Analyse und Weiterverarbeitung von Programmen integrieren können, da diese oft auf abstrakten Strukturbäumen aufsetzen.

Die Informationen, die bei der semantischen Analyse extrahiert werden, werden nicht nur für Prüfungen und für die Weiterverarbeitung der Programme benutzt. Sie werden auch benötigt, um beispielsweise spezielle Editieroperationen zu implementieren. Beim Synthesizer Generator [Reps und Teitelbaum 1989a, S. 137] kann etwa ein Prozeduraufruf zusammen mit einer passenden Anzahl an Platzhaltern für die Argumente erzeugt werden. Das PSG-System [Bahlke und Snelting 1992] unterstützt ähnliche Editierhilfen, die sinnvoll auch für visuelle Programme angewendet werden könnten.

Eine andere Methode, mit der Strukturediteren implementiert werden können, sind objektorientierte Klassenbibliotheken. In einem System wird hierbei meist ein Framework bereitgestellt, das eine Bibliothek vordefinierter Klassen enthält. Ein Sprachentwerfer implementiert einen Editor für eine visuelle Sprache, indem er die Klassen des Frameworks spezialisiert. Die durch den Editor implementierte Sprache wird dabei oft implizit durch die Spezialisierung festgelegt. Systeme, sind etwa Escalante [McWhirter und Nutt 1992, 1994], DV-Centro [Bronwn 1997] und Unidraw [Vlissides und Linton 1990].

2.3.3.2. Layout

Ein weiterer wichtiger Unterschied zwischen textuellen und visuellen Strukturediteren besteht im Layout der Programme. Ein textueller Strukturediteur übernimmt

meist vollständig die Aufgabe der Berechnung einer sinnvollen Anordnung. Das dazu eingesetzte Werkzeug wird oft “Unparser” genannt. Dieser wird durch Layoutregeln gesteuert, die im Strukturedi tor entweder fest vorgegeben sind oder die ein Benutzer in einem Konfigurationsdialog einstellen kann. Bei Strukturedi toren, die durch den Synthesizer Generator [Reps und Teitelbaum 1989b] hergestellt werden, geht dies etwa so weit, daß in unterschiedlichen Fenstern verschiedene Layouts derselben abstrakten Struktur dargestellt werden können.

Einige Generatoren visueller Strukturedi toren benutzen eine ähnliche Vorgehensweise. Ein Beispiel ist VPE [Grant 1998]. Hier wird die graphische Darstellung mittels einer festen Baumdurchlaufsstrategie aus der abstrakten Struktur gewonnen. Das ist für Sprachen ausreichend, deren Programmkonstrukte nur wenig Layoutfreiheiten bieten. Beispiele sind die Nassi-Shneiderman-Diagramme [Nassi und Shneiderman 1973] und VML [Cardelli 1983].

Um jedoch Sprachen zu implementieren, deren graphische Darstellung z.B. auf Graphen basiert, ist dies nicht ausreichend. Beispiele sind die Zustandsdiagramme oder die Klassendiagramme aus UML. Hier ist es schwierig, ohne weitere Layoutangaben des Benutzers eine übersichtliche und verständliche Darstellung des visuellen Programms aus seiner abstrakten Struktur zu erzeugen. Um derartige Sprachen zu implementieren, benutzt man besser ein *manuelles Layout*. Der Benutzer einer Sprache kann dabei das Layout einiger Sprachkonstrukte selbst bestimmen. Er kann dann z.B. die Klassen eines Klassendiagramms frei positionieren, ohne sich auf die Güte eines Graph-Layouters verlassen zu müssen.

Problematisch beim manuellen Layout ist, daß ein Benutzer auch ungültige Layouts erzeugen kann. Solche Diagramme sind irreführend, denn sie geben die abstrakte Struktur eines Programms nicht korrekt wider. Trotzdem sind derartige Editieroperationen manchmal nützlich, um ein Diagramm neu anzuordnen. Eine Verbesserung des Verhaltens der generierten Editoren läßt sich hier durch den Einsatz eines Constraint-Solvers erzielen. Dieser kann prüfen, ob eine Editieroperation des Benutzers die Gültigkeitsregeln für das Layout verletzt. Gegebenenfalls kann ein ungültig gewordenes Layout automatisch korrigiert werden. So kann etwa verhindert werden, daß zwei Klassen eines Klassendiagramms übereinander gelegt werden. Viele der neueren Generatoren für Strukturedi toren verwenden diese Strategie, z.B. GenGEd [Bardohl 1998] und Progress [Rekers und Schürr 1996].

2.3.4. Generatoren von Strukturedi toren

Im vorigen Abschnitt habe ich einige grundlegende Aspekte von Strukturedi toren für visuelle Sprachen diskutiert. Dabei sind bereits wichtige Eigenschaften einer Reihe von Generatoren identifiziert worden.

In den folgenden Abschnitten gehe ich nochmals auf einige der wichtigeren

Ansätze ein. Die Beschreibungen sollen einen Überblick über die Generatoren und die zur Spezifikation visueller Sprachen verwendeten Methoden geben.

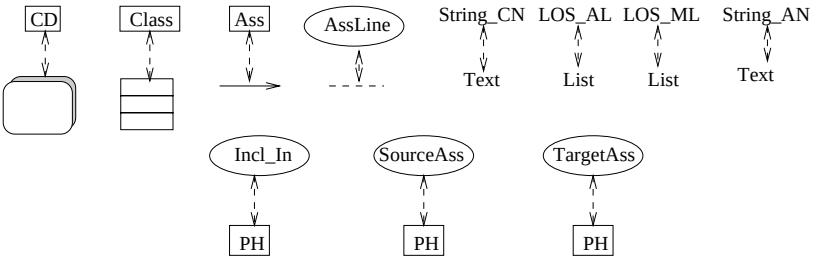
2.3.4.1. PROGRESS

Wie bereits oben erwähnt, beobachten Rekers und Schürr [1996], daß für die Implementierung einer visuellen Sprache in irgendeiner Form zwei Repräsentationen des visuellen Programms auftreten. Die Repräsentationen werden von Rekers und Schürr *Spatial Relations Graph* (SRG) und *Abstract Syntax Graph* (ASG) genannt und – wie die Namen schon erkennen lassen – beide durch Graphen modelliert. Während die Knoten im ASG Sprachkonstrukte und die Kanten die Beziehungen zwischen den Sprachkonstrukten modellieren, repräsentieren die Knoten im SRG einzelne Objekte der graphischen Darstellung (Rechteck, Kreis, Text, ...) und die Kanten Beziehungen zwischen ihnen (berührt, geschachtelt in, neben, ...).

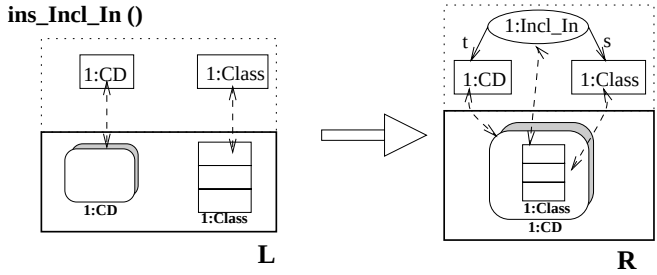
Der vorgestellte Ansatz sieht vor, daß beide Strukturen explizit und persistent gespeichert werden. Die Editieroperationen einer visuellen Sprache werden durch kontextsensitive Graphgrammatik-Produktionen für den ASG und den SRG spezifiziert. Die Produktionen sind jeweils paarweise miteinander gekoppelt. Auf dieser Basis können drei Arten von Editieroperationen unterschieden werden. Syntaxgesteuerte Editieroperationen werden realisiert, indem eine Produktion parallel im ASG und im SRG durchgeführt wird. Der geänderte SRG enthält dabei u.U. neue graphische Objekte oder geänderte Beziehungen, die im Layout reflektiert werden, indem ein Constraint-Netzwerk erzeugt und durch einen Constraint-Solver validiert wird.

Die anderen Arten von Editieroperationen sind Änderung des Layouts und freies Editieren. Bei Layoutänderungen kann der Benutzer einzelne Objekte der graphischen Darstellung verschieben. Der Constraint-Solver stellt dabei sicher, daß alle spezifizierten Beziehungen eingehalten werden. Die abstrakte Struktur wird dabei nicht geändert. Beim freien Editieren kann der Benutzer direkt die graphische Darstellung ändern. Nach Vollendung der gewünschten Manipulationen wird die graphische Darstellung dann durch einen visuellen Parser analysiert. Vorher muß die graphische Darstellung jedoch analysiert werden, um festzustellen, welche Beziehungen zwischen den Objekten in der geänderten Darstellung bestehen.

Einen ersten einsetzbaren Prototyp, der diesen Ansatz umsetzen soll, haben die Autoren im Rahmen einer studentischen Projektgruppe mit Hilfe des Werkzeugsystems PROGRESS, dem Constraint-Solver Delta-Blue [Freeman-Benson u.a. 1990] und weiteren Werkzeugen realisiert. Das Projekt wurde unter Engels fortgesetzt [Andries u.a. 1998], bis dieser die Universität Leiden in Holland verließ.



(a) Alphabet für UML-Klassendiagramme



(b) Visuelle Spezifikation einer Regel

Abbildung 2.17.: Visuelle Spezifikation visueller Sprachen bei GenGEd nach Bardohl [1998, Fig. 9, 11]

2.3.4.2. GenGEd

GenGEd [Bardohl 1998, 2000; Niemann und Bardohl 2000] ist ein universeller Strukturediteor für visuelle Programme. Er ist durch Spezifikationen, die von GenGEd interpretiert werden, an eine visuelle Sprache anpaßbar. Der Aufbau von GenGEd hat große Ähnlichkeit mit dem Ansatz von Rekers und Schürr. Auch hier wird die abstrakte Struktur und die graphische Darstellung durch Graphen repräsentiert. Bardohl benutzt zu deren Spezifikation jedoch algebraische Techniken. Bardohl und Taentzer [1997] nennen zwei wesentliche Unterschiede: Zum einen unterscheiden sich der ASG und der SRG bei GenGEd lediglich dadurch, daß im SRG zusätzliche Attribute enthalten sind, die die graphischen Objekte und die Relationen repräsentieren. Zum anderen kann bei GenGEd eine visuelle Sprache graphisch spezifiziert werden.

Die graphische Spezifikation basiert auf zwei Beschreibungsmitteln. Im sog. *Alphabet Editor* wird das Alphabet einer Sprache spezifiziert. Abbildung 2.17(a) zeigt etwa das Alphabet für eine eingeschränkte Version der UML-Klassendiagramme. Die Spezifikation besteht aus zwei Teilen. Den jeweils unten abgebildeten graphischen Symbolen ist oben ihre Repräsentation im abstrakten Syntaxgraph zugeordnet. Die in der unteren Reihe dargestellten Symbole haben als graphische Repräsentation einen Platzhalter und werden lediglich für die Spezifikation benötigt.

Im Rahmen dieser Zuordnung werden auch Layoutbedingungen spezifiziert, die im Editor später durch Constraints repräsentiert werden. Ein Benutzer kann dabei auf eine erweiterbare Bibliothek von Constraint-Beziehungen zurückgreifen, die auf einer tieferen Ebene mit Hilfe des Parcon-Systems [Griebel 1996] und seiner (unbenannten) Spezifikationssprache realisiert werden.

Für den zweiten Teil der Spezifikation wird der sog. *Rule Editor* benutzt. Dabei kann der Sprachentwerfer aus den Symbolen des Alphabets Regeln aufbauen. Diese können einerseits als (kontextsensitive) Graphgrammatik-Produktionen verstanden werden. Die Regeln spezifizieren aber gleichzeitig eine bestimmte Änderung der graphischen Darstellung. Eine Regel, mit der eine eingeschränkte Version der UML-Klassendiagramme spezifiziert wird, ist in Abbildung 2.17(b) wiedergegeben. Die Regel 'verbindet' eine (neue) Klasse mit einem Klassendiagramm.

2.3.4.3. VPE

Auch VPE [Grant 1998] ist ein universeller Strukturediteor, der Spezifikationen visueller Sprachen interpretiert. VPE kann auch Programme mehrerer verschiedener Sprachen zugleich editieren. Interessante Aspekte des Ansatzes sind, daß eine recht hohe Ausdruckskraft für geschachtelte Darstellungen erzielt wird und daß die Spezifikation einer Sprache recht kompakt ist. Die Implementierung der Nassi-Shneiderman-Diagramme benötigt beispielsweise nur etwa 210 Zeilen Quelltext (ohne Kommentar- und Leerzeilen). Die Spezifikation für das IF-Konstrukt ist in Abbildung 2.18 wiedergegeben.

Um diese Eigenschaften umzusetzen, wurde die Klasse der implementierbaren visuellen Sprachen beschränkt. Einerseits können in der Darstellung keine Linien verwendet werden, andererseits wird die Anordnung der graphischen Darstellung immer automatisch berechnet – es werden keine Modifikationen des Layouts durch den Benutzer unterstützt. Aufgrund dieser Einschränkungen kann ein Strukturbaum gewählt werden, um die Programme zu repräsentieren. Die graphische Darstellung wird durch einen festen Baumdurchlauf erzeugt.

```
(Object Choice
        (Contains (Box 15 35 15 20)
                ParentScaleX
                (Box 19 38 19 18) (Accepts Text)
        )

        (Thickness 2)
        (NoLineScale)
        (Box 40 40 40 40)
        (Line (From 40 0) (To 40 0))
        (Line (From 40 40) (To 0 0))
        (Line (To 40 40))
        (Text (Center 30 10) T)
        (Text (Center 30 10) F)
        (Text (Center 0 12) ?)
        (Line (From 0 0) (To 0 40))

        (Contains (Box 35 5 5 35)
                NoParentScaleXY
                ClientScaleX
                ClientScaleY
                Left
                (Accepts Text)
                (Box 40 0 0 40)
                (Accepts Choice Seq While Until Parallel Case)
        )
        (Contains (Box 5 5 35 35)
                NoParentScaleXY
                ClientScaleX
                ClientScaleY
                Right
                (Accepts Text)
                (Box 0 0 40 40)
                (Accepts Choice Seq While Until Parallel Case)
        )
)
```

Abbildung 2.18.: Spezifikation des IF-Konstrukts der Nassi-Shneiderman-Diagramme in VPE

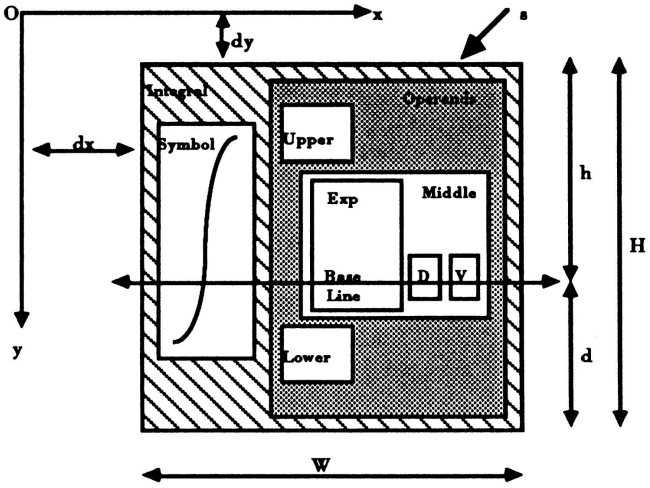
2.3.4.4. GIGAS

Der Generator Gigas [Chabrier u.a. 1988; Lextrait und Zarli 1990] basiert auf ähnlichen Einschränkungen. Auch hier werden automatisch angeordnete, auf Schachtelung basierende visuelle Sprachen implementiert, deren Programme durch Strukturbäume repräsentiert werden können. Die graphische Darstellung wird hier jedoch durch einen inkrementellen Attributauswerter erzeugt. Bei der Implementierung von Gigas wurden Werkzeuge und Spezifikationstechniken aus dem Synthesizer Generator [Reps und Teitelbaum 1989a] benutzt.

Mit Hilfe der attributierten Grammatik wird eine Abbildung der abstrakten Struktur auf ineinander geschachtelte rechteckige Bereiche spezifiziert. Die dabei auftretenden Abhängigkeiten führen zu einer ähnlichen Baumdurchlaufsstrategie, wie sie Grant [1998] für das VPE-System benutzt. Da die Attributberechnungen und ihre Abhängigkeiten für verschiedene Sprachen gemeinsame Eigenschaften aufweisen, wird bei Gigas die attributierte Grammatik aus einer höheren Spezifi-

```
lock EXP: integral hc (
     Symbol (graphic "integral")
          { h = Operands.h - Upper.h ;
          d = Operands.d - Upper.d ;
          W = (h + d) / 2.8 ; }
     Operands v (
          Upper (unlock EXP)
               { s = Symbol.s * 0.8 ; }
          Middle hc (
               Exp (unlock EXP)
               Diff ("d")
               Var (unlock VAR)
               ) { dx = 1.0 ; }
          Lower (unlock EXP)
               { s = Symbol.s * 0.8 ; }
          )
          { h = Upper.d + Upper.h + Middle.h ;
            d = Lower.d + Lower.h + Middle.d ; }
     ) ;
```

(a) Spezifikation



(b) Erzeugte Darstellung

Abbildung 2.19.: Spezifikation visueller Programme durch reckteckige Bereiche in GIGAS nach Franchi-Zannettacci [1989]

kationssprache generiert. Diese Sprache hat den Namen GSL und wird genauer von Franchi-Zannettacci [1989] beschrieben. Mit GSL kann ein Sprachentwerfer spezifizieren, wie sich rechteckige Bereiche aus anderen Bereichen zusammensetzen. Die in einem Bereich enthaltenen Bereiche können dabei horizontal und vertikal angeordnet werden. Aus GSL wird eine Spezifikationsdatei in der Sprache SSL, der Sprache des Synthesizer Generators generiert.

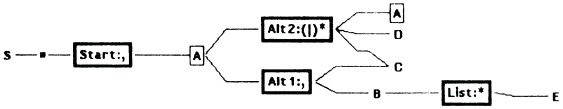
Abbildung 2.19 zeigt einen Ausschnitt aus einer Spezifikation in GSL und die dadurch erzeugte graphische Darstellung für ein Beispielprogramm. Die angegebene Regel ist der Produktion mit Namen “integral” zugeordnet, auf deren linker Seite das Symbol “EXP” steht. Sie spezifiziert einen rechteckigen Bereich, der zwei andere Bereiche “Symbol” und “Operands” enthält. Diese werden auf einer gemeinsamen Grundlinie nebeneinander angeordnet (Layoutdirektive “hc”). “Symbol” referenziert eine vorgefertigte Graphik. In “Operands” werden drei weitere, diesmal untereinander (Layoutdirektive “v”) angeordnete Bereiche eingefügt. In “Upper” und “Lower” werden rechteckige Bereiche eingesetzt, die Symbolen der rechten Seite der Produktion ”integral” zugeordnet sind. “Middle” setzt sich wiederum aus drei Bereichen zusammen. Hier werden zwei reckteckige Bereiche weiterer Grammatiksymbole und ein konstanter Text eingefügt.

In geschweiften Klammern können jeweils Attributberechnungen angegeben werden, die die Defaultberechnungen der Layoutdirektiven überschreiben. Damit kann der Sprachentwerfer das Layout eines Bereichs seinen Vorstellungen anpassen. Im Beispiel ist dies geschehen, indem die Größe des Integralzeichens angepaßt, ein Rand spezifiziert und der rechteckige Bereich “Middle” nach rechts versetzt wurde.

2.3.4.5. LOGGIE

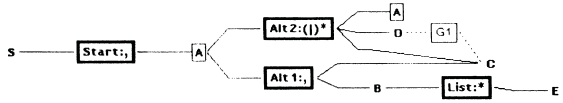
Das LOGGIE-System [Backlund u.a. 1990, 1989] besteht aus zwei Ebenen. Die untere, “derivation layer” genannte Ebene speichert das editierte Programm. Dazu wird ein Strukturbaum verwendet, der wie oben bereits erwähnt um zusätzliche Kanten erweitert ist. Ein Attributauswerter wird eingesetzt, um die obere, “structure-presentation-layer” genannte Ebene zu steuern. Diese implementiert die Benutzungsschnittstelle der Strukturediteren.

Ein wichtiger Aspekt im LOGGIE-System ist die Aufteilung der Aufgaben der Sprachimplementierung zwischen Attributauswerter und Präsentationsebene. Beim LOGGIE-System wird wesentliche Funktionalität in der Präsentationsebene umgesetzt. Hier werden einerseits graphische Objekte für die Knoten des Strukturbaums erzeugt und die verschiedenen Fenster verwaltet, die auch gleichzeitig unterschiedliche graphische Darstellungen für ein Programmobjekt zeigen können. Andererseits werden hier auch die graphischen Objekte eines Fensters angeordnet. Dazu werden einige Layoutalternativen zur Verfügung gestellt, die vom Sprachentwerfer mit Hilfe vordefinierter Constraints weiter spezialisiert werden




```
Start:   S ::= A
Alt1:    A ::= B C
Alt2:    A ::= ( C | D | A )*
List:    B ::= E*
```

(a) Visuelle Notation für eine kontextfreie Grammatik



(b) Notation für "*Garlands*"

Abbildung 2.20.: Spezifikation der abstrakten Struktur einer Sprache mit LOGGIE nach Backlund u.a. [1990, Fig. 1 und 2]

können. Dabei besteht aber keine Möglichkeit, die genauen Positionen und die Darstellungseigenschaften der graphischen Objekte zu spezifizieren – diese sind in der Präsentationsschicht gekapselt.

Um eine visuelle Sprache zu spezifizieren, muß ein Sprachentwerfer bei LOGGIE die abstrakte Struktur der Sprache definieren und eine attributierte Grammatik angeben. Die abstrakte Struktur im LOGGIE-System wird mit einer graphischen Notation für EBNF spezifiziert. Dabei können zusätzliche, "Garlands"[7] genannte Beziehungen zwischen je zwei Symbolen der Sprache spezifiziert werden. Abbildung 2.20 zeigt oben die dazu eingeführte Notation, die an auf der Seite liegende Ableitungsbäume erinnert. Mit den Textzeichen ",", "|", "*" und "+" werden dabei Tupel, Alternativen und Listen spezifiziert, davor kann der Name der Produktion angegeben werden. Die Abbildung enthält zwei visuelle Varianten der unten angegebenen Grammatik, die aus Abb. 2.20(a) enthält zusätzlich die Definition einer Garland-Kante.

Im weiteren muß ein Sprachentwerfer eine attributierte Grammatik eingeben, wobei er zum einen durch Strukturedi toren unterstützt wird, mit denen jeweils eine Attributgleichung eingegeben werden kann. Diese Strukturedi toren werden vom System anhand des Attributtyps ausgewählt. Zum anderen werden einige

[7] deutsch etwa "Kranz" oder "Girlande"

vordefinierte Attribute (samt Attributgleichungen) zur Verfügung gestellt, die ein Sprachentwerfer in seine Spezifikation übernehmen kann.

Für die Auswertung der attributierten Grammatiken werden inkrementelle Attributauswerter benutzt, die nach der Methode von Reps u.a. [1983] erzeugt werden. Diese Methode wurde von Backlund u.a. um entfernte Attributzugriffe entlang von Garland-Kanten erweitert. Die Autoren berichten dabei von Effizienzproblemen, die sie auf den für die Auswertung benötigten Speicherplatz zurückführen.

Eine weitere Besonderheit des LOGGIE-Systems sind Attribute mit programmierten Seiteneffekten. Hiermit werden Attribute bezeichnet, denen Programmcode zugeordnet ist. Immer, wenn die Attribute einen bestimmten Attributwert haben, wird der spezifizierte Code ausgewertet, der seinerseits wiederum den Strukturbaum ändern kann. Auf diese Weise können komplexe Editieroperationen implementiert werden.

2.3.4.6. VIVID

Das VIVID-System [Dangberg und Müller 1999] ist ein System, mit dem graphische Oberflächen für die Visualisierung und Änderung von Datenbankinhalten konstruiert werden können. Dabei können die Datenbankobjekte durch "graphische Symbole" und die Beziehungen durch Anordnung der Symbole oder durch Linien dargestellt werden. Dadurch kann das System auch eingesetzt werden, um visuelle Sprachen zu implementieren. Hierbei wird die abstrakte Struktur der Sprache durch eine Datenbankdefinition spezifiziert.

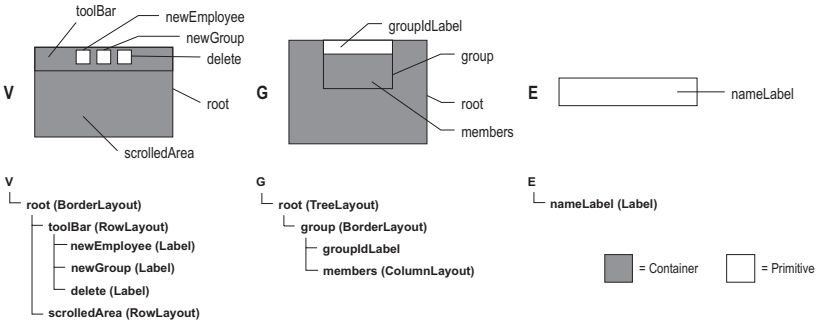
Die erzeugte graphische Darstellung für eine Datenbank ist bei VIVID hierarchisch aufgebaut. Weiterhin beschränkt die Spezifikation der Interaktion mit der Darstellung die verwendbaren Layoutmethoden. Dadurch ist die Menge der mit VIVID implementierbaren visuellen Sprachen beschränkt.

Die Spezifikation einer graphischen Datenbankoberfläche besteht in VIVID aus drei Teilen. Im ersten Teil wird die hierarchische Struktur der Benutzungsschnittstelle für die zu implementierende Sprache spezifiziert, der zweite Teil beschreibt die Visualisierung der Inhalte einer Datenbank und der dritte Teil die Benutzerinteraktionen.

Die hierarchische Struktur der Benutzungsschnittstelle wird in einzelnen Teilen, den "graphischen Symbolen" spezifiziert, die sich wiederum aus "Containern" und vordefinierten Anzeigeelementen zusammensetzen. In die Container werden im ablaufenden Editor (i.a. mehrere) graphische Symbole eingefügt. Die Anordnung der Symbole eines Containers wird durch Layoutmanager spezifiziert, die den Containern zugeordnet werden können. Dabei kann der Sprachentwerfer auf eine Reihe vordefinierter Layoutmanager zurückgreifen. Abbildung 2.21(a) zeigt die Definition der graphischen Symbole "V", "G" und "E" für eine Beispielanwendung.

Um die graphische Darstellung für die Inhalte der Datenbank zu erzeugen, werden graphische Symbole für Datenbankobjekte instanziiert und zu einem Baum zusammengesetzt. Dieser Vorgang wird durch drei Tabellen gesteuert. Die erste Tabelle enthält SQL-Abfragen und Namen graphischer Symbole, siehe Abb 2.21(b). Für jede Ergebniszeile einer Abfrage instanziiert VIVID das angegebene graphische Symbol. Die Attribute der Benutzungsschnittstellenelemente werden dabei aus der Datenbank mit Hilfe der Angaben aus der zweiten Tabelle (Abb. 2.21(c) initialisiert. In einer dritten Tabelle (Abb. 2.21(d) wird die Komposition des Baumes aus den instanziierten graphischen Symbolen beschrieben. Hierzu spezifiziert der Sprachentwerfer jeweils ein graphisches Symbol, einen Container darin, ein anderes Symbol und eine Bedingung. Für alle graphischen Symbole, die noch in keinem Container eingefügt sind, werden der Reihe nach die angegebenen Bedingungen geprüft. Ist eine Bedingung erfüllt, so wird das graphische Symbol in den spezifizierten Container eingefügt.

Mit Hilfe der ersten beiden Teile der Spezifikation kann das VIVID-System für eine bestimmte Datenbank eine visuelle Darstellung erzeugen. Im dritten Teil wird die Interaktion mit der erzeugten visuellen Darstellung beschrieben, die auf Drag&Drop-Interaktion beruht. Der Sprachentwerfer kann dabei jedem Paar von Drag- und Drop-Objekten, ausgehend von den graphischen Symbolen, eine SQL-Anweisung zuordnen. Zieht der Benutzer etwa einen Datensatz auf das Papierkorb-Symbol, so wird der Datensatz gelöscht, weil der Sprachentwerfer eine DELETE-Anweisung angegeben hat.



(a) Zusammensetzen der graphischen Symbole

Property	SQL Statement
$E \rightarrow sourceDef \rightarrow queryString$	$SELECT * FROM\ Employee$
$G \rightarrow sourceDef \rightarrow queryString$	$SELECT * FROM\ Group$

(b) Assoziation graphischer Symbole mit Datenbankinhalten

Property	Attibute Expression
$E \rightarrow nameLabel \rightarrow text$	$source \rightarrow Name$
$G \rightarrow root \rightarrow group \rightarrow groupIdLabel \rightarrow text$	$source \rightarrow GroupId$

(c) Assoziation von Eigenschaften der Symbole mit Attributen der Datenbank

Parent	Child	Condition	Symbol Container
V	G	$G \rightarrow source \rightarrow SuperGroup == NULL$	$V \rightarrow root \rightarrow scrolledArea$
G	G	$G[1] \rightarrow source \rightarrow GroupId == G[2] \rightarrow source \rightarrow SuperGroup$	$G[1] \rightarrow root$
G	E	$G \rightarrow source \rightarrow GroupId == E \rightarrow source \rightarrow MemberOf$	$G \rightarrow root \rightarrow group \rightarrow members$

(d) Komposition der graphischen Symbole

Abbildung 2.21.: Spezifikation von Datenbankoberflächen im VIVID-System nach Dangberg und Müller [1999]

3. Muster in visuellen Sprachen

In dieser Arbeit stelle ich eine Methode vor, mit der Entwicklungsumgebungen für visuelle Sprachen generiert werden können. Kern dieser Methode ist der Einsatz von Spezifikationsmodulen, die eine hohe Abstraktionsebene für die Sprachimplementierung erzielen. Dies ermöglicht es einerseits auch Nicht-Spezialisten, visuelle Sprachen zu implementieren, da erprobte Darstellungstechniken, zusammen mit spezialisierten Interaktions- und Layoutmechanismen gekapselt, den Sprachentwickler vor komplexen Details bewahren. Andererseits können so auch leicht Implementierungsalternativen für neue visuelle Sprachen ausprobiert werden, was den Sprachentwurf erleichtert. Mit der vorgestellten Methode können schließlich auch Werkzeuge erzeugt werden, mit denen visuelle Programme analysiert und übersetzt werden können. Hierzu werden Generatoren wiederverwendet, die zur Lösung dieser Aufgaben erfolgreich bei textuellen Sprachen eingesetzt werden.

Bevor ich im zweiten Abschnitt dieses Kapitels näher auf die Konzeption des Generators eingehe und damit die Methode der Sprachimplementierung erläutere, stellt der erste Abschnitt die Idee vor, die der Abstraktion der Sprachimplementierung zugrundeliegt. Diese Idee der Sprachimplementierung durch Bezug auf 'visuelle Standardtechniken' wird im dritten Abschnitt wieder aufgegriffen, um die Implementierung visueller Sprachen durch wiederverwendbare Spezifikationsmodule zu modularisieren. Visuelle Standardtechniken werden hier zu Entwurfsmustern für visuelle Sprachen oder kurz zu *visuellen Mustern* konkretisiert. Der vierte Abschnitt faßt die Arbeiten zusammen, die im Rahmen einer Diplomarbeit durchgeführt wurden, um die visuellen Muster zu erarbeiten. Hier wird auch ein Überblick über die identifizierten visuellen Muster gegeben, der als initialer Satz von Mustern zum Entwurf visueller Sprachen betrachtet wird.

3.1. Die Idee

In einer visuellen Sprache kann der Informationsgehalt einer Darstellung oft auf unterschiedliche Art und Weise präsentiert werden. Alternativen unterscheiden sich in ihrer Übersichtlichkeit für kleine und große Darstellungen, benötigen unterschiedlich viel Platz auf der Bildschirmfläche, unterstützen andersartige Lay-

Die Folge `Statements` soll als **Liste** dargestellt werden, wobei Trennungslinien und eine vertikale Ausbreitungsrichtung zu verwenden sind.

Die Folge `Statements` soll als **Stapel** dargestellt werden, wobei die Knöpfe zur Navigation links angeordnet werden sollen.

Abbildung 3.1.: Informelle Beschreibung der Darstellung einer Anweisungsfolge

outtechniken und erfordern verschieden geartete Interaktionen des Benutzers, um die Darstellung zu konstruieren oder zu ändern. Weitere Unterschiede ergeben sich bezüglich des Grades der Visualisierung und der Möglichkeit zur sekundären Strukturierung der Sprachkonstrukte, durch die einem Betrachter eines visuellen Programms wichtige Hinweise zum Verständnis gegeben werden können [Green und Petre 1996]. Schließlich können verschiedenartige Darstellungen unterschiedliche Informationen hervorheben und sich deshalb für die Lösung bestimmter Aufgaben besser eignen als andere Darstellungen [Whitley 1997].

Eine Folge von Anweisungen kann beispielsweise durch eine Liste dargestellt werden, in der die Anweisungen untereinander angeordnet sind. Hier sieht man alle Anweisungen gleichzeitig und kann neue zwischen den anderen einfügen. Solange die dafür benötigte Zeichenfläche klein ist, ist die resultierende graphische Darstellung übersichtlich. Wird die benötigte Fläche für eine Anweisung jedoch größer, so wächst auch der Platzverbrauch der Folge. Die zur Verfügung stehende Bildschirmfläche reicht zu ihrer Anzeige möglicherweise nicht mehr aus. Ein Benutzer verliert dann schnell die Übersicht, vor allem, wenn die Größe der Anweisungsfolge bewirkt, daß andere wichtige Bildelemente weit über die Anzeigefläche verstreut sind.

Alternativ können die Anweisungen der Folge hintereinander gestapelt dargestellt werden. Hier sieht man immer nur eine Anweisung der Folge auf einmal und reduziert dadurch den Platzbedarf für die Folge. Das kann bewirken, daß sich andere Details besser überschauen lassen und die vollständige Darstellung übersichtlicher wird.

Welche dieser beiden Alternativen für eine Sprache vorzuziehen ist, kann nicht allgemein entschieden werden, sondern muß von Fall zu Fall neu beurteilt werden. Beide Möglichkeiten sind Standardtechniken zur visuellen Darstellung von Folgen.

Die Idee zu dieser Arbeit ist, daß Techniken wie die beschriebenen verwendet werden können, um visuelle Sprachen zu implementieren. Ein Sprachentwerfer könnte etwa eine der in Abbildung 3.1 angegebenen informellen Beschreibungen verwenden, um das Aussehen und auch mögliche Editieroperationen für eine Anweisungsfolge festzulegen.

Zusammen mit ähnlichen Beschreibungen für andere Aspekte der Sprache

könnte so ein Strukturedítor für eine visuelle Sprache beschrieben werden. Die Vorteile, die sich dadurch ergeben sind eine hohe Spezifikationsebene, Einfachheit, Flexibilität und Allgemeinheit:

Hohe Spezifikationsebene Die Standardtechniken assoziieren Aussehen und Verhalten von Teilen einer visuellen Sprache mit einem aussagekräftigen Namen. Für die Sprachimplementierung werden diese Namen mit bestimmten Konstrukten einer visuellen Sprache assoziiert und spezifische Details angegeben. Durch Komposition derartiger Spezifikationen kann eine visuelle Sprache vollständig beschrieben werden.

Einfachheit Das Know-How einer Sprachimplementierung wird durch den Bezug auf die Standardtechniken gekapselt und in einfacher Weise anwendbar gemacht. So können visuelle Sprachen auch durch Nicht-Spezialisten implementiert werden.

Flexibilität Die Implementierung eines Aspektes einer Sprache wird durch Bezug auf eine Standardtechnik beschrieben. Weitreichende Sprachänderungen können realisiert werden, indem für einen Aspekt eine andere Technik verwendet wird, beispielsweise einen Stapel statt einer Liste. So lassen sich Sprachänderungen und Entwurfsexperimente leicht durchführen.

Allgemeinheit Ein breites Spektrum verschiedener Standardtechniken ist anwendbar, um visuelle Sprachen zu implementieren. Beim Bezug auf eine visuelle Technik können die Details der Darstellung, wie z.B. die Ausbreitungsrichtung bei Listen festgelegt werden. Bei Bedarf können besondere oder einzigartige Konstrukte einer visuellen Sprache aber auch auf einer niedrigeren Ebene realisiert werden.

Um diese Ziele umzusetzen, kann das Konzept der Spezifikationsmodule wiederverwendet werden, das im Bereich textueller Sprachen bereits erfolgreich eingesetzt wird [Kastens und Waite 1994, "Attribution Modules"]. Hier kapseln Spezifikationsmodule Standardlösungen für Aufgaben, die bei der semantischen Analyse textueller Sprachen anfallen. Sie werden angewendet, indem sie instanziiert und mit den verschiedenen Konstrukten einer Sprache assoziiert werden. Es gibt beispielsweise Spezifikationsmodule, die die Namens- und Typanalyse für eine Sprache kapseln. Ein Sprachentwerfer kann damit Spezifikationen schreiben, die informell etwa so beschrieben werden könnten, wie dies Abbildung 3.2 skizziert.

Diese Spezifikationsmodule sind mit Hilfe von Abstraktionstechniken für attributierte Grammatiken umgesetzt worden. Mit attributierten Grammatiken können Berechnungen in Bäumen spezifiziert werden. Damit können sehr allgemein Lösungen für Analyse-, Transformations- und Syntheseaufgaben für hierarchisch

strukturierte Informationen formuliert werden. Die Spezifikationsmodule basieren also auf einer Spezifikationstechnik, die potentiell ein weites Aufgabengebiet abdecken kann.

Die durch den hier verwendeten Generator Liga [Kastens und Waite 1994] umgesetzten Abstraktionskonzepte machen die Spezifikation der Baumberechnungen zum einen unabhängig von einem bestimmten Baum und zum anderen unabhängig von anderen Baumberechnungen formulierbar. Diese Abstraktionskonzepte bilden die Grundlage für die Realisierung der Spezifikationsmodule, die in Form von Attributierungsmodulen abgelegt und später mit bestimmten Kontexten einer Sprache assoziiert werden können.

Auch für die Implementierung visueller Sprachen sind attributierte Grammatiken bereits erfolgreich verwendet worden [Backlund u.a. 1990; Franchi-Zannettacci 1989]. Die Idee, den Generator Liga zur Implementierung visueller Sprachen zu verwenden und mit den unterstützten Abstraktionstechniken Spezifikationsmodule für visuelle Sprachen bereitzustellen, liegt also nah. Diese Idee wird im folgenden Abschnitt zu einer Methode konkretisiert, mit der visuelle Sprachen implementiert werden können. Abschnitt 3.3 und 3.4 sind der systematischen Suche nach visuellen Standardtechniken gewidmet, die –durch Spezifikationsmodule umgesetzt– die Beschreibungsmittel liefern, mit denen Editoren für visuelle Sprachen spezifiziert werden können.

3.2. Die Methode

Visuelle Sprachen werden in dieser Arbeit durch Entwicklungsumgebungen implementiert, deren zentrale Komponente ein Strukturedtor ist. Strukturedtoren editieren nicht die Darstellung, sondern die Struktur eines visuellen Programms. Für einen Anwender des Editors bedeutet dies, daß strukturelle Editieroperationen in einfacher Weise realisiert werden können. Das äußert sich beispielsweise dadurch, daß durch den Editor Platz für ein eingefügtes Sprachkonstrukt geschaffen wird, wozu u.U. umfangreiche Änderungen in der graphischen Darstellung erforderlich sind.

Führe die Namensanalyse mit den Regeln für Algol-Gültigkeitsbereiche durch, wobei durch `Block` Gültigkeitsbereiche und durch `DefIdent` definierte Bezeichner in dieser Sprache dargestellt werden.

Die Programmobjekte, die durch `Decl` repräsentiert werden, führen typisierte Objekte ein, deren Typ durch `TypeNotation` festgelegt wird.

Abbildung 3.2.: Informell formulierte Anwendungen existierender Spezifikationsmodule für textuelle Sprachen

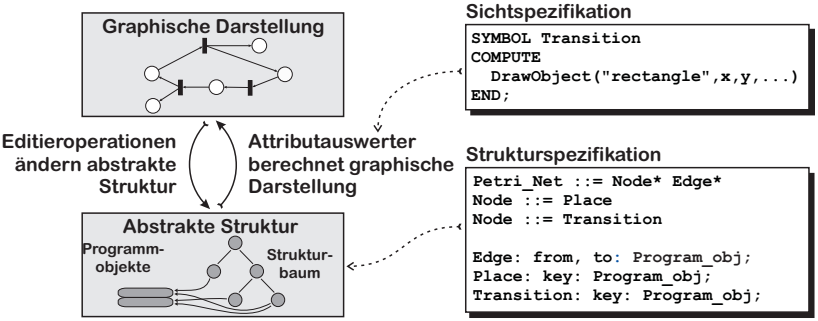


Abbildung 3.3.: Der Editierzyklus und seine Spezifikation

Neben anderen wichtigen Eigenschaften haben Strukturediteren für visuelle Sprachen den Vorteil, daß sie sich relativ leicht realisieren lassen. Ihr Aufbau ist geeignet, um ihre wesentlichen Bestandteile durch Werkzeuge textueller Sprachen, insbesondere durch Generatoren für attributierte Grammatiken zu erzeugen. Wie in den meisten Editoren für visuelle Sprachen lassen sich auch in den hier erzeugten Strukturediteren zwei Repräsentationen der visuellen Programme unterscheiden, die graphische Darstellung eines visuellen Programms und dessen abstrakte Struktur.

In dem hier vorgestellten Ansatz wird lediglich die abstrakte Struktur eines visuellen Programms explizit gespeichert, die graphische Darstellung implizit aus der abstrakten Struktur bei Bedarf neu erzeugt.[1] Ein Benutzer kann mit der graphischen Darstellung interagieren. Er kann etwa einen Teil der Darstellung auswählen und löschen, ein Konstrukt in der Darstellung verschieben oder ein neues Konstrukt an einer bestimmten Stelle der Darstellung neu einfügen. Diese Interaktionen lassen sich realisieren, indem dazu passende Änderungen in der abstrakten Struktur durchgeführt werden. Welche Interaktionen möglich sind und wie eine Interaktion durch Änderung der abstrakten Struktur realisiert werden kann, wird durch zusätzliche Informationen bestimmt, die bei der Erzeugung der Darstellung berechnet und in die Darstellung integriert werden. Nach der Durchführung einer Änderung wird durch eine erneute Berechnung der graphischen Darstellung der nächste Editierzyklus eingeleitet.

Um einen Strukturediteur zu implementieren, sind also im wesentlichen zwei Aufgaben zu lösen: die Speicherung der abstrakten Struktur eines visuellen Programms, sowie die Berechnung seiner graphischen Darstellung und der durchführbaren Editieroperationen. Abbildung 3.3 veranschaulicht den Editierzyklus

[1] Einige Angaben zum Layout der Darstellung werden, soweit es manuell durch den Benutzer festgelegte Werte betrifft, als zur abstrakten Struktur zugehörig betrachtet.

eines Strukturediteors und zeigt, wie die Lösung dieser Aufgaben durch Spezifikationen beschrieben werden kann. Die abstrakte Struktur wird hierzu durch einen Strukturbaum implementiert, dessen Knoten um Attribute ergänzt werden.

Dieser Strukturbaum kann durch eine kontextfreie Grammatik spezifiziert werden. Das ist eine Voraussetzung, um attributierte Grammatiken anzuwenden. Mit attributierten Grammatiken können voneinander abhängige Berechnungen für Bäume spezifiziert werden. Die Attributberechnung, die in Abbildung 3.3 oben rechts skizziert ist, erzeugt für jeden `Transition`-Knoten der abstrakten Struktur ein schwarzes Rechteck in der graphischen Darstellung. Ein Generator erzeugt daraus ein Programm, das für jeden Strukturbaum die spezifizierten Berechnungen in der richtigen Reihenfolge anstößt. Mit Hilfe attributierter Grammatiken werden also Berechnungen formuliert, die aus dem Strukturbaum die graphische Darstellung eines visuellen Programms erzeugen und die Editieroperationen festlegen.

Neben diesen Aufgaben werden diverse weitere Spezifikationen benötigt, um einerseits den generierten Strukturediteor zu vervollständigen und andererseits Aufgaben durchzuführen, die in einer Entwicklungsumgebung außer dem Editieren von Programmen noch gelöst werden müssen. Beispiele dafür sind die Beschreibung sprachabhängiger Teile der Benutzungsschnittstelle, der Ex- und Import von Programmen mit speziellen Dateiformaten sowie die Analyse und Weiterverarbeitung erstellter Programme. Die letzten beiden dieser Aufgaben betreffen ebenfalls das Problemgebiet übersetzererzeugender Systeme und können unter Verwendung von Attributauswertern und weiterer hilfreicher Werkzeuge gelöst werden, da die abstrakte Struktur durch einen Strukturbaum repräsentiert wird.

Abbildung 3.4 zeigt die aufeinander aufbauenden Spezifikationsebenen des Werkzeugsystems VLEli, das Entwicklungssysteme für visuelle Sprachen generiert. Die unterste Ebene baut auf drei anderen Werkzeugsystemen auf. Aus Eli [Kastens u.a. 1998; Gray u.a. 1992] werden der Generator für Attributauswerter und andere kooperierende Werkzeuge verwendet, um visuelle Programme zu speichern, weiterzuverarbeiten und den zentralen Editierzyklus des Strukturediteors zu implementieren. Die Benutzungsschnittstelle und die Ablaufsteuerung der erzeugten Sprachimplementierung sind durch ein Rahmenwerk implementiert, das mit Hilfe des Werkzeugs Tcl/Tk [Welch 1997] umgesetzt wurde.

Schließlich wird noch ein Constraint-Solver benötigt, um die visuelle Darstellung eines Programms zu erzeugen. Einige Programmkonstrukte visueller Sprachen besitzen Layoutfreiheiten, die gewisse Konsistenzbedingungen erfüllen müssen. Die Abhängigkeiten zwischen den Konstrukten können gut durch Einschränkungen (engl. Constraints) beschrieben werden, die durch die abstrakte Struktur der Sprache erfüllt sein müssen.[2] Wird durch eine Editieroperation eine Einschrän-

[2] Genauer: Die Layoutangaben der abstrakten Struktur müssen diese Constraints erfüllen.

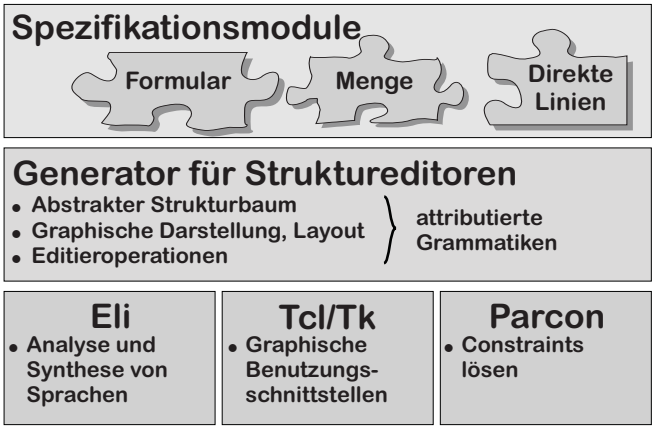


Abbildung 3.4.: Die drei Spezifikationsebenen in VLEli

kung verletzt, so kann oft durch weitere Änderungen der abstrakten Struktur wieder ein gültiger Zustand hergestellt werden. Der Constraint-Solver berechnet diese zusätzlichen Änderungen aus einer Eingabe, die aus der abstrakten Struktur des Programms von einer attributierte Grammatik erzeugt wird.

Die mittlere Ebene dieses Ansatzes integriert die Werkzeuge der unteren Ebene zu einem System, mit dem aus Spezifikationen Entwicklungsumgebungen für visuelle Sprachen generiert werden können. Obwohl die Ausdruckskraft dieser Schicht letztlich für das breite Einsatzgebiet dieses Ansatzes verantwortlich ist, ist sie nicht in erster Linie zur Verwendung durch den Sprachentwerfer gedacht – dazu ist sie zu technisch. Statt dessen kann die Spezifikation visueller Sprachen weiter abstrahiert werden, indem die Abstraktionstechniken des Generators `Liga` ausgenutzt werden.

Die Abstraktionstechniken können verwendet werden, um visuelle Standardtechniken durch Spezifikationsmodule umzusetzen. Diese können beispielsweise eingesetzt werden, um die Berechnungen zu kapseln, die die graphische Darstellung erzeugen und die Editieroperationen bestimmen. Spezifikationsmodule, die diese Aufgabe lösen, bilden die oberste Ebene im System VLEli.

Ein wesentliches Problem, das dabei zu lösen ist, ist eine geeignete Festlegung für die Aufgaben, die durch die Spezifikationsmodule zu lösen sind. Abschnitt 3.3 geht näher auf die Anforderungen ein, die an die Aufteilung gestellt werden und beschreibt den hier gegangenen Lösungsweg.

3.2.1. Anwendbarkeit

Die vorgestellte Methode der Implementierung visueller Sprachen ist für eine große Klasse visueller Sprachen geeignet. Diese Sprachen sind charakterisiert durch eine zugrundeliegende abstrakte Struktur, die durch graphische Darstellungen visualisiert und bearbeitet werden sollen. Das schließt viele gängige visuelle Sprachen und Notationen ein. Nach der Klassifikation von Shu [Shu 1988, S. 12] wird diese Sprachklasse am ehesten durch die Kategorien "Programmierung mit visuellen Ausdrücken basierend auf Diagrammen" und "(...) Formularen" ausgedrückt.

Die Verwendung von Piktogrammen (engl. Icons) wird nur insoweit unterstützt, als daß diese Bestandteil von Diagrammen sein können. Mit Piktogrammen können in Diagrammen aussagekräftigere Darstellungen erzeugt werden, als es mit einfachen geometrischen Figuren möglich ist, siehe z.B. die Darstellung paralleler Programme durch Streets in Abschnitt 2.1.2.4

Piktogrammsprachen (oder Piktogrammsätze) bezeichnen dagegen "visuelle Ausdrücke mit Piktogrammen, bei denen die Position eine wesentliche Rolle spielen" [Myers 1994, Schiffer 1998, S. 126]. Piktogrammsprachen und Sprachen, die auf der Blox-Methodik [Glinert 1987] beruhen, sind Sprachen, die sich nicht sinnvoll mit Strukturediteren implementieren lassen. Mit der hier beschriebenen Methode können zwar einfache Editoren für derartige Sprachen erzeugt werden. Die dazu benötigte abstrakte Struktur würde jedoch nicht die Struktur eingegebener Programme widerspiegeln, so daß Analyse und Weiterverarbeitung sich nur schwer durchführen lassen.

Visuelle Sprachen, die eine andersartige Benutzerschnittstelle benötigen, lassen sich ebenfalls nicht mit diesem Ansatz implementieren. Beispiele dafür sind Sprachen, die auf der Technik der "Programmierung durch Beispiel" beruhen sowie gemischt textuelle und visuelle Sprachen, wie sie Erwig und Meyer [1995] implementieren. Der zuletzt genannte Ansatz unterstützt graphische Darstellungen, die in ein textuelles Programm einer beliebigen Wirtssprache eingebettet sind.

Schließlich entstehen auch aus der Verwendung des Systems Tcl / Tk Einschränkungen für die umsetzbaren visuellen Sprachen. Obwohl das System gut erweiterbar ist, können zunächst nur die Darstellungsmöglichkeiten verwendet werden, die das Basossystem zur Verfügung stellt. Das schließt etwa dreidimensionale visuelle Sprachen, z.B. SAM [Müller u.a. 1998] zunächst aus.

3.3. Visuelle Muster

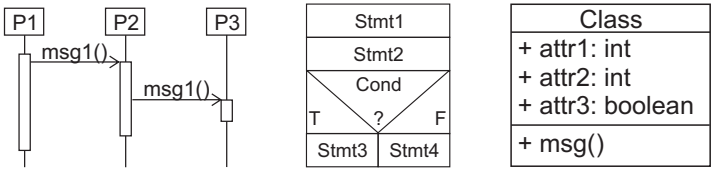
Im vorigen Abschnitt wurde eine Methode vorgestellt, bei der attributierte Grammatiken einsetzt werden, um Strukturediteren für visuelle Sprachen zu erzeugen. Der Generator Liga unterstützt dabei eine Abstraktionstechnik, die die Kapselung

und Wiederverwendung von Attributberechnungen in Modulen, sog. *Spezifikationsmodulen* gestattet. Diese Technik wurde im Bereich der Analyse textueller Sprachen bereits erfolgreich eingesetzt, um die Namens- und Typanalyse zu modularisieren [Kastens und Waite 1994].

In dieser Arbeit zeige ich, daß sich diese Abstraktionstechniken auch anwenden lassen, um die Implementierung von Strukturediteren für visuelle Sprachen zu erleichtern. Für die Entwicklung der Spezifikationsmodule ist die Frage, wie die Aufgabe der Editorimplementierung in Teilaufgaben unterteilt werden kann, von zentraler Bedeutung. Die Aufteilung sollte dabei so erfolgen, daß die Teilaufgaben in möglichst vielen Sprachen auftreten und von Sprachentwicklern möglichst gut identifiziert werden können. Die Teillösungen für diese Aufgaben sollen sich für möglichst viele Sprachen wiederverwenden und flexibel miteinander kombinieren lassen. Ihr Zusammensetzen soll möglichst wenige zusätzliche Angaben an den Schnittstellen benötigen. Außerdem sollen sich dadurch visuelle Sprachen möglichst vollständig implementieren lassen.

Die Idee, die der Konzeption der Spezifikationsmodule zugrundeliegt, habe ich schon in Abschnitt 3.1 vorgestellt: die abstrakte Struktur eines visuellen Sprachkonstrukts wird vielfach durch gewisse, eingeführte Techniken visualisiert, die zur Definition der Spezifikationsmodule verwendet werden können. Als Beispiel dafür sind in Abbildung 3.5 Konstrukte aus drei visuellen Sprachen abgebildet. Es handelt sich um ein UML-Ablaufdiagramm (Abb. 3.5(a)), ein Nassi-Shneiderman-Diagramm (Abb. 3.5(b)) und um eine Klasse aus einem UML-Klassendiagramm (Abb. 3.5(c)). Die Prozesse (*ClassifierRoles*) des Ablaufdiagramms, die Anweisungen des Nassi-Shneiderman-Diagramms sowie die Attribute des Klassendiagramms haben alle die Struktur einer Folge. Die zu den Strukturen gehörenden Darstellungen sind auf einer geometrischen Achse angeordnet, einmal nebenund ein anderes Mal untereinander. Weiterhin werden jeweils gleichartige Editieroperationen benötigt, beispielsweise um Folgenelemente einzufügen oder zu verschieben.

Techniken, wie die Visualisierung von Folgen durch Anordnung auf einer geo-



(a) UML-Ablaufdiagramm (b) NSD (c) UML-Klassendiagr.

Abbildung 3.5.: Graphische Darstellungen von Folgen

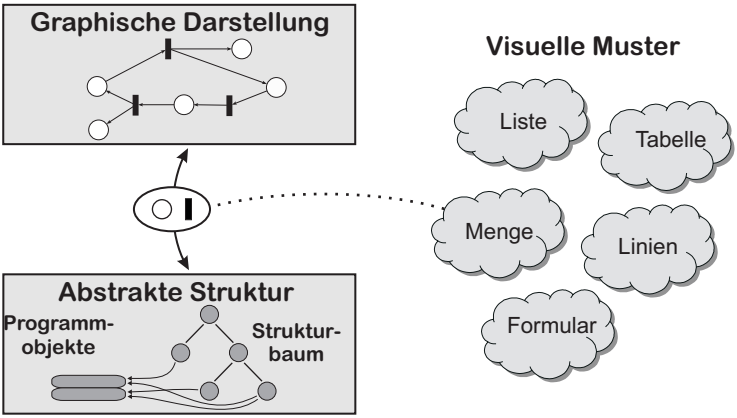


Abbildung 3.6.: Beschreibung wesentlicher Eigenschaften visueller Sprachen durch visuelle Muster (vgl. [Schmidt und Schindler 2000, Abb. 8]

metrischen Achse, werden in dieser Arbeit durch Entwurfsmuster für visuelle Sprachen oder kurz *visuelle Muster* abstrahiert. Wie in Abbildung 3.6 skizziert, beschreibt ein visuelles Muster die abstrakte Struktur und die graphische Darstellung für ein Sprachkonstrukt und definiert Editieroperationen, die zur Interaktion mit dem Sprachkonstrukt verwendet werden sollten. Ein visuelles Muster kapselt somit wesentliche Entwurfseigenschaften eines visuellen Sprachkonstrukts. Der Name "Muster" ist hierbei gerechtfertigt, weil ein visuelles Muster lediglich wesentliche Eigenschaften der Darstellung, der abstrakten Struktur und der Editieroperationen definiert. Die Details bleiben offen und können bei der jeweiligen Anwendung in unterschiedlicher Weise festgelegt werden. So lassen sich die drei Sprachkonstrukte aus Abbildung 3.5 durch ein einziges visuelles Muster beschreiben. Dieses Muster erhält in Abschnitt 3.4.3 den Namen *Listen-Muster*.

Visuelle Muster kapseln gewisse Entwurfsaspekte einer visuellen Sprache und sind mit den Entwurfsmustern von Gamma u.a. [1996] für objektorientierte Programme vergleichbar. Wie die Entwurfsmuster können visuelle Muster durch einen prägnanten Namen, eine Problemstellung, eine Lösung für das Problem und durch Konsequenzen der Anwendung beschrieben werden.

Der *Name* eines visuellen Musters spielt dieselbe Rolle, wie bei den Entwurfsmustern. Er ist ein "Stichwort", mit dem ein bestimmtes visuelles Muster assoziiert werden kann. Er "erweitert das Entwurfsvokabular" für visuelle Sprachen und ist hilfreich, um deren Konzepte zu beschreiben oder zu dokumentieren.

Das zu lösende *Problem* besteht darin, eine möglichst prägnante graphische

Darstellung für eine gegebene abstrakte Struktur der Informationen zu finden. Dabei sollten sich die Beziehungen der abstrakten Struktur möglichst gut auf die graphische Darstellung übertragen. Ein wichtiger Aspekt ist dabei, daß es im allgemeinen mehrere Darstellungsalternativen für eine abstrakte Struktur gibt. Dem wird durch mehrere visuelle Muster, die alternativ beim Sprachentwurf für eine abstrakte Struktur benutzt werden können, Rechnung getragen.

Die *Lösung* für das Problem beschreibt bestimmte Charakteristika einer Sorte von graphischen Darstellungen für die gegebene abstrakte Struktur und legt gewisse grundlegende Editieroperationen fest. Diese Lösung kann noch ausgestaltet und vervollständigt werden. Dies ermöglicht oft eine große Bandbreite verschiedener Darstellungen. Zusammen mit der Möglichkeit, ein anderes Muster für eine bestimmte abstrakte Struktur einzusetzen, eröffnen sich breit gefächerte Realisierungsalternativen.

Die Lösung impliziert *Konsequenzen*, die ihre Anwendung für eine visuelle Sprache hat. Diese Konsequenzen drücken sich beispielsweise in Form von Ausprägungen bestimmter Eigenschaften der visuellen Sprache aus. Green und Petre [1996] haben verschiedene Eigenschaften (visueller) Sprachen und ihrer Implementierung katalogisiert, siehe Abschnitt 2.1.3.1 (Seite 23). Das entstandene Framework kann verwendet werden, um die Konsequenzen der Anwendung eines visuellen Musters zu dokumentieren oder um verschiedene visuelle Muster miteinander zu vergleichen. Ein Beispiel für einen Aspekt eines derartigen Vergleichs findet sich bei der Motivation in Abschnitt 3.1. Weitere Erkenntnisse, aus denen Konsequenzen der Musteranwendung ableitbar sind, faßt Whitley [1997] zusammen.

Für den Entwurf der Spezifikationsmodulbibliothek definieren die visuellen Muster die Teilaufgaben, zu deren Lösung Spezifikationsmodule implementiert werden können. In einem Modul sind somit Attributberechnungen gekapselt, die aus der abstrakten Struktur eines Programmkonstrukts die graphische Darstellung für das Konstrukt berechnen und die Editieroperationen festlegen. Ein visuelles Muster kann dabei meist in unterschiedlicher Weise implementiert werden, so daß i.a. mehrere Spezifikationsmodule für ein visuelles Muster existieren. Da den visuellen Mustern die sie implementierenden Spezifikationsmodule zugeordnet werden können, ist es für den Sprachentwerfer aber leicht, die Spezifikationsmodule zu identifizieren, die benutzt werden können, um eine mit visuellen Mustern entworfene Sprache zu implementieren.

Im weiteren wurden die visuellen Muster durch eine Analyse bedeutender visueller Sprachen erarbeitet. Durch die Auswahl der Sprachen und die Vorgehensweise bei der Identifikation der Muster sollte sichergestellt werden, daß die visuellen Muster bedeutende, oft verwendete Techniken visueller Sprachen beschreiben. Für die durch die Muster konzipierte Spezifikationsmodulbibliothek bedeutet das,

daß die Module zur Implementierung vieler visueller Sprachen angewendet werden können.

Die anderen Anforderungen an die Konzeption der Spezifikationsmodule sind nicht automatisch durch die Vorgehensweise erfüllt. Auf die Kombinierbarkeit der Spezifikationsmodule, auf die Austauschbarkeit von Modulen zu ähnlichen abstrakten Strukturen und auf möglichst gut zusammenpassende Schnittstellen muß bei deren Entwicklung geachtet werden. Dies wird aber dadurch begünstigt, daß sich auch die visuellen Muster flexibel miteinander kominieren lassen und gegeneinander ausgetauscht werden können, wenn sie ähnliche abstrakte Strukturen visualisieren. Aufbauend auf den Mustern, deren Erarbeitung im folgenden Abschnitt beschrieben wird, geht Kapitel 5 näher auf die Entwicklung der Spezifikationsmodule ein.

3.4. Suche nach visuellen Mustern

Im Rahmen ihrer Diplomarbeit haben Schmidt und Schindler [2000] einige visuelle Sprachen untersucht. Das Ziel war es dabei, Techniken zu identifizieren, die zur Visualisierung der zugrundeliegenden Informationen in visuellen Sprachen häufig und erfolgreich verwendet werden. Aufgeschlüsselt nach der abstrakten Struktur der zugrundeliegenden Daten liefern diese Techniken eine Grundlage zur Definition eines initialen Satzes visueller Muster.

Im ersten Unterabschnitt nenne ich die untersuchten Sprachen und beschreibe die Kriterien, die zur Auswahl dieser Sprachen geführt haben. Im Anschluß daran wird exemplarisch die Sprache der Zustandsdiagramme untersucht. Dabei werden Kandidaten für visuelle Muster identifiziert und Alternativen aufgezeigt. Abschnitt 3.4.3 gibt schließlich einen Überblick über die erarbeiteten Muster.

3.4.1. Auswahl untersuchter Sprachen

Zur Suche nach visuellen Mustern wurden in der Diplomarbeit acht visuelle Sprachen ausgewählt. Die Kriterien, nach denen die Sprachen ausgewählt wurden, sind die Bekanntheit und Verbreitung der Sprachen, die Verwendung möglichst unterschiedlicher Strukturen in der graphischen Darstellung und die Verwendung möglichst unterschiedlicher Darstellungsvarianten.

Durch die *Bekanntheit* und die *Verbreitung* einer Sprache sollte sichergestellt werden, daß die in ihr verwendeten Muster bedeutend für praktische Anwendungen sind. Indem Sprachen mit *unterschiedlichen Strukturen* der graphischen Darstellung ausgewählt werden, sollte erreicht werden, daß eine breite Palette unterschiedlicher visueller Muster identifiziert werden können. Es war das Ziel, möglichst mehrere alternative visuelle Muster für abstrakte Sprachstrukturen zu erarbeiten, damit später Entwurfsalternativen für visuelle Sprachen zur Verfügung

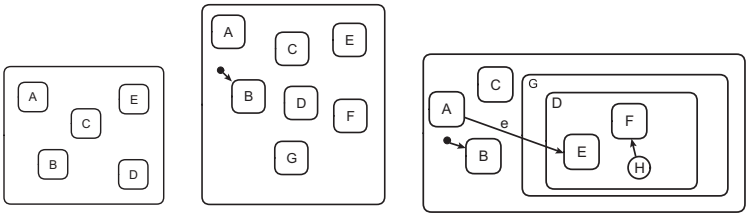
stehen. Schließlich wird durch *unterschiedliche Darstellungsvarianten* ein breites Einsatzgebiet einzelner visueller Muster erzielt.

Unter den ausgewählten Sprachen befinden sich drei bedeutende Teilsprachen der Unified Modelling Language (UML) [Rational Software Corporation u.a. 1997]. Die *Klassendiagramme* weisen die abstrakte Struktur eines Graphen auf. Interessante graphische Eigenschaften der Sprache sind ihr Reichtum an Alternativen für die Darstellung von Kanten sowie die Anwendung von n-stelligen Relationen. Die *Zustandsdiagramme* verwenden gegenüber "normalen" Graphen zusätzlich Hierarchien und erlauben Relationen über Hierarchiegrenzen hinweg. Die Zustandsdiagramme werden exemplarisch im folgenden Abschnitt untersucht. Die dritte untersuchte Teilsprache sind *Ablaufdiagramme*. Der interessante Aspekt ist hier die Restriktion des Pfeilverlaufs, was aus der Beziehung zwischen Sende- und Empfangszeitpunkt der dadurch modellierten Kommunikation resultiert. Die komplexeren Konstrukte wie Verzweigung oder Rekursion wurden nicht berücksichtigt.

Die anderen Sprachen sind verschiedenen Anwendungsgebieten entnommen. Die abstrakte Struktur der *Nassi-Shneiderman-Diagramme* [Nassi und Shneiderman 1973] verwendet hierarchische Beziehungen, die durch ineinander geschachtelte, über- und nebeneinander angeordnete graphische Elemente dargestellt werden. Ein Strukturedi tor kann automatisch eine größenoptimierte Darstellung berechnen. Die Sprache *Query by Example* [Zloof 1975] ist eine formularbasierte Datenbankabfragesprache. Die Tatsache, daß die Struktur der Formulare nicht vorgegeben, sondern einer zugrundeliegenden Datenbank entnommen ist, stellt besondere Anforderungen an einen Strukturedi tor. *Pictorial Janus* [Kahn und Sraswat 1990] ist eine logische Programmiersprache. Ein besonderer Aspekt sind die durch Berührung dargestellten Beziehungen der Sprachelemente. Die auch kommerziell erfolgreiche Sprache *LabView* [Vose und Williams 1986] ist eine datenflußorientierte Programmiersprache für meßtechnische Anwendungen. Ihre Besonderheiten sind die übereinanderliegenden Darstellungen bei Fallunterscheidungen und die benutzerfreundlichen Editieroperationen zur Erzeugung von Datenflußlinien. Schließlich ist *Streets* [Kastens und Jung 1998] eine imperative Programmiersprache, mit der parallele Programme erstellt werden können. Interessant ist hier die Zusammensetzung komplexer Zeichnungen aus einzelnen Piktogrammen sowie die automatisch angeordnete, größenoptimierte Darstellung.

3.4.2. Durchführung der Untersuchung

Ein initialer Satz visueller Muster wurde im Rahmen der Diplomarbeit [Schmidt und Schindler 2000] erarbeitet, indem die ausgewählten Sprachen untersucht wurden. Die Sprachen wurden hierzu zunächst einzeln betrachtet und die jeweils verwendeten visuellen Muster extrahiert. Im Anschluß daran wurden die Muster ver-



(a) Einfache Zustandsmenge (b) Zustandsmenge mit Startzustand (c) Hierarchische Zustandsmenge

Abbildung 3.7.: Beispiele für UML-Zustandsdiagramme

allgemeinert, indem die Ausprägungen eines Musters in verschiedenen Sprachen betrachtet wurden. Dabei wurden auch diverse weitere Ausprägungen in anderen Sprachen berücksichtigt, um ein möglichst breites Anwendungsgebiet der Muster zu erhalten.

Visuelle Muster, die lediglich in einer Sprache auftreten, wurden dabei als Speziallösungen betrachtet und zunächst verworfen. Damit sind wir den Prinzipien gefolgt, die Gamma u.a. angewendet haben, um die Entwurfsmuster objektorientierter Sprachen zu identifizieren: "*Wir haben lediglich solche Entwürfe berücksichtigt, die mehrfach angewendet wurden und sich in unterschiedlichen Systemen bewährt haben.*" [Gamma u.a. 1996, S. 2]. Diese Vorgehensweise macht es wahrscheinlicher, daß die visuellen Muster tatsächlich praktisch relevante Entwurfskonzepte für visuelle Sprachen dokumentieren. In späteren Betrachtungen können die verworfenen Darstellungsvarianten näher untersucht werden, um festzustellen, ob sie trotzdem wiederverwendbare Visualisierungsalternativen für visuelle Sprachen bereitstellen.

Um die Analyse einer visuellen Sprache durchzuführen, wurde zunächst die Struktur der graphischen Darstellung und die abstrakte Struktur der Informationen in dieser Darstellung betrachtet. Dazu reicht es zunächst, verschiedene Sätze einer visuellen Sprache zu untersuchen. Um die Abhängigkeiten verschiedener Sprachkonstrukte voneinander, beispielsweise in Bezug auf das Layout zu untersuchen, wurden ferner (wo möglich) Editieroperationen eines Standardeditors für die Sprache betrachtet.

Als Beispiel für die Untersuchung einer Sprache sind in Abbildung 3.7 einige graphische Darstellungen von UML-Zustandsdiagrammen (hier nach Harel [1988]) dargestellt. Wie man an den Abbildungen sieht, hat die Darstellung der "Programme" die Struktur geschachtelter Mengendiagramme. Die zugrundeliegende abstrakte Struktur ist ein hierarchisches System von Mengen. Auf die-

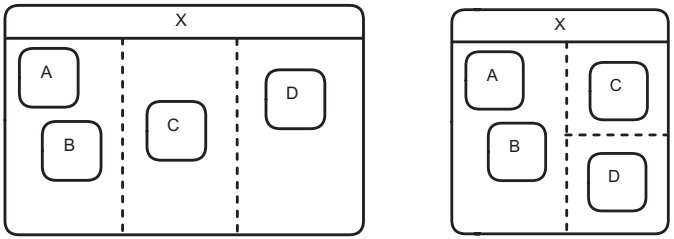
sem System von Mengen wird eine Übergangsrelation durch Pfeile dargestellt, die durch Berührung die Zustände kenntlich machen, zwischen denen ein Übergang existiert. Die Pfeile können mit den Namen von Ereignissen beschriftet sein.

Bei der Untersuchung einer visuellen Sprache stellt sich im weiteren die Frage, welche Sprachkonstrukte man als zu einem Muster gehörig betrachtet. Oft ergeben sich in Abhängigkeit dieser Entscheidung verschiedene Alternativen für visuelle Muster, die als zu verschiedenen Entwurfsebenen gehörig angesehen werden können. Im Beispiel der Zustandsdiagramme ergeben sich verschiedene Kandidaten für Muster, je nachdem ob man die Zustände und die Übergänge als zu einem einzigen Muster oder als zu zwei verschiedenen Mustern gehörig betrachtet. Der erste Fall führt zur Identifikation eines Musters, das eine Menge und eine zweistellige Relation auf dieser Menge durch einen Graphen visualisiert. Beim zweiten Fall würde man zwei kooperierende Muster identifizieren: ein erstes Muster visualisiert Mengen durch Mengen-Diagramme, das zweite Muster stellt Relationen durch Linien dar.

Um zu entscheiden, welche dieser Überlegungen weiter verfolgt werden soll, kann man untersuchen, inwieweit die betrachteten Sprachkonstrukte voneinander abhängen, beispielsweise beim Layout oder bei strukturellen Änderungen. Dahinter steht die Überlegung, daß lediglich die Abhängigkeiten innerhalb eines Musters gekapselt und die dahinterstehenden Entwurfskonzepte dadurch wiederverwendbar gemacht werden können. Die Abhängigkeiten zwischen den Sprachkonstrukten, für die Muster identifiziert werden, können nicht durch die visuellen Muster gekapselt werden. Sie erschweren somit später trotz Anwendung der Muster den Entwurf einer Sprache. Indem man mehrere, in hohem Maße voneinander abhängige Sprachkonstrukte zu *einem* kombinierten Muster abstrahiert, erleichtert man also die Musteranwendung. Dabei nimmt man aber ein eventuell kleineres Anwendungsgebiet des kombinierten Musters in Kauf und erhält dadurch eine größere Anzahl sich nur geringfügig unterscheidender Muster.

Bei den Zustandsdiagrammen könnten umfangreiche Abhängigkeiten zwischen den Zuständen und den Übergängen darin bestehen, daß das Layout von Zuständen und Übergängen gemeinsam durch einen Graph-Layouter bestimmt wird. Die Kombination zweier Muster für Zustände und für Übergänge ließe in diesem Fall die Details unbeachtet, die erforderlich sind, damit der Graph-Layouter ein ansprechendes Layout erzeugen kann. Diese Details müßte ein Sprachentwerfer dann selbst festlegen.

Sind die visuellen Muster einer Sprache identifiziert, so kann man durch weitere Betrachtung der graphischen Darstellungen Anforderungen an die Umsetzung des Musters gewinnen. Wird bei der Untersuchung der Zustandsdiagramme etwa die Darstellung einer Menge durch ein Mengendiagramm als visuelles Muster identifiziert, so muß dieses Muster Strukturen unterstützen, bei denen ein Element der Menge selbst wieder eine Menge ist und durch Anwendung desselben Mu-



(a) Horizontales Layout (b) Kompakte Darstellung

Abbildung 3.8.: AND-Superstates in Zustandsdiagrammen

sters dargestellt wird. Dies ermöglicht die Visualisierung von Hierarchien. Darüberhinaus müssen die Elemente verschiedenartig darstellbar sein, beispielsweise um einfache Zustände, Superstates, History-Zustände und Startzustände der Zustandsdiagramme zu unterscheiden. Diese Hinweise illustrieren verschiedene Anwendungen der visuellen Muster. An ihnen wurde der Entwurf der Spezifikationsmodule überprüft.

Die Sprache der UML-Zustandsdiagramme enthält noch weitere Muster, z.B. in der Darstellung der AND-Superstates. Bei erster Betrachtung der Darstellung in Abbildung 3.8(a) könnte man annehmen, daß die durch die gestrichelten Linien abgetrennten Teile des AND-Superstates grundsätzlich nebeneinander dargestellt werden. Daraus könnte man schließen, daß es sich um die Darstellung einer Folge handelt. Man könnte also die abstrakte Struktur einer Folge und die nebeneinander angeordneten Darstellungen durch ein visuelles Muster abstrahieren.[3]

Harel [1988] hat aber bei den AND-Superstates eher die abstrakte Struktur einer Menge gemeint, bei der die Reihenfolge *keine* Rolle spielt. Das ermöglicht platzoptimierte Darstellungen, wie sie beispielsweise Abbildung 3.8(b) darstellt und in ähnlicher Weise auch bei Harel auftreten. Die abstrakte Struktur einer Menge ist hier durch horizontal und vertikal benachbarte graphische Objekte ausgedrückt, wobei offen bleibt, ob ein Strukturedit or die Darstellung automatisch platzoptimieren muß oder die Anordnung durch einen Benutzer gesteuert wird. Man könnte diese Darstellung zum Anlaß nehmen, um ein weiteres visuelles Muster für die Darstellung von Mengen zu definieren. Da derartige Darstellungen aber (bislang) ausschließlich bei Zustandsdiagrammen auftreten, ist hier darauf verzichtet worden.

[3] Dieses Muster erhält später den Namen Listen-Muster.

3.4.3. Ergebnis

In diesem Abschnitt wird ein Überblick über die erarbeiteten Muster gegeben und auf ihre wesentlichen Eigenschaften eingegangen. Die visuellen Muster werden dazu anhand der von den Mustern visualisierten abstrakten Strukturen zusammengefaßt. Die strukturellen Abstraktionen wurden zusammen mit den Mustern im Verlauf der Untersuchung visueller Sprachen identifiziert. Dies sind zunächst Tupel, Folgen, Mengen und Relationen. Hinzu kommen komplexere Strukturen, die als aus einfacheren Strukturen zusammengesetzt betrachtet werden können, aber meist zusätzliche charakteristische Eigenschaften haben. Tabelle 3.1 gibt einen Überblick über die visuellen Muster.

3.4.3.1. Tupel: das Formular- und das Registerkarten-Muster

Das Tupel ist die strukturelle Abstraktion von Sprachkonstrukten, die aus einer festen Anzahl anderer Sprachkonstrukte bestehen oder in die eine feste Anzahl anderer Sprachkonstrukte eingesetzt werden kann. Viele der Sprachkonstrukte der Nassi-Shneiderman-Diagramme sind Tupel: wie in Abbildung 3.9 links zu sehen ist, können in das IF-Konstrukt (im Bild links oben) genau drei andere Sprachkonstrukte eingesetzt werden, in das Schleifenkonstrukt (rechts darunter) können genau zwei Sprachkonstrukte eingefügt werden. Tupel treten in einer (visuellen) Sprache meist sehr häufig auf. Einige andere abstrakte Strukturen kann man sich auch als rekursiv aus Tupeln zusammengesetzt vorstellen.

Sprachkonstrukte, deren Informationsgehalt durch Tupel abstrahiert werden kann, benötigen lediglich dann Editieroperationen, wenn für die Elemente des Tupels unterschiedliche Sprachkonstrukte eingesetzt werden können. In Abbildung 3.9 können etwa für die unten abgebildeten Tupelelemente der IF-Anweisung verschiedene Sprachkonstrukte eingesetzt werden, hier die Konstrukte “einfache Anweisung”, “IF-Anweisung” und “Schleife”, während für das oben visualisierte Sprachkonstrukt lediglich das Konstrukt “Expr” erlaubt ist. Entsprechend müssen die Anweisungen im IF-Konstrukt durch andere austauschbar sein, während der Ausdruck zwar editiert, aber nicht strukturell durch andere Sprachkonstrukte ersetzt werden kann. Verschiedene Techniken, mit denen Tupel visualisiert werden können, sind das *F*ormular-Muster und das *R*egistrierkarten-Muster.

Durch ein *Formular-Muster* werden die Teile eines Tupels dargestellt, indem die graphische Darstellung der Tupelelemente in eine Zeichnung an bestimmte, feste Stellen eingefügt wird. Eine Zeichnung wird dazu als aus graphischen Elementen und Platzhaltern bestehend aufgefaßt. In Abbildung 3.9 sind auf der linken Seite zwei Zeichnungen abgebildet, die in der Sprache der Nassi-Shneidermann-Diagramme auftreten. Die Platzhalter sind darin durch graue Rechtecke dargestellt. In sie werden die graphischen Darstellungen der Tupelelemente eingesetzt,

Abstrakte Struktur	*Muster*	*Kurzbeschreibung*
Tupel	Formular-Muster	Formularelemente werden in eine dehnbare dynamische Zeichnung eingebettet. Basiert auf dem Ansatz von Grant [1998]. Zeichnungen können auch aus Piktogrammen aufgebaut sein.
	Registerkarten-Muster	Elemente sind übereinandergestapelt um Anzeigefläche zu sparen.
Folge	Listen-Muster	Listenelemente werden horizontal oder vertikal benachbart dargestellt. Kann auch durch rekursive Anwendung dreier dynamischer Zeichnungen dargestellt werden.
	Stapel-Muster	Wie beim Registerkarten-Muster erscheinen die Elemente der Folge übereinandergestapelt. Tritt z.B. in den Fallunterscheidungen in LabView [Vose und Williams 1986] auf.
Mengen	Mengen-Muster	Elemente werden durch ein Mengendiagramm innerhalb eines Mengenrahmens dargestellt.
Relationen	Linien-Muster	Zweistellige Beziehungen werden durch Linien dargestellt.
	Attribut-Relations-Muster	n-stellige Beziehungen werden durch eindeutige Kennzeichnungen dargestellt.
Komplexere Strukturen	Tabellen-Muster	Stelle eine Folge von gleichartigen Tupeln in einer Tabelle dar.
	Matrix-Muster	Stellt die Elemente einer Matrix in einem Gitter dar.
	Graph-Muster	Stellt eine Menge mit darauf definierter zweistelliger Relation dar und setzt einen Graph-Layouter ein.

Tabelle 3.1.: Überblick über visuelle Muster

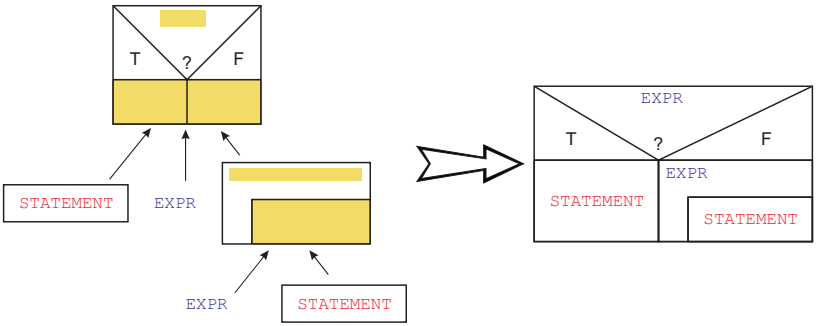


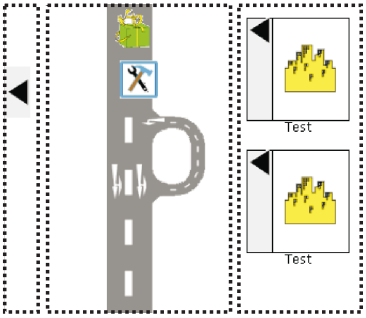
Abbildung 3.9.: Einsetzen in die Zeichnung eines Formular-Musters

wozu die Zeichnungen zunächst in geeigneter Weise skaliert werden müssen. Wie dies genau geschieht, ist im Formular-Muster nicht festgelegt.

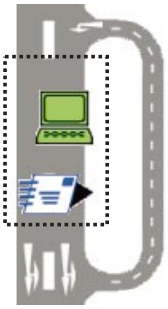
Im Beispiel aus Abbildung 3.9 bilden zwei Anweisungen und ein Ausdruck die Sprachkonstrukte, aus denen eine IF-Anweisung besteht. Ihre graphischen Darstellungen werden in die Zeichnung für das IF-Konstrukt eingefügt. Während die einfache Anweisung nicht weiter unterstrukturiert ist, wird die Schleife wiederum durch Anwendung des Formular-Musters visualisiert. Auf der rechten Seite der Abbildung ist die resultierende Darstellung für das IF-Konstrukt abgebildet. Wie man sieht, wurde die Zeichnung auf der linken Seite dazu entsprechend skaliert.

Das Formular-Muster tritt in den untersuchten Sprachen vielfach auf. Die Sprache der Nassi-Shneiderman-Diagramme läßt sich beispielsweise weitgehend als aus Formularen zusammengesetzt betrachten. Weiterhin lassen sich die Klassen des UML-Klassendiagramms durch Formulare beschreiben. Zeichnungen ohne zusätzliche Zeichnungselemente sind einsetzbar, um etwa wie bei Streets die Anordnung eines Fensters festzulegen, siehe dazu Abbildung 3.10(a). Weitere Einsatzmöglichkeiten ergeben sich, wenn man die Zeichnungen nicht durch Vektorgraphiken sondern durch replizierte Piktogramme (Icons) bildet, siehe Abb. 3.10(b). Schließlich liefern Tupel mit nur einem Tupelelement ein weiteres Anwendungsfeld. Bei diesen Tupeln degeneriert die ein Formular-Muster charakterisierende Zeichnung zu einem Rahmen, in den andere Darstellungen eingebettet werden können.

Während beim Formular-Muster die Bestandteile eines Tupels nebeneinander angeordnet sind, liegen diese beim *Registerkarten-Muster* “hintereinander”; es ist immer nur ein Element des Tupels sichtbar. Dafür enthält die Darstellung weitere graphische Objekte, die einerseits kennzeichnen, welches Element gerade dargestellt wird und andererseits die Umschaltung zwischen verschiedenen Elementen ermöglichen.



(a) Anordnung einer Prozedur

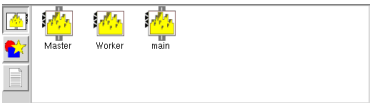


(b) Schleife

Abbildung 3.10.: Formulare in Streets: Platzhalter sind durch gestrichelte Rechtecke gekennzeichnet (vgl. [Schmidt und Schindler 2000, Abb. 17])

Das Registerkarten-Muster tritt bei bei Streets auf. Das in Abbildung 3.11 angezeigte Tupel besteht aus drei verschiedenen Mengen, von denen jeweils eines dargestellt wird. Weitere Anwendungen findet man beispielsweise in Dialogen von Programmen mit graphischen Benutzungsschnittstellen. Die Struktur der graphischen Darstellung entspricht der Darstellung des im folgenden diskutierten Stapel-Musters, das aber auf Folgen statt Tupeln basiert.

Da beim Registerkarten-Muster immer nur ein Teil der Darstellung sichtbar ist, ist das Muster nur dann anwendbar, wenn keine Beziehungen zwischen den Teildarstellungen unterschiedlicher Tupelelemente visualisiert werden sollen. Es hat gegenüber dem Formular-Muster Vorteile, wenn eine u.U. unübersichtliche Informationsflut strukturiert werden muß, weil diese sonst zu viel Platz auf der Bildschirmfläche beanspruchen würde. Die Anwendung des Registerkarten-Musters wirkt besonders elegant, wenn wie bei LabView verschiedene Tupelelemente (oder Folgenelemente beim Stapelmuster) gleiche Teile besitzen, siehe Abbildung 3.13.



(a) Darstellung der Menge der Prozeduren

(b) Darstellung der Menge der Typen

Abbildung 3.11.: Registrierkarten-Muster bei Streets

Durch die umschaltbare Darstellung werden dann die Ähnlichkeiten und Unterschiede verstärkt hervorgehoben.

Bei der Anwendung dieses Musters geht jedoch die Eigenschaft der statischen Darstellbarkeit von Programmen einer visuellen Sprache verloren, da nicht alle Teile des visuellen Programms gleichzeitig sichtbar sind. Der vermehrte Einsatz des Registerkarten-Musters in hierarchischen Strukturen kann sich dadurch negativ auswirken, wie Green und Petre [1992] am Beispiel LabView zeigen. Poswig [1996, S.73-74] empfiehlt deshalb, bei Platzproblemen nicht das Registerkarten-Muster anzuwenden, sondern das vollständige Tupel durch ein Formular-Muster in einem neuen Fenster darzustellen.

3.4.3.2. Folgen: das Listen- und das Stapel-Muster

Eine Folge besteht aus einer variablen Anzahl möglicherweise verschiedenartiger Elemente. Die Reihenfolge der Elemente wird in den Mustern auf unterschiedliche Weise dargestellt. Um Folgen zu editieren, müssen Elemente an beliebigen Stellen der Folge eingefügt und existierende Elemente der Folge gelöscht werden können. Sinnvoll ist weiterhin, wenn ein Benutzer die Anordnung der Elemente ändern kann. Für Folgen haben wir das Listen- und das Stapel-Muster identifiziert.

Das *Listen-Muster* visualisiert eine Folge, indem die Teildarstellungen der Folgenelemente entlang einer Richtung angeordnet werden. Die Reihenfolge der Teildarstellungen entspricht dabei der Reihenfolge der Elemente in der Folge. Um die graphische Darstellung eines Listen-Musters zu erzeugen, lassen sich verschiedene Strategien einsetzen. Hier werden zwei verschiedene benutzt, eine einfache und eine formularbasierte Strategie. Diese Strategien bestimmen die Anwendbarkeit und den Spezifikationsaufwand des Listen-Musters.

Die einfache Strategie reiht die Darstellungen der Folgenelemente in horizontaler oder vertikaler Richtung auf. Einstellungsmöglichkeiten erlauben die Ausrichtung der Darstellungen oder die Skalierung auf gleiche Breite bzw. Höhe. Weiterhin können als zusätzliche Zeichnungselemente Trennungslinien und ein Rahmen verwendet werden. Die einfache Strategie reicht bereits aus, um eine große Palette verschiedener Anwendungen des Listen-Musters umzusetzen. In den untersuchten Sprachen sind dies beispielsweise die Anordnung der Prozesse in UML-Ablaufdiagrammen (Abb. 3.5(a)) sowie die Anordnung der Attribute und der Methoden in einer Klasse eines UML-Klassendiagramms (Abb. 3.5(c). Faßt man die Teile eines AND-Superstate in UML-Zustandsdiagrammen als Folge auf (siehe hierzu Abschnitt 3.4.2), so kann man auch die Darstellung eines AND-Superstate (Abb. 3.8(a)) durch ein einfaches Listen-Muster erhalten.

Bei der formularbasierten Strategie wird die Folge als rekursiv aus Tupeln bestehend aufgefaßt. Ein Tupel besteht hierbei aus einem Element und einem “Folgenrest” und wird durch Anwendung des Formular-Musters dargestellt. Diese Strategie kann bei komplexeren Anwendungen des Listen-Musters eingesetzt wer-

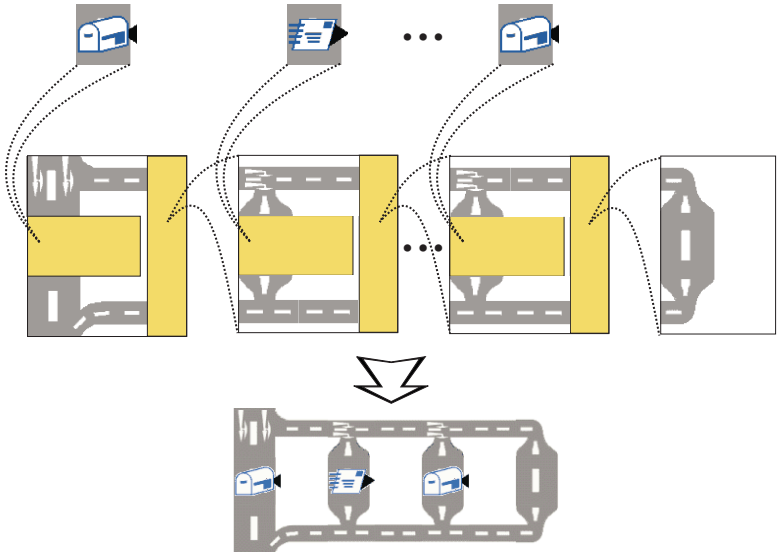


Abbildung 3.12.: Zusammensetzen von Zeichnungen beim formularbasierten Listen-Muster

den, bei denen in der Darstellung zusätzliche Zeichnungselemente für jedes Element benötigt werden. Anwendungen in den untersuchten Sprachen sind die Fallunterscheidungen (switch-Konstrukte) in Nassi-Shneiderman-Diagrammen und Streets.

Das Zusammensetzen von Zeichnungen bei einem formularbasierten Listen-Muster ist in Abbildung 3.12 veranschaulicht. In der Mitte ist der Satz von Zeichnungen abgebildet, der die graphische Darstellung der Fallunterscheidungen in Streets erzeugt. Es handelt sich um drei Zeichnungen, von denen die mittlere bei Bedarf so oft repliziert wird, bis die gewünschte Anzahl von Platzhaltern für die Folgenelemente vorhanden ist. Die in der Abbildung oben dargestellten Folgenelemente werden dann eingesetzt, um das unten abgebildete Resultat zu erhalten.

Die Strategie, eine Folge durch rekursiv angewendete Formulare darzustellen, ist dem System VPE [Grant 1998] entnommen. Die Umsetzung im VPE-System basiert jedoch auf rekursiv angewendeten Tupeln, nicht auf Folgen. Das führt in visuellen Sprachen, die damit implementiert sind zu gewöhnungsbedürftigen Editieroperationen. Um zum Beispiel an einer beliebigen Stelle einer Folge ein Element einzufügen, muß zunächst die Folge aufgespalten werden. Anschließend

(a) True-Fall (b) False-Fall

Abbildung 3.13.: Darstellung der Fallunterscheidung in LabView durch das Stapel-Muster nach Schiffer [1998, S. 187]

wird ein Platzhalter “verdoppelt”. Danach kann das neue Element und der Folgenrest eingefügt werden.

Das Stapel-Muster stellt die Elemente einer Folge nicht nebeneinander, sondern hintereinander, auf derselben Anzeigefläche dar. Es entspricht somit in der Struktur der Darstellung dem Registerkarten-Muster, basiert aber auf Folgen statt auf Tupeln. Das Stapel-Muster wird in LabView und in einer leicht abgewandelten Form auch in Prograph [Cox u.a. 1989] verwendet.

Abbildung 3.13 zeigt beide Fälle einer Fallunterscheidung über einen Wahrheitswert in LabView (im allgemeinen Fall kann dasselbe Konstrukt aber auch für Fallunterscheidungen über ganze Zahlen verwendet werden). Die Anwendung des Stapel-Musters in LabView hebt die gemeinsame Schnittstelle einer Fallunterscheidung nach außen hin hervor.

In Prograph wird das Stapel-Muster ebenfalls verwendet, um Fallunterscheidungen eines Datenflußdiagramms zu veranschaulichen. Die verschiedenen Fälle erscheinen hier jedoch in verschiedenen Fenstern anstatt hintereinander. Das ermöglicht es einem Benutzer, mehrere oder alle Fälle gleichzeitig zu sehen.

Die Art der Darstellungen von Folgen entsprechen denen von Tupeln. Die dort angeführten Vor- und Nachteile der Muster treffen auch hier zu.

3.4.3.3. Mengen: das Mengen-Muster

Wie Folgen bestehen Mengen aus einer variablen Anzahl von Elementen, die auch bei Mengen verschiedenartig sein können. Auch bei Mengen erwartet man in der Regel Editieroperationen zum Einfügen und Löschen von Elementen, es kommt dabei aber nicht auf die Einfügeposition in der Menge an.

Für Mengen ist bislang lediglich ein Muster, das sog. Mengen-Muster identifiziert. Eine Alternative könnte aus der Darstellung der AND-Superstates abgeleitet werden, siehe Abschnitt 3.4.2. Diese Art der Darstellung einer Menge tritt jedoch

(bislang) in keiner anderen Sprache auf und wurde deshalb nicht zu einem visuellen Muster abstrahiert.

Beim *Mengen-Muster* werden die Teildarstellungen der Elemente der zugrundeliegenden Menge an beliebigen, meist durch den Benutzer festgelegten Punkten innerhalb eines Rahmens positioniert. Durch die freie Wahl der Positionen kann ein Benutzer die Elemente der Menge nach eigenen Vorstellungen strukturieren. Beispielsweise kann ein Benutzer eines UML-Klassendiagramms die Wurzel der Klassenhierarchie oben und davon abgeleitete Klassen darunter plazieren, um die Hierarchie hervorzuheben. In UML-Zustandsdiagrammen können so die Zustände eines XOR-Superstates sinnvoll gruppiert und so die Übergänge zwischen den Zuständen besser verdeutlicht werden.

Die freie Wahl der Positionen der Elemente ist also wichtig, um die Menge nach sekundären Kriterien strukturieren zu können. Das Mengen-Muster kann einen Benutzer bei der Positionierung aber unterstützen. Beispielsweise kann geprüft werden, daß sich die Teildarstellungen der Elemente nicht überlappen oder über den Rand der Menge hinausragen. Sind diese Bedingungen verletzt, so können die Positionen der Elemente durch einen Constraint-Solver so korrigiert werden, daß sich die Darstellung insgesamt möglichst wenig ändert. Weiterhin können Editieroperationen bereitgestellt werden, mit denen ein Benutzer die Elemente der Menge nach bestimmten Kriterien sortiert, in Zeilen oder Spalten anordnen kann. Derartige Operationen findet man beispielsweise in Prograph [Cox u.a. 1989] und auch oft in graphischen Benutzungsschnittstellen von Betriebssystemen, z.B. im Explorer von Microsoft Windows oder im Dateimanager von IBM OS/2. Eine interessante Lösung dafür bietet auch Argo/UML: Hier kann der Benutzer mehrere Mengenelemente gleichzeitig mit einem "Schneeschieber" anordnen, dessen Breite er einstellen kann.

In vielen Sprachen wird das Mengen-Muster zusammen mit dem Linien-Muster (s.u.) verwendet, um Graphen darzustellen. Das Mengen-Muster tritt in visuellen Sprachen aber auch ohne zwischen den Elementen gezeichnete Linien auf. In Streets und Prograph werden so beispielsweise Mengen von Definitionen durch Piktogramme (Icons) dargestellt. In Prograph bilden diese Mengen sogar eine Hierarchie, siehe z.B. Abbildung 4.8(a) auf Seite 4.8(a).

3.4.3.4. Relationen: Das Linien- und das Attribut-Relations-Muster

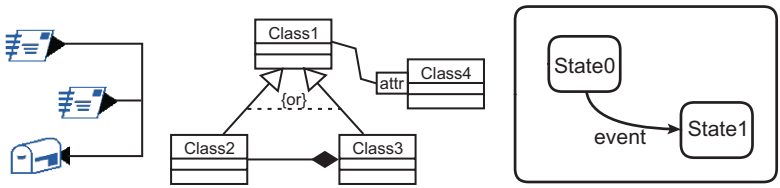
Relationen abstrahieren die Beziehungen zwischen mehreren Konstrukten eines visuellen Programms. Solche Beziehungen werden in visuellen Sprachen oft, aber nicht grundsätzlich durch Linien visualisiert. Eine Datenflußlinie beschreibt beispielsweise eine Datenflußoperation, also eine Beziehung zwischen einer Datenquelle und einer Datensenke. Eine Menge solcher Operationen kann durch eine Relation abstrahiert werden. Andere Relationen in visuellen Sprachen sind verschiedenartige Beziehungen zwischen den Klassen ei-

nes UML-Klassendiagramms, Übergänge zwischen den Zuständen eines UML-Zustandsdiagramms oder Definiert-Benutzt-Relationen, die in vielen (visuellen) Sprachen vorkommen.

Die Relationen in den untersuchten Sprachen wurden in zwei visuelle Muster zusammengefaßt. Im Linien-Muster werden Relationen durch Linien dargestellt. Im Attribut-Relations-Muster werden eindeutige Marken bei den Sprachkonstrukten verwendet, die an einer Beziehung beteiligt sind. Die Muster verwenden unterschiedliche Editieroperationen.

Das *Linien-Muster* stellt Relationen zwischen Konstrukten eines visuellen Programms durch Linien dar. Eine Beziehung setzt dabei genau zwei Programmstellen miteinander in Relation und verbindet die Darstellungen dieser Programmstellen durch Linien. Das graphische Erscheinungsbild der Linie wird dabei bestimmt durch die Linienführung, die Zeichnungen an den Linienenden, eventuelle Annotationen an einer Linie sowie durch Linienfarbe und -stil. Verschiedene Arten der Linienführung sind direkte Linien, (direkte Verbindung der Endpunkte), orthogonale Linien (senkrechte und waagerechte Liniensegmenge), Polylinien (mehrere Liniensegmenge) und Splines. An den Linienenden können etwa verschiedenartige Pfeilspitzen erscheinen oder auch Detailinformationen dargestellt werden.

Abbildung 3.14 enthält Beispiele für Linien in verschiedenen Sprachen. In Streets werden Sendeoperationen zu einer Datensenke zusammengefaßt und durch eine orthogonale Linie unter Verwendung eines einzigen vertikalen Segments dargestellt (Abb. 3.14(a)). In UML-Klassendiagrammen werden unterschiedliche Beziehungen durch verschiedene Linientypen dargestellt. Assoziationen (in Abb. 3.14(b) zwischen Class1 und Class4) haben weitere Eigenschaften, die am Linienende dargestellt werden können. Weiterhin können auch Linien wiederum durch Linien verbunden werden. Schließlich treten in Zustandsdiagrammen, zumindest in der Darstellung von Harel [1988] Splines auf (Abb. 3.14(c)).



(a) Streets (b) UML-Klassendiagramme (c) UML-Zustandsdiagramme

Abbildung 3.14.: Linien in verschiedenen visuellen Sprachen

Bei der Realisierung des Linien-Musters müssen neben der Darstellung der Linien zusätzlich Editieroperationen angeboten werden, mit denen Linien erzeugt

und gelöscht werden können. Enthalten Linien zusätzliche Informationen (wie bei den Assoziationen der Klassendiagramme) so ist es wünschenswert, zusätzlich die Endpunkte einer Linie ändern zu können, ohne dazu die Linie löschen und neu erzeugen zu müssen. Falls die Linienführung bei orthogonalen Linien, Polylinien oder Splines manuell beeinflußt werden kann, so sind auch dafür Interaktionen vorzusehen.

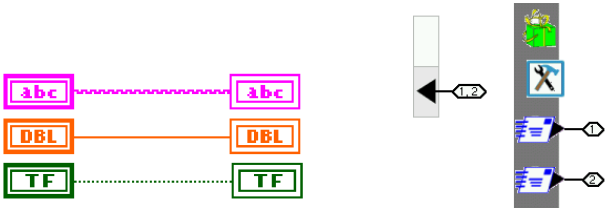
In den verschiedenen betrachteten Sprachen werden zum Bearbeiten der Linien unterschiedliche Interaktionen verwendet. In LabView werden beispielsweise nacheinander die zu verbindenden Stellen angegeben, wozu vorher jedoch in den Verdrahtungsmodus gewechselt werden muß. Zur Erzeugung einer Datenflußkante wird in LabView dabei gegebenenfalls automatisch eine zusätzlich benötigte Schnittstelle zwischen den verbundenen Bauteilen erzeugt. In Streets ist die Verbindung ohne Moduswechsel realisiert, indem nacheinander geeignete Stellen markiert und die Verbindung durch Anwahl einer Verbindungsfunktion bestätigt wird. Bei Rational Rose, einer Implementierung der UML-Klassendiagramme, werden schließlich die Beziehungen zwischen den Klassen durch Ziehen einer Datenquelle auf eine Datensenke erzeugt, wobei wieder vorher ein Moduswechsel die Art der zu zeichnenden Linie festlegt.

Es scheint bislang offenbar noch keine allgemein anerkannten Konventionen für Interaktionen zu geben, mit denen Linienoperationen in visuellen Sprachen angestoßen werden sollten. Für die Realisierung der Editieroperationen können also keine Empfehlungen aufgrund der Analyse existierender visueller Sprachen gegeben werden. Allgemeine Richtlinien für Benutzungsschnittstellen empfehlen jedoch, auf unnötige Moduswechsel zu verzichten.

Ein weiterer interessanter Aspekt beim Linien-Muster sind Relationen zwischen mehr als zwei Endpunkten. In UML-Klassendiagrammen können beispielsweise Assoziationen auch zwischen mehr als zwei Klassen spezifiziert und mit einer zusätzlichen Assoziationsklasse annotiert werden. Die Linienführung in Streets (Abb. 3.14(a)) kann ebenfalls als eine einzige Relation zwischen den beteiligten Endpunkten angesehen werden.

Um solche Relationen mit Hilfe des Linien-Musters darstellen und editieren zu können, kann das Linien-Muster auf n-stellige Relationen generalisiert werden. Alternativ kann aber eine n-stellige Beziehung auch als n zweistellige Beziehungen aufgefaßt werden. In der Sprache der UML-Klassendiagramme könnte so beispielsweise zunächst ein Assoziationspunkt (samt Assoziationsklasse) zur Darstellung hinzugefügt und dieser im weiteren mit den verschiedenen Klassen durch einzelne Linien verbunden werden.

Eine andere Methode, um Relationen in visuellen Programmen darzustellen, ist das *Attribut-Relations-Muster*. Das Muster ist nicht auf zweistellige Relationen beschränkt und verwendet statt einer Linie ein identifizierendes Merkmal. Dieses Merkmal ist beispielsweise ein Name, eine Zahl, eine Farbe, ein Linientyp, eine



(a) LabView (Farben und Linientyp) (b) Streets (Zahlen)

Abbildung 3.15.: Visualisierung von Beziehungen durch Merkmale beim Attribut-Relations-Muster

Schraffur oder eine Kombination dieser Attribute und wird an allen, an einer Beziehung beteiligten Programmstellen in die Darstellung integriert.

Das Muster kommt in zwei verschiedenen Ausprägungen vor. In der ersten Ausprägung kann eine Programmstelle genau einmal mit anderen Programmstellen in Beziehung gesetzt werden. Hier lassen sich identifizierende Merkmale direkt in die Darstellung der in Beziehung gesetzten Programmkonstrukte integrieren, beispielsweise indem Teile der Darstellung eindeutig eingefärbt werden, indem ein bestimmter Linienstil benutzt wird oder indem ein eindeutiger Name verwendet wird.

Ein Beispiel für derartige Beziehungen sind Definiert-Benutzt-Beziehungen. Verschiedene visuelle Sprachen verwenden Namen, um diese zu visualisieren, z.B. Namen von Komponenten in Streets, Namen von Klassen in UML-Klassendiagrammen, Namen von Schaltkreisen in LabView. Alternativen sind Schraffuren für Matrixelemente in Forms/3 (siehe Abb. 4.17 auf Seite 141) oder Kombinationen aus Farbe und Linienstil zur Kennzeichnung des Datentyps in LabView (siehe Abb. 3.15(a)).

Bei der anderen Ausprägung des Musters gibt es Programmstellen, die an mehreren Beziehungen der Relation gleichzeitig beteiligt sein können. Bei der Anwendung in Streets (Abb. 3.15(b)) können beispielsweise einige Datenflußbeziehungen durch eindeutig vergebene Zahlen statt Linien angezeigt werden, wobei an manchen Programmstellen mehrere Zahlen eingefügt werden müssen. Das stellt größere Anforderungen an die graphische Darstellung und schränkt die graphischen Merkmale ein, die dur Kennzeichnung der Beziehung verwendet werden können. In VPE [Tan und Cai 1995][4] tritt das Muster ebenfalls auf. Hier werden farblich gekennzeichnete Kreise übereinander gezeichnet.

[4]Es handelt sich hierbei um eine visuelle Programmiersprache, nicht zu verwechseln mit dem VPE-System von Grant [1998]

Falls sehr viele Beziehungen in einem visuellen Programm durch Linien dargestellt werden, besteht die Gefahr, daß der Überblick verloren geht. Ein Beispiel hierfür ist die Sprache PARTS, bei der selbst einfache Programme der Bezeichnung "Spagetti-Code" eine neue Bedeutung geben [Sayles u.a. 1994, Abb. 9.36 auf Seite 292, auch in Schiffer 1998, Abb. 3-8, Seite 89]. Das Attribut-Relations-Muster kann hier Abhilfe schaffen. Es kann darüberhinaus auch Beziehungen visualisieren, bei denen die beteiligten Programmstellen entweder nicht alle sichtbar oder in unterschiedlichen Fenstern enthalten sind[5]. Das Attribut-Relations-Muster ist aber auch weniger flexibel als das Linien-Muster. Bei Linien können an den Linienenden oder in Annotationen zusätzliche Informationen dargestellt werden. Beispielsweise ließen sich die Übergänge in Zustandsdiagrammen nur schwer durch das Attribut-Relations-Muster darstellen, weil die Beschriftungen der Übergänge nicht darstellbar wären.

3.4.3.5. Kombinationen: das Tabellen-, das Matrix- und das Graph-Muster

Die in diesem Abschnitt vorgestellten visuellen Muster basieren auf abstrakten Strukturen, die wie schon andere Strukturen zuvor als aus einfacheren Strukturen zusammengesetzt betrachtet werden können. Identifizierend für die in diesem Abschnitt präsentierten visuellen Muster ist, daß sie alternativ auch durch Kombination anderer Muster präsentiert werden könnten. Wie schon in Abschnitt 3.4.2 skizziert, würden dabei aber einige Abhängigkeiten zwischen den Programmkonstrukten, z.B. beim Layout oder bei der Implementierung von Editieroperationen unberücksichtigt bleiben. Diese Abhängigkeiten werden durch die hier vorgestellten Muster berücksichtigt.

Bislang wurden drei Muster identifiziert, die vergleichbar auch durch Kombinationen anderer Muster ersetzt werden könnten. Das Tabellen-Muster stellt eine Folge von Tupeln dar, die durch das Listen- und das Formular-Muster visualisiert werden könnten. Das Matrix-Muster basiert auf Matrizen, die sich aber auch als Folge von Folgen auffassen und durch zweifache Anwendung des Listen-Musters präsentieren lassen. Das Graph-Muster präsentiert Graphen, die auch durch Mengen und Relationen beschrieben werden können. Den Mehrwert, den das Tabellen-, das Matrix-, und das Graph-Muster zur Verfügung stellen, wird in den folgenden Abschnitten diskutiert.

Das *Tabellen-Muster* stellt Informationen dar, die als eine Folge oder Menge gleichartiger Tupel strukturiert sind. Diese Informationen werden in einer Tabelle präsentiert, in der die Tupel die Zeilen und zueinander passende Elemente verschiedener Tupel die Spalten bilden (oder umgekehrt). Die Gleichartigkeit der Tupel wird zum Ausdruck gebracht, indem gleichartige Tupelelemente in einer Spalte (oder Zeile) ausgerichtet sind und eine gemeinsame Überschrift haben. Dies

[5] hierzu werden z.T. auch Linien benutzt, beispielsweise in PARTS und in Forms/3

könnte nicht erreicht werden, wenn man ein Listen-Muster für die Folge und ein Formular-Muster für die Tupel anwenden würde.

Das Tabellen-Muster tritt in der Sprache Query-By-Example (QBE) auf. Hier ist interessant, daß die Struktur der Tabelle nicht fest vorgegeben ist, sondern durch die Definition der zugrundeliegenden Datenbank bestimmt wird. Eine auf QBE basierende Abfragesprache ist auch in Microsoft Access enthalten.

Die Idee des Tabellen-Musters wird durch das *Matrix-Muster* fortgeführt. Das Matrix-Muster visualisiert Matrizen. Matrizen können als Folgen von Folgen aufgefaßt werden, bei denen die inneren Folgen die gleiche Länge besitzen und gleichartige Elemente enthalten. Der Zusammenhang der Matrixelemente untereinander wird dargestellt, indem die zu den Elementen gehörenden Darstellungen an den Positionen eines Gitters angeordnet werden. Es werden Editieroperationen benötigt, mit denen man Zeilen und Spalten zu der Matrix hinzufügen oder aus der Matrix löschen kann.

Das Matrix-Muster kommt in den untersuchten Sprachen nicht vor und ist in visuellen Sprachen auch nicht weit verbreitet. Trotzdem läßt sich das Muster sinnvoll einsetzen, beispielsweise, um Editoren für mathematische Formeln zu implementieren. Im weiteren könnte es auch eingesetzt werden, um eine visuelle Sprache für die piktogrammbasierten Zeichnungen umzusetzen, die beim Formular-Muster benötigt werden, siehe [Schmidt und Schindler 2000, Abb. 48 auf Seite 64].

Das Matrix-Muster kann durch zweifache Anwendung des Listen-Musters nur ungenügend ersetzt werden. Hierbei würde man nicht das Layout einer Matrix in der Darstellung erhalten. Weiterhin würden die so zur Verfügung stehenden Editieroperationen auch nicht die gitterartige Struktur respektieren.

Im *Graph-Muster* werden Graphen visualisiert. Graphen lassen sich durch eine Menge und durch eine zweistellige Relation über den Elementen der Menge repräsentieren. Die graphische Darstellung des Graph-Musters ist ein Graph, bei dem die Elemente der Menge die Rollen der Knoten und die Relationen die Rolle von Kanten übernehmen.

Graphen kommen in visuellen Sprachen sehr häufig vor und können prinzipiell auch realisiert werden, indem das Mengen-Muster mit dem Linien-Muster kombiniert wird. Der Vorteil des Graph-Musters besteht hierbei darin, daß komplexere Abhängigkeiten (s.u.) zwischen den Elementen der Menge und den Linien existieren können. Diese Abhängigkeiten werden durch das Graph-Muster gekapselt, während sie bei der Kombination des Mengen-Musters mit dem Linien-Muster durch den Sprachentwickler berücksichtigt werden müssen.

Einige der oben genannten Abhängigkeiten zwischen den Knoten und den Kanten sind beispielsweise Editieroperationen und die Anbindung eines Graph-Layouters. Bei den Editieroperationen besteht eine solche Abhängigkeit, wenn beim Löschen eines Knotens auch die an diesem Knoten endenden Kanten mitgelöscht werden sollen. Ein Graph-Layouter berücksichtigt die Knoten und die

Kanten und verbessert die Linienführung oder ändert die Positionen der Knoten so, daß der Kantenverlauf leichter verständlich ist. Der Graph-Layouter sollte dabei so verwendet werden, daß er durch einen Benutzer explizit bei Bedarf und nicht automatisch zwischen je zwei Editieroperationen aufgerufen wird. Auf diese Weise behindert er einen Benutzer nicht durch ständiges Ändern der Knotenpositionen und nur geringfügig durch seine Laufzeit.

3.5. Literaturbezug

In diesem Kapitel habe ich *visuelle Muster* eingeführt. Visuelle Muster kapseln das Entwurfs-Know-How visueller Sprachen und vereinfachen so deren Entwicklung. Verwandte Ansätze habe ich in Abschnitt 2.2.1 beschrieben. In Abschnitt 2.1.3 habe ich die Arbeiten zusammengefaßt, die Hinweise für den Entwurf guter Notationen für eine visuelle Sprache geben. In diesen Abschnitt will ich Bezüge meiner Arbeiten zu Arbeiten anderer herstellen.

Visuelle Muster beschreiben die graphische Darstellung von Sprachkonstrukten visueller Sprachen und die Editieroperationen, die zur Interaktion mit diesen Sprachkonstrukten verwendet werden können. Die Muster kapseln so wiederverwendbare Entwurfsentscheidungen für Konstrukte visueller Sprachen, die ein Sprachentwerfer leicht anhand der Struktur einer Sprache identifizieren und anwenden kann.

Durch Kombination der visuellen Muster kann ein Großteil einer neuen visuellen Sprache beschrieben werden. Ein Sprachentwerfer hat dabei zahlreiche und weitreichende Realisierungsalternativen, indem er einerseits offen gelassene Details der Darstellung und der Interaktion unterschiedlich festlegt und andererseits verschiedene visuelle Muster für ein Sprachkonstrukt anwendet. Für die Auswahl eines Musters wird er durch die Konsequenzen unterstützt, die ich der Anwendung der Muster in Form von Auswirkungen auf die kognitiven Aspekte der Sprachbenutzung [Green und Petre 1996] zugeordnet habe. Bei Entwurfsalternativen wird somit eine Entscheidungsgrundlage geboten.

Es gibt eine ganze Reihe alternativer Techniken, um den Entwurf visueller Sprachen zu erleichtern. Die “Cognitive Dimensions” von [Green und Petre 1996] können etwa bereits in einem frühen Stadium den Entwurf einer Sprache beeinflussen. Yang u.a. [1997] entwickeln aus ihnen Benchmarks, anhand derer die statische Repräsentation einer neuen visuellen Sprache objektiv eingeschätzt werden kann. Weitere, für die Skalierbarkeit einer visuellen Sprache bedeutende Entwurfshinweise geben Burnett u.a. [1995]. Schließlich hat Whitley [1997] einige Untersuchungen zusammengetragen, in denen verschiedene Techniken der visuellen Darstellung von Sprachkonstrukten miteinander verglichen werden. Derartige Arbeiten erleichtern zwar den Entwurf visueller Sprachen, können jedoch nicht

als Bibliothek erfolgreicher Entwürfe benutzt werden, aus der Techniken wiederverwendet und miteinander komponiert werden können.

Eine Reihe weiterer Arbeiten präsentiert einen Katalog visueller Techniken. Das in diesen Arbeiten dokumentierte Expertenwissen ist in der vorliegenden Form aber nur schlecht für den Entwurf visueller Sprachen benutzbar. Beispiele sind die Arbeiten von Selker und Koved [1988], die einen Überblick über Techniken geben, die für die Visualisierung von Informationen in visuellen Sprachen benutzt werden können, sowie die Arbeiten von Lohse u.a. [1994], die eine Kategorisierung von 60 visuellen Darstellungen versuchen.

Unter den Arbeiten, die Kategorien zur Unterteilung visueller Sprachen vorschlagen, beschreiben einige auch Visualisierungstechniken, die beim Sprachentwurf als Bibliothek verwendet werden können. In der Klassifikation von Myers [1994, 1990] werden Sprachen etwa anhand wesentlicher Charakteristika der Darstellung unterschieden. Costagliola u.a. [1997a] unterteilen visuelle Sprachen nach den Techniken, die diese hautsächlich für die Visualisierung ihrer Beziehungen verwenden. Die Arbeit von Costagliola u.a. ist für diese Arbeit dabei von besonderem Interesse, da sie als einzige auch als Basis für die Implementierung visueller Sprachen benutzt werden kann.

Wesentliche Eigenschaften der visuellen Muster, die in den oben genannten Ansätzen fehlen, sind einerseits die Auswirkungen, die die Benutzung der visuellen Muster für eine Sprache mit sich bringt und die in Form von Konsequenzen den visuellen Mustern zugeordnet wurden. Weiterhin decken die Muster nicht nur Visualisierungstechniken sondern auch Techniken der Interaktion mit der graphischen Darstellung ab. Visuelle Muster helfen daher beim Sprachentwurf in einen höheren Maße als andere Techniken.

Zum Abschluß dieses Kapitels sollen Arbeitsgebiete aufgeführt werden, die die Bezeichnung “Visuelle Muster” mit anderen Bedeutungen benutzen. Sajeev [1998] bezeichnet mit “visual patterns” etwa visuelle Darstellungen von Mustern zur parallelen Lösung bestimmter Problemklassen (parallele Paradigmen). In ähnlicher Weise bezeichnen auch Toyoda u.a. [1997] sowie Shizuki u.a. [1999] parallele Paradigmen mit “visual design patterns”, an die die Autoren aber gewisse Anforderungen an die Implementierung stellen. Schließlich identifizieren Kranzlmüller u.a. [1999] Muster der Kommunikation paralleler Programme. Sie nennen diese Muster ebenfalls “visual patterns” und entwickeln zu ihrer Identifizierung bei der Fehlersuche eine visuelle Spezifikationssprache.

Mit “visual patterns” wird schließlich auch das Problem bezeichnet, bestimmte Muster in Bildern zu erkennen. Chen und Bovik [1990] verwenden sie beispielsweise zur Komprimierung der Bilder, Fukushima [1995] findet Muster in Bildern mit Hilfe neuronaler Netzwerke.

4. Generieren von Strukturediteren mit VLEli

Visuelle Muster beschreiben verallgemeinerte Entwürfe für Teile visueller Sprachen. Sie kapseln erfolgreiche Methoden der graphischen Präsentation von Informationen und die Interaktionstechniken, die für die Änderung dieser Informationen benötigt werden. Die Anwendung der Muster bei der Entwicklung neuer Sprachen macht einem Sprachentwickler das Expertenwissen zugänglich, das durch sie gekapselt wird. Der Einsatz visueller Muster vereinfacht den Entwurf und ermöglicht ein hohes Abstraktionsniveau der Sprachbeschreibung. Das vorige Kapitel hat visuelle Muster eingeführt und einen initialen Satz dieser Muster beschrieben.

Über den Nutzen der Muster bei der Konzeption neuer Sprachen hinaus sollen die Muster verwendet werden, um die Implementierung visueller Sprachen zu vereinfachen. Dabei verfolgt diese Arbeit das Ziel, mit einfachen und flexiblen Spezifikationen eines hohen Niveaus Sprachimplementierungen für eine große Menge visueller Sprachen zu generieren. Diese Ziele werden erfüllt, indem eine mehrstufige Spezifikation benutzt wird. Visuelle Muster werden durch Spezifikationsmodule implementiert, die Spezifikationen einer grundlegenden, technischen Spezifikationsebene kapseln.

Dieses Kapitel beschreibt diese grundlegende Spezifikationsebene. Sie unterstützt eine große Menge visueller Sprachen und erzeugt für diese effiziente, benutzerfreundliche Sprachimplementierungen. Benutzerfreundlichkeit wird dabei erreicht, indem vielfältige Layoutalternativen und vielfältige Interaktionsmöglichkeiten angeboten werden. Hierfür werden in der Spezifikation weitgehende Einflußmöglichkeiten auf die erzeugte Sprachimplementierung benötigt, die von einem unerfahrenen Sprachentwickler zwangsläufig als komplexe technische Details wahrgenommen werden. Eine wichtige Eigenschaft dieser Spezifikationsebene ist deshalb die Unterstützung von Abstraktionskonzepten, mit deren Hilfe eine darüberliegende, einfache und flexible Spezifikationsebene realisiert werden kann.

Im vorigen Kapitel wurde bereits ein Überblick über die Methode der Sprachimplementierung gegeben, der an dieser Stelle nicht wiederholt werden soll. Abschnitt 4.1 diskutiert den Einsatz attributierter Grammatiken in dieser Arbeit. In

diesem Abschnitt werden unter anderem Abstraktionstechniken für attributierte Grammatiken vorgestellt und der Einsatz des Liga-Systems begründet.

Im darauffolgenden Abschnitt wird die Anwendung von Strukturbäumen zur Repräsentation der Programme visueller Sprachen beschrieben. Die Beziehungen in visuellen Programmen werden dargestellt, indem eine Methode zur Repräsentation von Beziehungen textueller Programme wiederverwendet wird. So können einerseits allgemeine Werkzeuge der Sprachimplementierung wie z.B. Generatoren für attributierte Grammatiken für die Implementierung visueller Sprachen wiederverwendet werden. Andererseits können zu visuellen Sprachen textuelle Notationen eingeführt werden, für deren Analyse und Weiterverarbeitung sich Werkzeuge aus gemeinsamen Spezifikationen herstellen lassen. Ein Beispiel für eine visuelle Sprache, für die eine äquivalente textuelle Notationen existiert, ist etwa SDL [Belina u.a. 1991].

Abschnitt 4.3 beschreibt die Spezifikation der Fenster eines Strukturediteors. Dieser Abschnitt geht auch auf eine Optimierungsmaßnahme ein, die die Erzeugung der graphischen Darstellung beschleunigt und ihre Spezifikation vereinfacht. Sie ist anwendbar, weil für die Darstellung eines Fensters fast nur Informationen aus einem Teil des Strukturbaums benötigt werden.

Die folgenden beiden Abschnitte beschreiben die Attributberechnungen, die die graphische Darstellung eines Fensters erzeugen. Abschnitt 4.4 geht auf die Berechnungen zur Unterstützung verschiedener Layouttechniken ein und beschreibt auch die Anbindung des Constraint-Solvers. Abschnitt 4.5 beschreibt die Festlegung der Editieroperationen, wozu Operationen aus einem Framework benutzt werden.

Abschnitt 4.6 beschreibt schließlich die Spezifikation von Werkzeugen, die die Analyse und Weiterverarbeitung visueller Programme durchführen. Eine bedeutende Verbesserung der Benutzbarkeit der Sprachimplementierungen kann dabei erzielt werden, indem anhand von Analyseergebnissen automatisch Strukturänderungen durchgeführt werden.

In den folgenden Abschnitten meine ich mit “Anwender” und “Benutzer” die Anwender oder Anwenderinnen der generierten Editoren. Die Benutzer des VLEli-Systems werde ich im folgenden dagegen mit “Sprachentwerfer” bezeichnen.

4.1. Attributierte Grammatiken

In Abschnitt 3.2 habe ich einen Überblick über die Methode gegeben, mit der in dieser Arbeit Entwicklungsumgebungen für visuelle Sprachen generiert werden. Mit dieser Methode können visuelle Strukturediteoren erzeugt sowie Analysatoren und Transformatoren generiert werden. Diese Werkzeuge werden in einen festen

Rahmen eingebettet und so zu einer Entwicklungsumgebung integriert. Kernpunkt dieser Methode ist der Einsatz attributierter Grammatiken (AG) [Knuth 1968; Kastens 1991a, b].

In diesem Ansatz werden AGs benutzt, um aus einer (abstrakten) Repräsentation eines visuellen Programms dessen graphische Darstellung zu erzeugen bzw. zu aktualisieren. Hierbei muß die Anordnung entweder direkt berechnet oder ein Constraint-Netzwerk erzeugt werden, das gültige Anordnungen beschreibt. Weiterhin werden mit attributierten Grammatiken die zulässigen Editieroperationen beschrieben sowie die Konsistenz zwischen verschiedenen Teilen des visuellen Programms hergestellt. Das betrifft zum einen die Konsistenz verschiedener Sichten, zum anderen die Konsistenz einer Definition mit ihren Anwendungsstellen. Auf diese Anwendungsgebiete gehen die Abschnitte 4.3 bis 4.6 ein.

Im Anwendungsgebiet der Erzeugung von Editoren für visuelle Sprachen interessieren einige weitere Themenbereiche. Dies ist zum einen die Repräsentation visueller Programme durch Bäume. Zum anderen treten bei der Spezifikation von Editoren für visuelle Sprachen recht komplexe Berechnungen und weitreichende Abhängigkeiten auf, die sich mit den Methoden attributierter Grammatiken, so wie sie ursprünglich von Knuth [1968] eingeführt wurden, nur umständlich spezifizieren lassen. Hier werden vereinfachende Abstraktionstechniken benötigt. Weiterhin sollen Teile des Know-Hows zur Implementierung von Strukturediteren durch Module gekapselt werden. Dazu werden Techniken benötigt, mit denen Teile einer attributierten Grammatik in Module gekapselt und für verschiedene Sprachen wiederverwendet werden können. Schließlich gibt es unterschiedliche Methoden, mit denen Attributauswerter realisiert werden können. Diese beeinflussen auch die Verarbeitungsgeschwindigkeit der Attributauswerter und damit auch entscheidend die Effizienz des erzeugten Editors. Der letzte Abschnitt begründet den Einsatz des Generators Liga.

4.1.1. Strukturbäume

Attributierte Grammatiken können in dieser Arbeit nur dann verwendet werden, wenn die Programme (oder Sätze) visueller Sprachen sich durch Bäume repräsentieren lassen. Viele visuelle Sprachen enthalten Beziehungen, die nicht der hierarchischen Struktur der visuellen Sprache folgen. In einer attributierten Grammatik besteht oft die Notwendigkeit, Informationen entlang dieser Beziehungen zu transportieren. Es werden also zusätzliche Ausdrucksmittel benötigt, um diese Beziehungen zu repräsentieren.

Eine Parallele zu diesen Beziehungen tritt bei der Analyse textueller Sprachen auf, z.B. bei Sprachen mit Marken und Sprüngen. Im Bereich textueller Sprachen werden solche quer zur Hierarchie der Sprache verlaufenden Beziehungen

zunächst durch gemeinsam verwendete Namen gekennzeichnet. Während der Namensanalyse werden dann die Beziehungen identifiziert und Vorkehrungen getroffen, um in nachfolgenden Analysephasen Informationen entlang der Beziehungen transportieren zu können. Dies geschieht beispielsweise durch zusätzliche Kanten im Strukturbaum [Hedin 1999] oder durch Speicherung von Referenzen einer Definitionstabelle [Waite und Kadhim 1995]. Eine ähnliche Vorgehensweise ist auch im Bereich visueller Programme möglich.

Abschnitt 4.2 geht auf diese Thematik ein und untersucht die in visuellen Sprachen auftretenden abstrakten Strukturen und ihre Abbildung auf Strukturbäume. Das Ergebnis ist, daß sich viele visuelle Sprachen durch Strukturbäume repräsentieren lassen, wenn zusätzliche Ausdruckmittel für quer zur Hierarchie verlaufende Beziehungen vorhanden sind.

4.1.2. Abstraktionstechniken

Attributierte Grammatiken eignen sich gut, um Berechnungen zu spezifizieren, deren Abhängigkeiten sich über Sprachkonstrukte erstrecken, die im Strukturbaum benachbart sind. Müssen dagegen weitreichende Abhängigkeiten spezifiziert oder Daten zwischen weit auseinanderliegenden Baumteilen transportiert werden, neigen attributierte Grammatiken dazu, komplex, fehleranfällig und schwer wartbar zu werden, wenn die Spezifikationssprache keine weiteren Abstraktionstechniken zur Verfügung stellt. Die wichtigsten Techniken sind (auch) im Liga-System umgesetzt [Kastens und Waite 1994; Kastens 1991b]. Der Unterstützung dieser Techniken ist eine Reihe nützlicher Aspekte von VLEli zuzuschreiben. Deshalb sollen sie im folgenden kurz vorgestellt werden.

Hierzu ist in Abbildung 4.1 eine Attributberechnung abgebildet, die durch Anwendung verschiedener Abstraktionstechniken sukzessive vereinfacht wird. Diese Attributberechnungen transportieren in der Sprache der endlichen Automaten die Koordinaten der Zustände zu den in den Zuständen endenden Übergängen. Sie benutzen dazu Operationen, deren Seiteneffekte durch Attributberechnungen spezifiziert werden.

Eine erste dieser Abstraktionstechiken ist die Einführung von Sprachkonstrukten für oft auftretende Abhängigkeitsmuster in attributierten Grammatiken. Diese Konstrukte spezifizieren Zugriffe auf weit entfernte Attribute im Strukturbaum. Spezifikationen, die diese Konstrukte einsetzen, lassen sich leichter entwerfen, lesen und ändern. Das Beispiel in Abbildung 4.1(b) verwendet zwei dieser Konstrukte, CONSTITUENTS und INCLUDING, die in einem Vorläufer des Liga-Systems [Kastens 1976; Kastens u.a. 1982] erstmals eingeführt wurden. Ein weiteres, CHAIN, wurde von Lorho [1977] eingeführt. Durch die Anwendung dieser Konstrukte wird die attributierte Grammatik aus Abbildung 4.1(a) bereits erheblich verkürzt und vereinfacht, wie man in Abb. 4.1(b) sieht.

```
RULE: Place ::= COMPUTE
   Place.coords_computed =
      ORDER(ResetXPos(Place.Key, Place.x),
            ResetYPos(Place.key, Place.y));
END;

RULE: Places ::= COMPUTE
   Places.coords_computed = TRUE;
END;

RULE: Places ::= Places Place COMPUTE
   Places[1].coords_computed = Places[2].coords_computed
      DEPENDS_ON Place.coords_computed;
END;

RULE: Root ::= Places Transitions COMPUTE
   Transitions.coords_computed = Places.coords_computed;
END;

RULE: Transitions ::= Transitions Transition COMPUTE
   Transitions[2].coords_computed =
      Transitions[1].coords_computed;
   Transition.coords_computed =
      Transitions[1].coords_computed;
END;

RULE: Transition ::= COMPUTE
   Transition.startx = GetXPos(Transition.startKey, 0)
      DEPENDS_ON Transition.coords_computed;
   Transition.starty = GetYPos(Transition.startKey, 0)
      DEPENDS_ON Transition.coords_computed;
   Transition.endx = GetXPos(Transition.endKey, 0)
      DEPENDS_ON Transition.coords_computed;
   Transition.endy = GetYPos(Transition.endKey, 0)
      DEPENDS_ON Transition.coords_computed;
END;
```

(a) Keine Abstraktionstechniken

```
RULE: Place ::= COMPUTE
   Place.coords_computed =
      ORDER(ResetXPos(Place.Key, Place.x),
            ResetYPos(Place.key, Place.y));
END;

RULE: Root ::= Places Transitions COMPUTE
   Root.coords_computed =
      CONSTITUENTS Place.coords_computed;
END;

RULE: Transition ::= COMPUTE
   Transition.startx = GetXPos(Transition.startKey, 0)
      DEPENDS_ON INCLUDING Root.coords_computed;
   Transition.starty = GetYPos(Transition.startKey, 0)
      DEPENDS_ON INCLUDING Root.coords_computed;
   Transition.endx = GetXPos(Transition.endKey, 0)
      DEPENDS_ON INCLUDING Root.coords_computed;
   Transition.endy = GetYPos(Transition.endKey, 0)
      DEPENDS_ON INCLUDING Root.coords_computed;
END;
```

(b) Entfernte Attributzugriffe

```
SYMBOL Place COMPUTE
   SYNT.coords_computed =
      ORDER(ResetXPos(THIS.Key, THIS.x),
            ResetYPos(THIS.key, THIS.y));
END;

SYMBOL Root
   SYNT.coords_computed =
      CONSTITUENTS Place.coords_computed;
END;

SYMBOL Transition COMPUTE
   SYNT.startx = GetXPos(THIS.startKey, 0)
      DEPENDS_ON INCLUDING Root.coords_computed;
   SYNT.starty = GetYPos(THIS.startKey, 0)
      DEPENDS_ON INCLUDING Root.coords_computed;
   SYNT.endx = GetXPos(THIS.endKey, 0)
      DEPENDS_ON INCLUDING Root.coords_computed;
   SYNT.endy = GetYPos(THIS.endKey, 0)
      DEPENDS_ON INCLUDING Root.coords_computed;
END;
```

(c) Symbolattributierung

Bibliothek

```
CLASS SYMBOL LineEndpoint COMPUTE
   SYNT.coords_computed =
      ORDER(ResetXPos(THIS.Key, THIS.x),
            ResetYPos(THIS.key, THIS.y));
END;

CLASS SYMBOL LineRoot
   SYNT.coords_computed =
      CONSTITUENTS LineEndpoint.coords_computed;
END;

CLASS SYMBOL DirectLine COMPUTE
   SYNT.startx = GetXPos(THIS.startKey, 0)
      DEPENDS_ON INCLUDING LineRoot.coords_computed;
   SYNT.starty = GetYPos(THIS.startKey, 0)
      DEPENDS_ON INCLUDING LineRoot.coords_computed;
   SYNT.endx = GetXPos(THIS.endKey, 0)
      DEPENDS_ON INCLUDING LineRoot.coords_computed;
   SYNT.endy = GetYPos(THIS.endKey, 0)
      DEPENDS_ON INCLUDING LineRoot.coords_computed;
END;
```

Anwendung

```
SYMBOL Place INHERITS LineEndpoint END;
SYMBOL Transition INHERITS DirectLine END;
SYMBOL Root INHERITS LineRoot END;
```

(d) Vererbung

```
Root ::=   Places Transitions
Places ::= Places Place
Places ::=
Transitions ::= Transitions Transition
Transitions ::=
Place ::=
Transition ::=
```

(e) Zugrundeliegende Grammatik

Abbildung 4.1.: Abstraktionstechniken für attributierte Grammatiken

Die Technik der Symbolberechnungen erlaubt es, Berechnungen mit Baumsymbolen, statt mit Produktionen zu assoziieren. Dadurch wird eine attributierte Grammatik robuster gegenüber Änderungen der zugrundeliegenden Baumgrammatik. Weiterhin kann dadurch auch eine weitere Verkürzung der AG erzielt werden. Diese äußert sich darin, daß identische Berechnungen für ein Symbol, daß in mehreren Produktionen vorkommt, nicht mehrfach spezifiziert werden müssen.

Abbildung 4.1(c) enthält ein Beispiel für die Anwendung von Symbolberechnungen. Die Spezifikation ähnelt zwar strukturell stark ihrer Vorgängerversion aus Abbildung 4.1(b). Falls der Grammatik jedoch eine zusätzliche Produktion

```
Transition ::= TransitionLabel
```

für einen benannten Übergang im Automaten hinzugefügt würde, so wäre nun keinerlei Änderung der attributierten Grammatik mehr erforderlich. Die für das Grammatiksymbol `Transition` definierten Attributberechnungen würden auch im Kontext der neuen Produktion angestoßen.

Die dritte und für diese Arbeit wichtigste Technik zur Abstraktion von Attributberechnungen ist die *Vererbung*. Bei Liga werden hierzu Symbolattributierungen für Berechnungsrollen definiert. Die Namen der Berechnungsrollen kommen nicht in der zugrundeliegenden Grammatik vor. Voneinander abhängende Berechnungen können so vollständig unabhängig von einer konkreten (visuellen) Sprache definiert und in einer Bibliothek ausgelagert werden. Abbildung 4.1(d) zeigt oben ein Beispiel für derartige Berechnungen.

Die so definierten Berechnungsrollen werden angewendet, indem sie den Grammatiksymbolen zugeordnet werden. Zum einen erbt dadurch das Grammatiksymbol die Berechnungen der Rolle. Diese Berechnungen können bei Bedarf auch überschrieben werden, wodurch sich Defaultwerte gut umsetzen lassen. Zum anderen werden die verschiedenen CONSTITUENTS- und INCLUDING-Konstrukte dadurch passend ausgeweitet, wodurch die Abhängigkeiten zwischen den Attributberechnungen der Rollen auch nach der Zuordnung an Grammatiksymbole erhalten bleiben.

Abbildung 4.1(d) enthält unten die Anwendung der im oberen Teil definierten Berechnungsrollen. Lediglich den unteren Teil müßte ein Sprachentwerfer noch selbst schreiben, um Berechnungen zu spezifizieren, für die ohne Abstraktionstechniken sehr viel mehr Aufwand erforderlich wäre. Im weiteren ist wichtig, daß auch mehrere Berechnungsrollen an ein Grammatiksymbol vererbt werden können, wobei sich die Attributberechnungen ergänzen (Mehrfachvererbung). Berechnungen zur Lösung verschiedener Probleme können so voneinander getrennt in verschiedenen Modulen realisiert und für eine konkrete Sprache miteinander kombiniert werden.

Die Vererbung von Symbolberechnungen wird vom Liga-System seit 1992 unterstützt. Eine wesentliche Eigenschaft ist, daß die Berechnungsrollen unabhängig von der Grammatik definiert werden können und Mehrfachvererbung leicht

zu realisieren ist. Vererbung wird zwar auch in anderen Spezifikationssprachen für attributierte Grammatiken unterstützt [Grosch 1989; Hedin 1989; Koskimies 1991], jedoch beziehen sich die vererbten Attributberechnungen auf Produktionen statt auf Symbole. Dies erhöht die Abhängigkeit von einer Sprache und verhindert Mehrfachvererbung.

Wie angedeutet, eignet sich das Konzept der Vererbung der Symbolattributierung, um Bibliotheken wiederverwendbarer Berechnungsrollen zur Verfügung zu stellen. Das Eli-System, in das auch Liga integriert ist, bietet eine solche Bibliothek an. Hierzu können die Spezifikationen, die einen Satz von Berechnungsrollen definieren, zusammen mit Spezifikationen für andere Werkzeuge und eventuell benötigten weiteren Codefragmenten zu einem Spezifikationsmodul zusammengefaßt werden.

Die Spezifikationsmodule des Eli-Systems enthalten Berechnungsrollen, um verschiedene, oft auftretende Aufgaben im Bereich der Analyse textueller Sprachen zu lösen, z.B. die Namens- und Typanalyse. Die Lösung derartiger Probleme wird durch Spezifikationsmodule erheblich vereinfacht. Dies ermöglicht es auch Nichtspezialisten, Analysewerkzeuge für neue Sprachen zu erstellen.

Durch den Einsatz von Spezifikationsmodulen wird im Eli-System weiterhin eine gewisse Flexibilität von Spezifikationen erreicht. Für Probleme, die auf unterschiedliche Weise gelöst werden können, werden dazu Spezifikationsmodule bereitgestellt, die dieselbe Schnittstelle aufweisen, d.h. die Berechnungsrollen mit gleichen Namen, gleicher Vor- und Nachbedingung zur Verfügung stellen. Ein Sprachentwickler kann dann ein Spezifikationsmodul schnell durch ein anderes austauschen, ohne darauf aufbauende Attributierungen ändern zu müssen.

Die Anwendung von Spezifikationsmodulen ist auch für die Implementierung visueller Sprachen vielversprechend. Die grundlegende Spezifikationstechnik attributierter Grammatiken liefert eine Basis, mit der ein weiter Bereich visueller Sprachen implementiert werden kann. Durch die verschiedenen, aufeinander aufbauenden Abstraktionstechniken wird die Erzeugung attributierter Grammatiken so weit vereinfacht, daß auch Nichtspezialisten die zur Implementierung visueller Sprachen notwendigen Berechnungen spezifizieren können. Entwurfsalternativen für visuelle Sprachen können dabei so umgesetzt werden, daß ein Sprachentwerfer eine Lösung später flexibel durch eine Alternative ersetzen kann. Hierbei ist zu berücksichtigen, daß es zwar einige Systeme gibt, die Vererbung unterstützen, aber lediglich das Liga-System die Vererbung von Symbolattributierung unterstützt. Gerade diese Technik ist aber wesentlich für die Erzeugung der Spezifikationsmodule.

4.1.3. Attributauswertung und Effizienz

Ein Attributauswerter ist ein Programm, das die Berechnungen einer attributierten Grammatik für beliebige, auf die Grammatik passende Strukturbäume ausführt. Der Auswerter wird in der Regel durch einen Generator aus einer attributierten Grammatik erzeugt.

Man unterscheidet *inkrementelle* und *nichtinkrementelle* Attributauswerter. Das Einsatzgebiet inkrementeller Attributauswerter ist die wiederholte Auswertung attributierter Grammatiken für Bäume, die zwischen zwei Läufen des Attributauswerters geringfügig geändert werden. Anstatt wie bei nichtinkrementellen Attributauswertern alle Attributberechnungen bei jedem Lauf erneut auszuführen, versucht man bei inkrementellen Attributauswertern, Attributwerte für nicht geänderte Programmteile möglichst wiederzuverwenden. Das Ziel ist einerseits eine Beschleunigung der Auswertung, andererseits die einfachere Nutzung anderer inkrementeller Werkzeuge und Algorithmen.[1]

Um diesen Effekt zu erreichen, muß in inkrementellen Attributauswertern im Vergleich zu nichtinkrementellen ein zusätzlicher Aufwand investiert werden. Zum einen müssen alle Attribute über das Ende eines Attributauswerterlaufs hinaus gespeichert werden, was den Einsatz wichtiger Verfahren zur Reduktion des für Attributwerte erforderlichen Speicherplatzes verhindert. Im weiteren müssen bei Programmänderungen die davon betroffenen Attributwerte ausfindig gemacht und für eine Neuberechnung vorgemerkt werden. Hierfür ist gegenüber nichtinkrementellen Attributauswertern ein gewisser zusätzlicher Verwaltungsaufwand, sowohl bei der Speicherung des Strukturbaums als auch bei der Durchführung der Attributberechnungen und der Änderung des Strukturbaums erforderlich.

Weiterhin geht man bei inkrementellen Attributauswertern von einer funktionalen, seiteneffekt-freien Semantik der attributierten Grammatik aus. Deshalb ist die Technik, die in Abbildung 4.1 zum Transport von Informationen zwischen weit auseinanderliegenden Teilen des Strukturbaums benutzt wird, bei inkrementellen Attributauswertern nicht anwendbar.

In verschiedenen Generatoren für inkrementelle Attributauswerter werden unterschiedliche Alternativen verwendet. Beim Synthesizer Generator werden etwa Werte entlang der Hierarchie des Baumes transportiert, indem sie zunächst in einem Attribut zusammengefaßt und an den Anwendungsstellen aus diesem komplexwertigen Attribut extrahiert werden. Diese Technik erfordert einen erhöhten Implementierungsaufwand und schmälert durch zusätzliche Abhängigkeiten den Vorteil der inkrementellen Attributauswertung. Bei APPLAB [Hedin 1992, 1994, 1999] können in Attributen Referenzen auf Knoten im Strukturbaum gespeichert und Informationen entlang dieser Referenzen transportiert werden. Bei diesem Ansatz treten jedoch komplexe Attributabhängigkeiten auf, die sich

[1] Auf den zweiten Aspekt gehe ich im Zusammenhang mit der Anbindung des Constraint-Solvers in Abschnitt 4.4.2.4 ein.

nicht mehr statisch auf Zyklenfreiheit prüfen lassen. Deshalb wird bei APPLAB die attributierte Grammatik durch einen universellen Algorithmus zur Attributauswertung interpretiert.

In anderen Ansätzen wird die Benutzung der Referenzen eingeschränkt, beispielsweise indem Zugriffe auf Attribute der referenzierten Kontexte erst nach der Abarbeitung des Attributauswerters erlaubt werden [Reps und Teitelbaum 1989a; Augusteijn 1990]. Die Referenzen sind dann zwar immer noch nützlich, um Konsistenzprüfungen durchzuführen. Für ihren Einsatz in dieser Arbeit liefern sie dadurch aber keine Vorzüge.

Der in dieser Arbeit benutzte Generator Liga generiert nichtinkrementelle Attributauswerter. Für die Anwendung im Bereich visueller Sprachen stellt sich die Frage, ob eine Steigerung der Arbeitsgeschwindigkeit der generierten Entwicklungsumgebungen für diese Arbeit zu erwarten ist, wenn inkrementelle Attributauswerter statt nichtinkrementeller Attributauswerter verwendet werden. Zwei Aspekte lassen dies fragwürdig erscheinen: Zum einen muß für inkrementelle Attributauswerter einiges an zusätzlichem Aufwand investiert werden, zum anderen sind die vom Liga-System erzeugten Attributauswerter recht effizient implementiert [Prott 1996; Sloane 1995].

Auf weitere Aspekte, die zur Auswahl des Liga-Systems geführt haben, geht der folgende Abschnitt ein.

4.1.4. Auswahl eines Generators

Um attributierte Grammatiken für die Implementierung visueller Sprachen benutzen zu können, wird ein Generator benötigt, der Attributauswerter zu attributierten Grammatiken erzeugen kann. In dieser Arbeit soll dafür das Liga-System verwendet werden.

Der wichtigste Grund für diese Entscheidung sind die Techniken, die das Liga-System zur Definition wiederverwendbarer Module unterstützt. Wie das Eli-System [Eli 2000] zeigt, lassen sich damit Spezifikationsmodule erzeugen, die unabhängig von einer konkreten Sprache sind und die einfach und flexibel angewendet werden können. Damit sind sogar Nichtspezialisten in der Lage, das in den Modulen gekapselte Know-How anzuwenden. Dies haben unter anderem auch Seminare gezeigt, in denen Anwender aus der Industrie bereits nach geringer Einarbeitungszeit Übersetzer für Spezialsprachen implementiert haben.

Ein weiterer Vorteil des Generators Liga ist seine Einbindung in das Eli-System. Mit attributierten Grammatiken soll auch die Analyse und Weiterverarbeitung visueller Programme spezifiziert werden. Das Eli-System stellt hierfür unterstützende Werkzeuge und Bibliotheken bereit, die beispielsweise Eigenschaften von Programmobjekten in einer Definitionstabelle speichern oder die die Typanalyse unterstützen. Die Produkte dieser Werkzeuge und die bereitgestellten Bibliothe-

ken sind auf die generierten Attributauswerter abgestimmt, so daß sie komfortabel benutzt werden können.

Die Auswahl eines Generators wurde außerdem durch die Tatsache beeinflußt, daß in unserer Arbeitsgruppe langjährige Erfahrungen mit dem Liga-System existieren. Für dessen Anpassung an das neue Einsatzgebiet stand so deutlich mehr Know-How als bei anderen Generatoren zur Verfügung. Anpassungen waren beim Liga-System erforderlich, um etwa das generierte Modul zur Speicherung des Strukturbaums zu ergänzen. Ähnliche Änderungen wären auch bei anderen Generatoren erforderlich gewesen, weil für die Implementierung visueller Sprachen Techniken zur Repräsentation nichthierarchischer Beziehungen benötigt werden, siehe dazu auch den folgenden Abschnitt.

Schließlich ist auch die Effizienz der generierten Attributauswerter ein wichtiges Entscheidungskriterium. Wie ich im vorigen Abschnitt ausgeführt habe, kann nicht ausgeschlossen werden, daß sich bei der Verarbeitung großer visueller Programme in manchen Fällen ein Effizienzgewinn einstellt, wenn das Liga-System durch einen Generator für inkrementelle Attributauswerter ausgetauscht wird. Dies wurde aber nicht in Erwägung gezogen, da das Modulkonzept von Liga und die Anbindung an das Eli-System dringend benötigt werden. Außerdem müßte dann auch darauf verzichtet werden, Seiteneffekte in Attributberechnungen zu verwenden.

4.2. Repräsentation visueller Programme durch Strukturbäume

Die Programmrepräsentation ist eine strukturelle Abstraktion des (visuellen) Programms. Im VLEli-System speichert sie alle wesentlichen Informationen über das Programm. Aus ihr wird einerseits für den jeweils nächsten Editierschritt die graphische Darstellung eines visuellen Programms erzeugt. Sie dient im weiteren als Basis für die Analyse und Weiterverarbeitung der visuellen Programme. Die Programmrepräsentation des VLEli-Systems erfüllt also ähnliche Aufgaben wie der *abstract syntax graph* des PROGRESS-Systems [Rekers und Schürr 1996], enthält aber zusätzlich einige Angaben, die zur Erzeugung der graphischen Darstellung erforderlich sind, z.B. Koordinaten- oder Größenangaben von Sprachkonstrukten mit Layoutfreiheiten.

In diesem Abschnitt zeige ich, daß Programme visueller Sprachen repräsentiert werden können, indem man Strukturbäume um Definitionstabellen ergänzt. Strukturbäume und Definitionstabellen sind Standardmethoden zur Implementierung textueller Sprachen. Die Definitionstabellen repräsentieren dabei allgemeine Beziehungen zwischen den Konstrukten eines Programms, die durch die Bäume nicht direkt dargestellt werden können. Bei textuellen Sprachen werden

die Definitionstabellen im Rahmen der Namensanalyse aus den Bezeichnern im Programm aufgebaut, also *berechnet*.

Für die Implementierung visueller Sprachen kann man visuelle Programme durch einen Strukturbaum *und* eine Definitionstabelle repräsentieren. Die Definitionstabelle wird also *nicht* aus den Informationen des Strukturbaums berechnet, sondern zusammen mit dem Strukturbaum als abstrakte, persistente Darstellung des visuellen Programms verwendet. Aus dieser Festlegung der Programmrepräsentation resultiert ein bedeutender Vorteil: Visuelle Programme können mit den gleichen Werkzeugen verarbeitet werden, mit denen auch textuelle Programme weiterverarbeitet werden. Das schließt Attributauswerter ein, zu deren Generierung somit auch eingeführte, erprobte Werkzeuge wie das Liga-System verwendet werden können. Weiterhin können durch die Kombination aus Strukturbaum und Definitionstabelle alle Programmstrukturen dargestellt werden, die auch durch Ansätze, die Graphen dafür verwenden, repräsentiert werden können.

Ein wesentlicher Beitrag dieser Arbeit ist in diesem Zusammenhang die Anwendung der Definitionstabelle in einem neuen Gebiet, nämlich dem Gebiet der Implementierung visueller Sprachen. Dadurch können Generatoren für Attributauswerter zur Implementierung visueller Sprachen verwendet werden, ohne wie beim LOGGIE-System [Backlund u.a. 1990] das Kalkül der attributierten Grammatiken zu ändern oder wie bei GIGAS [Chabrier u.a. 1988; Lextrait und Zarli 1990] auf die Repräsentation nichthierarchischer Beziehungen zu verzichten.

Im folgenden beschreibe ich zunächst die Kombination von Strukturbaum und Definitionstabelle genauer. Dabei nenne ich auch Gemeinsamkeiten und Unterschiede zur Implementierung textueller Sprachen. Im Anschluß gehe ich auf die Spezifikation der Programmrepräsentation für visuelle Sprachen ein, die ich in dieser Arbeit verwende.

Abschnitt 4.2.3 vergleicht die durch diesen Ansatz darstellbaren Informationsstrukturen mit anderen Arbeiten. Der letzte Abschnitt beschreibt die Spezifikation der Repräsentation aus der praktischen Perspektive.

4.2.1. Kombination aus Strukturbaum und Definitionstabelle

Ein wesentlicher Unterschied zwischen visuellen und textuellen Sprachen ist, daß bei visuellen Sprachen sehr viel mehr Möglichkeiten existieren, um die Beziehungen in einem Programm zu visualisieren. Benutzt man hierfür etwa Linien, so muß in einem Strukturediteor unter anderem auch die Editieroperation ‘Einfügen einer Linie zwischen zwei Programmkonstrukten’ realisiert werden. Hierzu müssen, wie ich in Abschnitt 2.3.3.1 motiviert habe, alle Beziehungen eines visuellen Programms direkt in der Programmrepräsentation gespeichert werden.

Mit Hilfe von Strukturbäumen lassen sich jedoch zunächst lediglich hierarchische Beziehungen wie “besteht-aus” direkt darstellen. Zur Repräsentation der an-

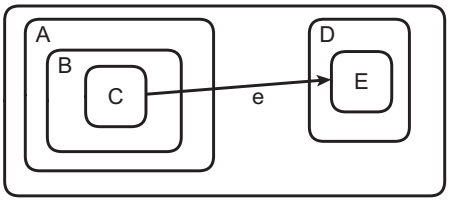


Abbildung 4.2.: Verschiedenartige Beziehungen in Zustandsdiagrammen

deren, im folgenden *nichthierarchisch* genannten Arten von Beziehungen wird in dieser Arbeit eine Definitionstabelle verwendet. In diesem Abschnitt liefere ich anhand der Zustandsdiagramme [Harel 1988] ein Beispiel für die Repräsentation der visuellen Programme und insbesondere der nichthierarchischen Beziehungen.

Zustandsdiagramme bestehen aus Zuständen und Übergängen. Zwischen diesen gibt es verschiedenartige Beziehungen. Zum ersten können die Zustände einander enthalten. Diese Beziehung ist hierarchisch und wird durch Schachtelung der graphischen Abbildungen der Zustände visualisiert. Mit Hilfe von Bäumen können diese Beziehungen in natürlicher Weise modelliert werden. Zum zweiten verbinden die Übergänge jeweils zwei Zustände, definieren also die Beziehungen “Übergang-von” und “Übergang-nach”. Diese Beziehungen sind nichthierarchisch. Im visuellen Programm werden sie dargestellt, indem der Übergangspfeil so gezeichnet wird, daß seine Enden die in Beziehung gesetzten Zustände berühren, siehe Abbildung 4.2.

Die in dieser Arbeit benutzte Programmrepräsentation basiert auf einer wichtigen Beobachtung: auch Sprachen mit textueller Notation enthalten nichthierarchische Beziehungen. Diese Beziehungen werden in textuellen Programmen durch gemeinsame Benutzung gleicher Namen kenntlich gemacht. Hierbei stehen die Namen für Programmobjekte, die der Anwender einer Sprache an vielen Stellen referenzieren kann. So kann etwa der Name für eine Variable in einer Definition und in beliebig vielen Ausdrücken verwendet werden. Abbildung 4.3 zeigt als Beispiel eine (erfundene) textuelle Notation für das Zustandsdiagramm aus Abbildung 4.2. Die Zustände sind hier Programmobjekte, die in Zustandsdeklarationen eingeführt und in beliebig vielen Übergängen benutzt werden können. Die Zustände werden im Programm durch Namen bezeichnet.

Bei der Analyse textueller Sprachen werden im Rahmen der Namensanalyse die Programmobjekte identifiziert. Dabei wird jeder Stelle im Programm, an der ein Name auftritt, ein Eintrag in eine Definitionstabelle [Kastens und Waite 1991; Waite und Kadhim 1995] zugeordnet. Zwei Stellen, die dasselbe Programmobjekt referenzieren, verweisen dabei auf denselben Definitionstabelleneintrag. Abbildung 4.4 skizziert den Strukturbaum, der sich nach der Namensanalyse für das

Programm aus Abbildung 4.3 einstellen könnte. Den Kontexten, in denen Namen auftreten, sind die jeweils referenzierten Einträge in der Definitionstabelle zugeordnet.

Den Einträgen der Definitionstabelle können im weiteren Eigenschaften zugeordnet werden. In Abbildung 4.4 ist dies etwa der Name des Zustands und einige weitere Eigenschaften. So können von den auf die Namensanalyse folgenden Übersetzerphasen Informationen zwischen den an einer nichthierarchischen Beziehung beteiligten Programmstellen ausgetauscht werden. Dies kann etwa der Typ einer Variable sein, der von der Definitionsstelle zu den Anwendungsstellen transportiert wird.

Um visuelle Programme zu speichern, wird in dieser Arbeit die Repräsentation gewählt, die sich nach Durchführung der Namensanalyse für ein äquivalentes Programm mit textueller Notation einstellen würde. Damit ist also die in Abbildung 4.4 skizzierte Datenstruktur gleichzeitig die Programmrepräsentation des Zustandsdiagramms aus Abbildung 4.2, wenn man von den Eigenschaften, die in der Definitionstabelle zur Erläuterung angegeben sind, absieht. Wesentlich ist dabei, daß die durch Pfeile skizzierten Referenzen auf Einträge der Definitionstabelle hier *nicht* berechnet werden, sondern als Datenstruktur für die Repräsentation der Programme benutzt werden. Dadurch können die nichthierarchischen Beziehungen, für die ein Programmobjekt steht, in einfacher Weise editiert werden. Dies zeigt beispielsweise Abbildung 4.20 auf Seite 148 für Petri-Netze.

In derselben Weise, in der bei der Analyse textueller Programme Informatio-

```
Statechart {
  XORSuperState A {
    XORSuperState B {
       State C;
    }
  }

  XORSuperState D {
    State E;
  }

  Transition e: C -> E;
}
```

Abbildung 4.3.: Textuelle Notation des Zustandsdiagramms aus Abb. 4.2

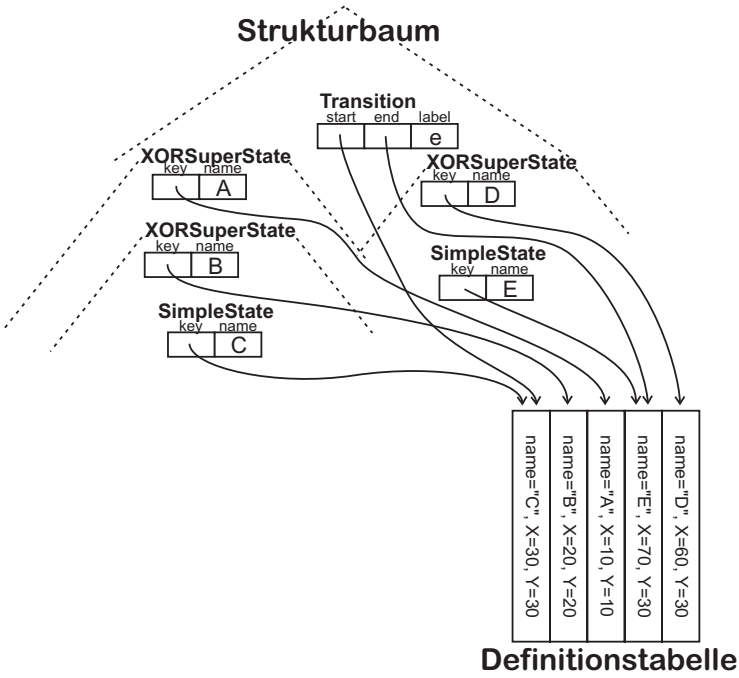


Abbildung 4.4.: Repräsentation des Zustandsdiagramms aus Abb. 4.2 und Abb. 4.3

nen entlang nichthierarchischer Beziehungen transportiert werden können, können auch bei der Verarbeitung visueller Programme Daten zwischen den Knoten des Strukturbaums ausgetauscht werden. Abbildung 4.4 skizziert dies durch die Eigenschaften der Definitionstabelleneinträge: Die Koordinaten der graphischen Abbildungen der Zustände werden in der Definitionstabelle gespeichert, so daß sie für das Layout der Übergänge benutzt werden können.

Diese Festlegung der Programmrepräsentation hat weitere Vorzüge. Zum einen können (wie beschrieben) in Attributberechnungen Informationen entlang nichthierarchischer Beziehungen transportiert werden. Dazu braucht das Kalkül der attributierten Grammatiken nicht geändert zu werden, wie dies für den Generator LOGGIE [Backlund u.a. 1990] erforderlich war. Zum anderen gibt es Sprachen, für die eine visuelle und eine textuelle Notation existiert, etwa SDL [Belina u.a. 1991] oder HeNCE [Beguelin und Dongarra 1991]. Für solche Sprachen

können Analyseaufgaben, die auf den Ergebnissen der Namensanalyse aufbauen, gemeinsam für beide Notationen spezifiziert werden. Dies kann den Spezifikationsaufwand solcher Sprachen erheblich reduzieren, wie ich an einem Beispiel in Abschnitt 4.6.1.1 zeige.

4.2.2. Spezifikation der Repräsentation

Die Repräsentation für ein visuelles Programm besteht aus zwei Teilen, aus einem Strukturbaum und aus Referenzen auf Einträge in die Definitionstabelle. Um den Strukturbaum zu spezifizieren, können kontextfreie Grammatiken verwendet werden. Abbildung 4.5 enthält oben die kontextfreie Grammatik, die den in Abbildung 4.2 sichtbaren Teil der Sprache der Zustandsdiagramme spezifiziert.

Die Regeln dieser kontextfreien Grammatik definieren gleichzeitig Editieroperationen für die hierarchischen Anteile der Sprache. Die beiden Alternativen für das Symbol `State` können etwa überall dort angewendet werden, in denen das Symbol `State` auf der rechten Seite einer Produktion vorkommt. Eine Regel wird dabei angewendet, indem sie entweder in Sequenzen (spezifiziert durch das nachgestellte *-Symbol) eingefügt wird oder indem sie die bisherige Ableitung für das Symbol ersetzt.

Eine Randbedingung an die kontextfreie Grammatik, die zur Definition einer visuellen Sprache verwendet wird, entsteht dabei durch die Forderung, daß der Strukturbaum für ein editiertes visuelles Programm nach jeder Editieroperation wieder korrekt gemäß der kontextfreien Grammatik für die Sprache ist. Diese Forderung resultiert daraus, daß nach jedem Editierschritt Attributauswerter angewendet werden, etwa um die graphische Darstellung für das Programm zu berechnen. Um diese Forderung zu erfüllen, müssen die rechten Seiten angewendeter Produktionen zu vollständigen Teilbäumen ergänzt werden können. Dazu muß bei Symbolen, für die Produktionsalternativen existieren, eine “Default-Produktion” angegeben werden. Auf diese Angabe kann aber verzichtet werden, wenn wie bei den Produktionen zu `State` das Symbol der linken Seite in anderen Produktionen ausschließlich in Sequenzen benutzt wird.

Die Einträge in die Definitionstabelle können definiert werden, indem den Symbolen der kontextfreien Grammatik Attribute zugeordnet werden. Diese Attribute haben zunächst nichts mit den Attributen zu tun, die durch attributierte Grammatiken berechnet werden, können aber in den Berechnungen einer attributierten Grammatik benutzt werden. Sie gehören zur persistenten Repräsentation eines visuellen Programms und werden im folgenden deshalb *persistente Attribute* genannt.

Die Spezifikation für ein persistentes Attribut enthält die Angabe eines Symbolnamens, einen Attributnamen und einen Attributtyp. Durch persistente Attribute des Typs `DefTableKey` wird hierbei ein Eintrag in die Definitionstabelle

spezifiziert. Attribute anderen Typs speichern andere Informationen über ein visuelles Programm, etwa den Namen eines Zustands oder die Beschriftung eines Übergangs. Abbildung 4.5 enthält unten die zur Spezifikation der Sprache der Zustandsdiagramme erforderlichen persistenten Attribute. Die Attribute sind in Abbildung 4.4 durch die beschrifteten Rechtecke im Strukturbaum visualisiert.

Durch die Attribute, die Einträge in die Definitionstabelle repräsentieren, werden implizit auch Editieroperationen für die nichthierarchischen Beziehungen eines Programms definiert. Diese Editieroperationen erlauben aber jegliche Zuordnung zwischen den persistenten Attributen vom Typ `DefTableKey` und Einträgen der Definitionstabelle. Die Spezifikation der `DefTableKey`-Attribute definiert also lediglich die Programmrepräsentation, aber nicht die *legalen* Editieroperationen für eine visuelle Sprache.

Die durch einen Benutzer im jeweils nächsten Schritt anwendbaren Editieroperationen, die nichthierarchische Beziehungen betreffen, werden deshalb, zusammen mit der graphischen Darstellung, durch einen Attributauswerter berechnet. Dadurch können Informationen über das visuelle Programm ermittelt und für die Berechnung der Editieroperationen verwendet werden. So lassen sich nicht nur strukturelle, sondern auch komplexere Informationen über das visuelle Programm für deren Definition benutzen. Weiterhin können auch mehrere einzelne Transformationen zusammengefaßt und so komfortablere Editieroperationen realisiert werden. Auf die Spezifikation der Editieroperationen für ein visuelles Programm gehe ich detaillierter in Abschnitt 4.5 ein.

```
/* Grammatik */
Statechart ::= State* Transition*.
Transition ::= .
State ::= SimpleState.
State ::= XORSuperState.
SimpleState ::= .
XORSuperState ::= State*.

/* Attribute */
SimpleState:   key:        DefTableKey;
XORSuperState: key:        DefTableKey;
Transition:    start, end: DefTableKey;
Transition:    label:      String;
SimpleState:   name:       String;
XORSuperState: name:       String;
```

Abbildung 4.5.: Spezifikation der Repräsentation der Zustandsdiagramme (vereinfacht)

4.2.3. Darstellbare Strukturen

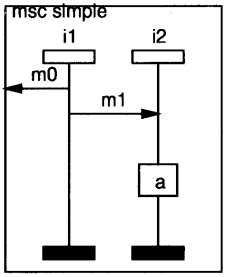
Mit der Erweiterung des Strukturbaums um eine Definitionstabelle können zusätzlich zu hierarchischen auch andere Beziehungen zwischen den Programmkonstrukten repräsentiert werden. Die so darstellbaren Strukturen sind dieselben, die sich auch durch Graphen darstellen lassen. In diesem Ansatz wird lediglich zusätzlich zwischen Kanten, die eine Hierarchie bilden und anderen Kanten unterschieden.

Die Strukturbäume sind also, was die durch sie darstellbaren Informationsstrukturen betrifft, mit Ansätzen, die Graphen dafür verwenden, vergleichbar. Da auch mehrere Kanten zwischen je zwei Programmkonstrukten existieren können und sich verschiedene Knoten- und Kantentypen unterscheiden lassen, können beispielsweise auch die Informationsstrukturen dargestellt werden, die Erwig [1997, 1998] durch Multi-Graphen repräsentiert. Auch die Graphen mit Kantenbeschriftung, die in vielen Ansätzen auftreten [Marriott u.a. 1998, S. 20 oben, Göttler 1986, 1989; Arefi u.a. 1990; Rekers und Schürr 1996; Andries u.a. 1998] sowie die Hypergraphen von Minas und Viehstaedt [1995]; Minas [1997] sind mit diesem Ansatz vergleichbar.

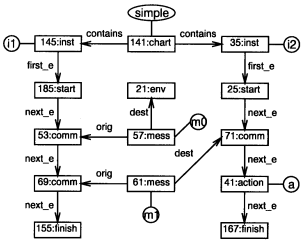
Dies soll an einem Beispiel verdeutlicht werden. Andries u.a. [1998] schlagen Graphen mit Kantenbeschriftung vor, um die abstrakte Struktur visueller Programme zu repräsentieren. Sie spezifizieren diese Graphen durch eine Graphgrammatik und führen dies am Beispiel der Message-Sequence-Charts vor. Abbildung 4.6 zeigt das visuelle Beispielprogramm und dessen abstrakte Struktur. Man erkennt bereits am Graphen in Abbildung 4.6(b), daß Message-Sequence-Charts zahlreiche Beziehungen enthalten, die sich auch durch Hierarchien ausdrücken lassen. Lediglich die durch Pfeile dargestellten Nachrichten benötigen zusätzliche Ausdrucksmittel.

Für die Teilmenge der Sprache der Message-Sequence-Charts, die das Beispiel verwendet, ergibt sich die in Abbildung 4.6(c) gezeigte Spezifikation. Die Beziehungen `contains`, `first_e` und `next_e` aus Abb. 4.6(b) sind hierarchisch und wurden durch Grammatikproduktionen ausgedrückt. Zusätzliche hierarchische Beziehungen wurden eingeführt, um die Nachrichten (`mess`-Knoten) und die Schnittstellen mit der Umgebung (`env`-Knoten) unterhalb der Wurzel aufzulisten. Für die Beziehungen `orig` und `dest` wurden persistente Attribute eingeführt, die auf Einträge der Definitionstabelle für die durch sie verbindbaren Programmkonstrukte verweisen, das sind die `comm` und `env`-Knoten.

Um keinen falschen Eindruck zu erwecken, sei an dieser Stelle darauf hingewiesen, daß sich dieser Vergleich lediglich auf die darstellbaren Informationsstrukturen bezieht. Die Graphgrammatiken, die Andries u.a. [1998] verwenden, spezifizieren nicht nur die Informationsstrukturen, sondern auch die Editieroperationen für Strukturediteren. Die Editieroperationen werden in diesem Ansatz zwar auch weitgehend durch die angegebene Spezifikation festgelegt. Um einen komforta-



(a) Ein einfaches Message-Sequence-Chart nach Andries u.a. [1998]



(b) Abstrakter Strukturgraph zu (a) nach Andries u.a. [1998]

```
/* Grammatik */

Chart ::= Instance* Mess* Env*.
Instance ::= Start Code* Finish.
Code ::= Comm.
Code ::= Action.

Start ::= .
Finish ::= .
Mess ::= .
Comm ::= .
Action ::= .
Env ::= .

/* Attribute */

Env: key: DefTableKey;
Comm: key: DefTableKey;
Mess: orig, dest: DefTableKey;
Mess: label: String;
Instance: name: String;
Action: label: String;
```

(c) Kontextfreie Grammatik und persistente Attribute für Message-Sequence-Charts

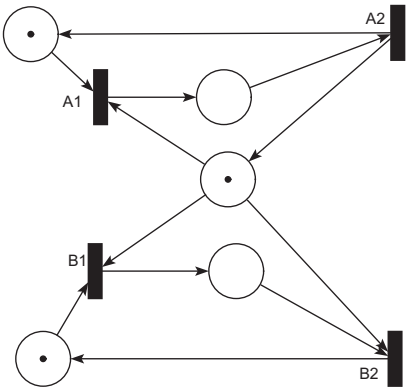
Abbildung 4.6.: Spezifikation der Repräsentation der Message-Sequence-Charts

blen Strukturediteur zu implementieren sind aber zusätzliche Angaben erforderlich.

4.2.4. Beispiel

Als durchgängiges Beispiel für die folgenden Abschnitte dieses Kapitels habe ich die Sprache der Petri-Netze [Petri 1962] ausgewählt. Petri-Netze sind allgemein bekannt und haben auch praktische Relevanz, beispielsweise, um das Verhalten nebenläufiger Systeme zu modellieren [Graf 1987; Nædele und Janneck 1998; Usher und Jackson 1998; Giese u.a. 1999]. Die Sprache der Petri-Netze hat den Vorteil, daß sie klein genug ist, um im Rahmen von Beispielen vollständig behandelt zu werden. Außerdem weist sie einige für diese Arbeit interessante Aspekte auf. Wo dies für das Verständnis förderlich ist, werden natürlich auch Beispiele aus anderen Sprachen verwendet.

Abbildung 4.7(a) zeigt ein Petri-Netz. Petri-Netze sind gerichtete bipartite Graphen mit zwei Knotentypen. Die Stellen (engl. *Places*) werden durch weiße Kreise, die Transitionen (engl. *Transition*) durch schwarze Rechtecke dargestellt. Den Stellen sind Marken (engl. *Tokens*) zugeordnet, die durch schwarze Punkte in den Kreisen dargestellt werden. Die Kanten des Graphen verbinden Stellen und Transitionen miteinander.



(a) Beispiel

```
/* Kontextfreie Grammatik */

RULE pRoot:     ROOT ::= PetriNet END;
RULE pNet:      PetriNet ::= Elem* Connection* END;
RULE pStateEl:  Elem ::= State END;
RULE pTransEl:  Elem ::= Transition END;
RULE pCommEl:   Elem ::= Comment END;
RULE pState:    State ::= END;
RULE pTrans:    Transition ::= END;
RULE pComm:     Comment ::= END;
RULE pConn:     Connection ::= END;

/* persistente Attribute */

/* Positionen der Zustände und Übergänge */
SYMBOL Elem: POS: VLPointArray;
SYMBOL Connection: POS: VLPointArray;

/* Start und Ende eines Übergangs */
SYMBOL State: persKey: DEFTABLEKEY;
SYMBOL Transition: persKey: DEFTABLEKEY;
SYMBOL Connection: persFrom  : DEFTABLEKEY,
                   persTo    : DEFTABLEKEY;

/* Anzahl Places eines Zustands */
SYMBOL State: numPlaces: VLInt EDITWITH ENTRY;

/* Text eines Kommentars */
SYMBOL Comment: text: VLString;
```

(b) Spezifikation der Repräsentation

Abbildung 4.7.: Petri-Netze und ihre Repräsentation

Abbildung 4.7(b) zeigt die kontextfreie Grammatik und die persistenten Attribute, die die abstrakte Struktur der Sprache der Petri-Netze beschreiben. Neben den Konstrukten für Stellen, Transitionen und Verbindungen sind noch Kommentare und Positionsangaben vorgesehen. Die Kommentare können in der graphischen Darstellung zu deren Erläuterung verwendet werden. Anders als bei Übersetzern für textuelle Sprachen müssen sie bei der Erzeugung der Darstellung berücksichtigt werden und finden deshalb Eingang in die Spezifikation der Programmrepräsentation.

Die Positionsangaben, also die persistenten Attribute, die den Namen POS tragen, speichern die Positionen verschiedener Sprachkonstrukte in der graphischen Darstellung. Sie werden benötigt, weil die Positionen der Stellen, Transitionen und Kommentare frei wählbar sein sollen. Die Positionen werden beim Einfügen und Verschieben eines Sprachkonstrukts durch den Maus-Cursor spezifiziert und müssen gespeichert werden, um die graphische Darstellung zu erzeugen. Die Position bei den Verbindungen wird benötigt, um eine Verbindung darzustellen, deren Enden noch nicht verbunden wurden. Dies ist ein Zwischenschritt während der Erzeugung einer Verbindung, für den u.U. auch eine graphische Darstellung erzeugt werden muß, siehe Abschnitt 4.5.2.1.

Für die Notation der Produktionen und der persistenten Attribute wurde eine Syntax gewählt, die an der Sprache Lido [Kastens 1999] angelehnt ist. Lido ist die Spezifikationssprache des Liga-Systems und enthält unter anderem Konstrukte für Produktionen einer kontextfreien Grammatik und für Attribute. Indem die Lido-Notation für Grammatikproduktionen und für Attribute übernommen wurde, wurde es vermieden, eine weitere, andersartige Notation dafür einzuführen. Im Nachhinein hat sich aber gezeigt, daß dies zu Verwirrung führt, weil beide Spezifikationssprachen, die hier eingeführte und Lido, gleichzeitig für die Spezifikation einer visuellen Sprache verwendet werden.

4.3. Sichten

In Entwicklungsumgebungen, die mit VLEli erzeugt werden, können mehrere Fenster verschiedene Teile eines visuellen Programms darstellen. Die verfügbaren Fenstertypen werden durch Sichten spezifiziert. Die Bezeichnung “Sicht” wurde für dieses Konzept gewählt, da Teile eines visuellen Programms durch unterschiedliche Fenstertypen in verschiedener Weise dargestellt werden können. Die Bezeichnung “Sicht” wird auch in anderen Autoren in diesem Zusammenhang verwendet, z.B. von Grundy u.a. [1998] und Meyers [1991].

Die ersten beiden Unterabschnitte motivieren den Wunsch nach verschiedenartigen Fenstern und gehen auf die Bestandteile ihrer Spezifikation in VLEli ein. Der dritte Abschnitt liefert ein Anwendungsbeispiel und diskutiert komplexere Anwendungen von Sichten. Abschließend gehe ich auf die Implementierung von

Sichten ein, die unterschiedliche abstrakte Strukturen für ihre Darstellung benötigen.

4.3.1. Motivation

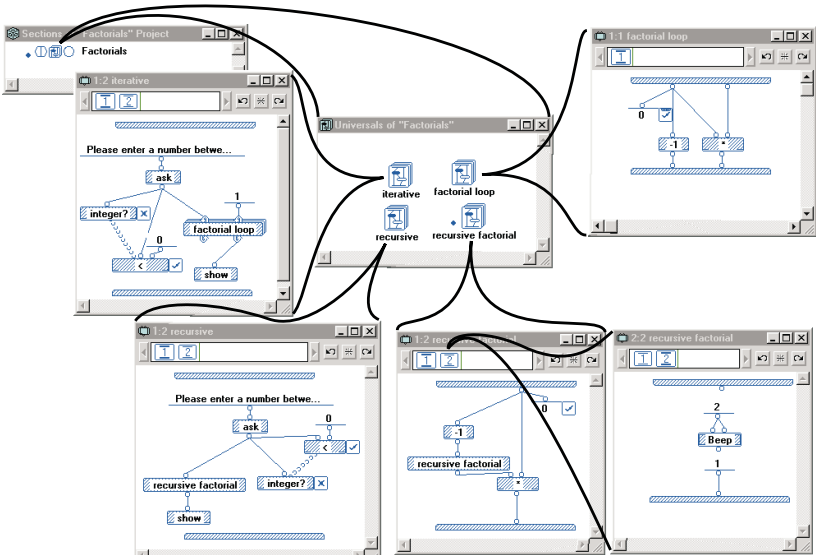
Die Informationen, die sich sinnvoll gleichzeitig auf einer Zeichenfläche darstellen lassen, sind beschränkt. Zum einen begrenzt das Ausmaß der Zeichenfläche, sei es das Papier oder der Bildschirm, die gleichzeitig darstellbare Informationsmenge. Zum anderen lassen sich umfangreichere Darstellungen nur schlecht überschauen und sind dadurch schwer verständlich. Im Bereich der visuellen Sprachen wurde in diesem Zusammenhang der Begriff *Deutsch-Limit* geprägt. Er besagt, daß sich bei der visuellen Programmierung "nicht mehr als 50 visuelle Primitive zur selben Zeit auf dem Bildschirm" [McIntyre 1998, Frage 12, übersetzt] darstellen lassen.

Es gibt verschiedene Techniken, um die Informationen größerer visueller Programme übersichtlich und verständlich anzuzeigen [Burnett u.a. 1995]. Eine sehr einfache Methode ist es, mehrere Fenster zu benutzen. Diese können verschiedene Teile eines visuellen Programms darstellen und sich im Detailreichtum unterscheiden. Einige Fenster liefern umfassendere Übersichten, stellen dafür aber nur grobe Zusammenhänge dar und reduzieren so den Platzverbrauch. Andere Fenster enthalten mehr Details, sind dafür aber auf einen kleineren Ausschnitt eines visuellen Programms beschränkt.

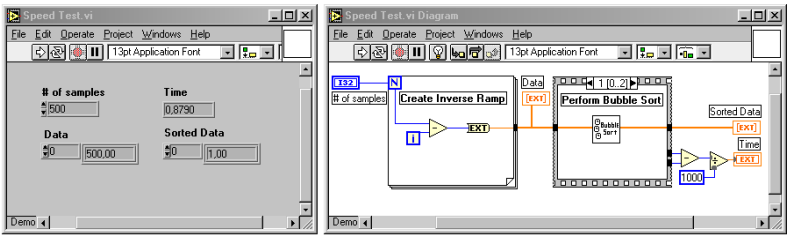
Ein Beispiel für die Benutzung von Fenstern liefert Prograph [Cox u.a. 1989], siehe Abbildung 4.8(a). Das Fenster oben links enthält die Namen aller Klassen eines Prograph-Programms. Das nächste Fenster in der Hierarchie zeigt alle Universals (Methoden) einer Klasse an. Es ist in der Mitte der Abbildung. Weiter untergeordnet sind Fenster, die die Fälle (Datenflußgraphen) einer Methode anzeigen. Um die Hierarchie zu verdeutlichen, sind Linien zwischen den Fenstern dargestellt, die nicht zur Sprache Prograph gehören. Diese sollen außerdem veranschaulichen, daß bei Prograph oft die Informationen einer Sicht in der übergeordneten Sicht durch ein Icon abstrahiert werden. Ein Benutzer kann ein Icon anwählen, um ein neues Fenster mit der Detaildarstellung zu erhalten.

Weiterhin können auch verschiedene Aspekte eines visuellen Programms in unterschiedlichen Fenstern dargestellt werden. Ein Beispiel dafür sind die virtuellen Instrumente in Labview [Vose und Williams 1986]. Im Beispiel aus Abbildung 4.8(b) wird im linken Fenster die Benutzungsschnittstelle eines virtuellen Instruments und im rechten Fenster die Programmlogik bearbeitet. Die einzelnen Teile der Benutzungsschnittstelle treten in der Programmlogik als Datenquellen oder Datensenken auf.

Ein drittes Beispiel für die Benutzung von Fenstern liefert die Sprache UML [Rational Software Corporation u.a. 1999]. In UML kann ein Benutzer beliebig viele verschiedene Klassendiagramme für eine Klassenhierarchie erstellen, die



(a) Hierarchie von Fenstern bei Prograph



(b) Fenster zeigen verschiedene Aspekte eines virtuellen Instruments in Labview

Abbildung 4.8.: Fenster in verschiedenen visuellen Sprachen

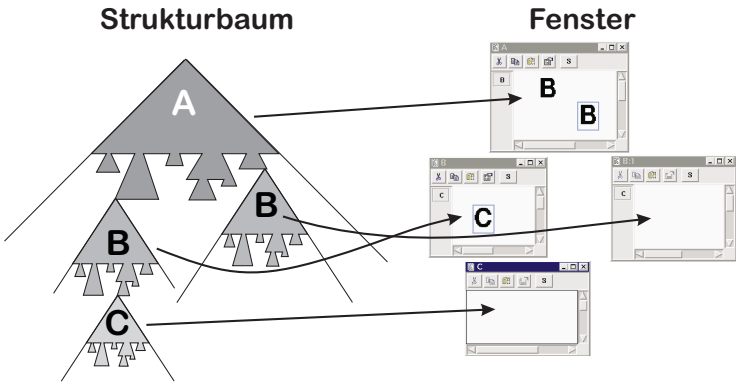


Abbildung 4.9.: Darstellung von Teilen eines visuellen Programms durch Fenster

jeweils unterschiedliche Entwurfsaspekte hervorheben. Alle Diagramme stellen dieselbe Hierarchie dar. In den verschiedenen Diagrammen können aber unterschiedliche Teilmengen der definierten Klassen an verschiedenen Positionen und in unterschiedlicher Weise dargestellt werden.

4.3.2. Spezifikation

Meyers [1991] führt die hier umgesetzte Art der Implementierung von Sichten unter dem Namen “Kanonische Repräsentation” (übersetzt). Sie ist dadurch charakterisiert, daß die verschiedenen Sichten aus einer gemeinsamen, kanonischen Repräsentation der Informationen gewonnen werden. Ihre Vorteile sind die einfache Konsistenthaltung, die Unterstützung sich gleichzeitig ändernder Sichten und die Vermeidung von Redundanz. Dies sind wichtige Eigenschaften für das Editieren eines visuellen Programms. Die Nachteile dieser Strategie sind die relativ hohe Komplexität für das Hinzufügen weiterer, vorher nicht eingeplaner Sichten. Dies erfordert u.U. die Änderung der Repräsentation, das ist hier die Grammatik, die den Strukturbaum beschreibt. Da die graphische Darstellung der Fenster aber weitgehend durch Symbolattributierung erzeugt wird, sind Erweiterungen des Strukturbaums relativ einfach durchführbar.

Abbildung 4.9 veranschaulicht die Implementierung der Sichten. Der Strukturbaum läßt sich in vier Bereiche aufteilen, die in vier Fenstern eines visuellen Editors dargestellt sind. Diese vier Fenster gehören zu drei verschiedenen Sichten: Sicht B wird derzeit durch zwei verschiedene Fenster für zwei unterschiedliche

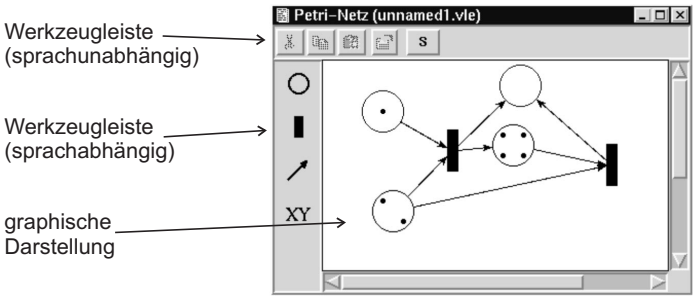
Teile des visuellen Programms verwendet. Die graphische Darstellung aller Fenster wird jedoch aus einem gemeinsamen Strukturbaum erzeugt.

Die verschiedenen Fenstertypen einer visuellen Sprache werden in VLEli durch Sichtdefinitionen spezifiziert. Abbildung 4.10 enthält die (einzige) Sichtspezifikation für die Sprache der Petri-Netze und das daraus erzeugte Fenster. Das Fenster enthält die graphische Darstellung eines Programmausschnitts und zwei Werkzeugleisten. Die beiden Wrkzeugleisten dienen der Bearbeitung des dargestellten Programmausschnitts. Die obere enthält Editieroperationen wie "Ausschneiden", "Einfügen" und "Eigenschaften des selektierten Sprachkonstrukts bearbeiten". Diese Editieroperationen sind allen generierten Entwicklungsumgebungen gemeinsam, so daß der Inhalt dieser Werkzeugleiste nicht spezifiziert zu werden braucht. Die Werkzeugleiste am linken Rand des Fensters ist dagegen von der editierten Sprache abhängig. Sie enthält Sprachkonstrukte, die aus der Werkzeugleiste in die graphische Darstellung gezogen und dort eingefügt werden können.

Die Sicht-Spezifikation beschreibt die Überschrift der Fenster und den Inhalt

```
VISUAL petri (ROOT):
        TITLE "PetriNetz";
        BUTTON IMAGE "circle" INSERTS pStateEl;
        BUTTON IMAGE "rectangle" INSERTS pTransEl;
        BUTTON IMAGE "arrow" INSERTS pConn;
        BUTTON IMAGE "xy" INSERTS pCommEl;
END;
```

(a) Spezifikation



(b) Resultierendes Fenster

Abbildung 4.10.: Sicht-Spezifikation für Petri-Netze

der sprachabhängigen Werkzeugleiste. Einer Sicht ist über ihren Namen im weiteren ein Attributauswerter zugeordnet, der die graphische Darstellung eines Fensters erzeugt. Die folgenden Abschnitte gehen auf die Spezifikation der graphischen Darstellung und auf die Werkzeugleiste ein.

4.3.2.1. Graphische Darstellung einer Sicht

Als wichtigster Bestandteil ist einer Sicht ein Attributauswerter zugeordnet. Dieser Attributauswerter berechnet für ein Fenster aus den Informationen eines Teils eines Strukturbaums die graphische Darstellung dieses Programmteils. Der dargestellte Programmteil wird nach oben hin abgegrenzt, indem ein Grammatiksymbol für die Sicht spezifiziert wird. Für die Sicht der Petri-Netze in Abbildung 4.10(a) ist dies ROOT, die Wurzel der Grammatik. Beim Öffnen eines Fensters für eine Sicht wird ein auf dieses Grammatiksymbol passender Kontext des Strukturbaums übergeben. Dieser bestimmt den im Fenster dargestellten Programmteil. Die Abgrenzung nach unten hin ist nicht erforderlich, da sie implizit durch die Berechnungen des Attributauswerters festliegt.

Um den Attributauswerter zu spezifizieren, wird eine attributierte Grammatik angegeben, die der Sicht über deren Name, bei den Petri-Netzen petri, zugeordnet ist. Teile einer attributierten Grammatik, die in Dateien mit der Endung .petri.lido geschrieben werden, spezifizieren die graphische Darstellung der petri-Sicht.

Bei Attributauswertern, die durch Besuchssequenzen gesteuert werden [Kastens 1980], ist hierbei eine Optimierung möglich. Man geht hierzu von der Annahme aus, daß zur Erzeugung der graphischen Darstellung tatsächlich lediglich die Informationen benötigt werden, die im Teilbaum einer Sicht enthalten sind. Ist dies der Fall, so kann man den Strukturbaum, für den die attributierte Grammatik ausgewertet wird, auf den Teilbaum der Sicht beschränken. Bei durch Besuchssequenzen gesteuerten Attributauswertern startet man dazu den Attributauswerter mit der Wurzel des *Teilbaums*, statt der Wurzel der Grammatik. Man vermeidet so Attributberechnungen für andere, in einem konkreten Fenster nicht dargestellte Kontexte. Dies spart nicht nur die Zeit, die diese Berechnungen benötigen, sondern vermeidet zudem noch in der Spezifikation die Unterscheidung zwischen dargestellten und nicht dargestellten Kontexten.

In der Praxis ist dies oft, aber leider nicht immer ohne weitere Maßnahmen durchführbar. Das liegt daran, daß zwischen dem Teilbaum einer Sicht und anderen Teilen des visuellen Programms nichthierarchische Beziehungen existieren können. Die graphische Darstellung eines Teilbaumes kann dadurch von Eigenschaften abhängen, die außerhalb des Teilbaums der Sicht gespeichert sind.

Ein Beispiel dafür ist die graphische Darstellung einer Streets-Prozedur, siehe Abbildung 2.9. Prozeduren werden bei Streets in separaten Fenstern dargestellt und enthalten unter anderem Ports, welche durch farbige Dreiecke dargestellt werden. Die Farbe eines Ports visualisiert dessen Typ und kann vom Sprachbenutzer

bei der Typdefinition festgelegt werden. Die Typdefinitionen sind im Strukturbaum aber außerhalb der Prozeduren gespeichert. Um eine Prozedur darzustellen, muß die Farbe entweder zusätzlich an den Anwendungsstellen gespeichert werden oder von der Definition zu ihren Anwendungsstellen im Prozedurfenster transportiert werden. Die zweite Technik vermeidet Inkonsistenzen und wurde deshalb hier bevorzugt.

Um die angeprochene Optimierungsmaßnahme trotzdem anwenden zu können, kann man einen weiteren Attributauswerter benutzen. Dieser wird mit der Wurzel des Strukturbaums *vor* den Attributauswertern für die Sichten gestartet. Er kann die benötigten Informationen, in diesem Fall also die Farbe eines Typs von ihrer Definitionsstelle zu den Anwendungsstellen transportieren. Auf diese Weise ist die Information für den Attributauswerter einer Sicht verfügbar und kann benutzt werden, um die graphische Darstellung zu erzeugen.

Diese Methode wurde in VLEli umgesetzt. Interessant war dabei, ob der für den Informationstransport eingesetzte Attributauswerter effizient genug ist, um den Editierablauf nicht zu stören. Dieser Attributauswerter muß schließlich nach jeder Editieroperation einmal aufgerufen werden.

In den von uns implementierten Beispielsprachen war dies der Fall. Der zusätzliche Attributauswerter führte auch bei umfangreicheren Programmen zu keiner bemerkenswerten Verzögerung, siehe dazu auch Abschnitt 5.2.2.3. Hierbei kommt hinzu, daß der Attributauswerter auch für weitere Aufgaben eingesetzt wird: mit ihm kann man auch semantische Prüfungen eines visuellen Programms durchführen und die Konsistenz zwischen Definition und Anwendung eines Programmobjekts aufrecht erhalten. Diese Anwendungsmöglichkeiten werden in Abschnitt 4.6.1 diskutiert.

4.3.2.2. Werkzeugleiste einer Sicht

Wie bereits oben erwähnt, enthält die sprachabhängige Werkzeugleiste Sprachkonstrukte, die in die graphische Darstellung gezogen und so dort eingefügt werden können. Die Programmkonstrukte der Werkzeugleiste können durch einen beschreibenden Text oder durch ein Bild kenntlich gemacht werden. Verwendet man ein Bild, so sollte dieses möglichst die graphische Darstellung des eingefügten Konstrukts widergeben, wie dies in Abbildung 4.10(b) der Fall ist.

Ein Benutzer kann die Sprachkonstrukte aus der Werkzeugleiste in die graphische Darstellung ziehen. Dabei wird jeweils die am nächsten befindliche legale Einfügestelle kenntlich gemacht. Diese Art der Benutzerinteraktion wird von Minör [1992] aufgrund einer Untersuchung textueller Strukturediteren empfohlen. Dieser Empfehlung liegt das Konzept der direkten Manipulation [Shneiderman 1983] zugrunde.

Technisch gesehen werden neue Sprachkonstrukte in der Programmrepräsentation des visuellen Programms eingefügt. Dabei wird in einem, durch die Ein-

fügestelle bestimmten Kontext des Strukturbaums eine Produktion angewendet. Die Spezifikation der Werkzeugleiste enthält deshalb auch Namen, die sich auf Produktionen aus der Definition der Programmrepräsentation beziehen, siehe Abb. 4.10(a) und Abb. 4.7(b) auf Seite 111.

Die in einem Fenster anwendbaren Sprachkonstrukte entsprechen einer Teilmenge der Produktionen. Es ist nicht sinnvoll, die Werkzeugleiste automatisch, beispielsweise aus den Namen der Produktionen zu generieren oder eine gemeinsame Werkzeugleiste für alle Fenster zu verwenden. Ein Grund dafür ist, daß nicht alle Produktionen in einem bestimmten Fenster anwendbar sind. Weiterhin kann ein Sprachentwerfer für die Anwendung einiger Produktionen andere Interaktionstechniken verwenden wollen, so daß einige Produktionen in der Werkzeugleiste nicht benötigt werden. Schließlich sollen die anwendbaren Produktionen möglichst durch Bilder beschrieben werden, die die graphische Darstellung der Sprachkonstrukte in einer Sicht wiedergeben. Ein Sprachkonstrukt kann aber in verschiedenen Fenstern in unterschiedlicher Weise dargestellt werden.

4.3.3. Beispiel und Diskussion

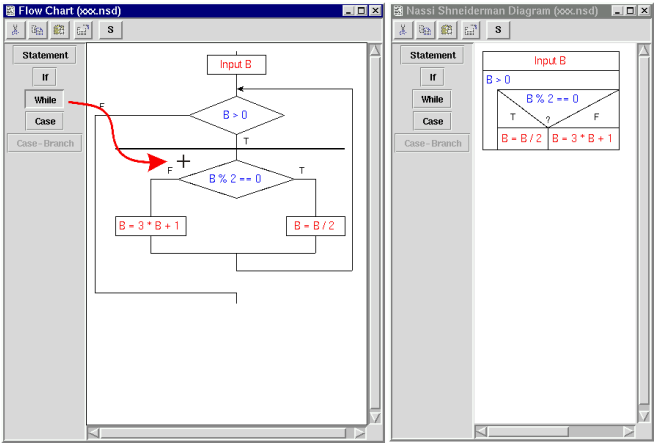
Bei der Motivation der Fenster in Abschnitt 4.3.1 wurden drei verschiedene Anwendungsfälle für Fenster identifiziert. Anwendungen, die wie Prograph für einen Programmteil immer nur eine Sorte von Fenstern benutzen, sind am häufigsten. Sie lassen sich mit dem gezeigten Ansatz sehr einfach umsetzen, weil die Informationen für die Fenster in unterschiedlichen Teilen des Strukturbaums gespeichert werden.

Ein Beispiel für die Anwendung von Sichten der LabView-Kategorie sind strukturierte Programme. Diese können zum einen durch Nassi-Shneiderman-Diagramme und zum anderen durch Flußdiagramme dargestellt werden. Die Fenster in Abbildung 4.11(a) zeigen verschiedene Sichten desselben Strukturbaums. Wenn für eines der Fenster eine Editieroperation durchgeführt wird, so wird sie durch eine Änderung des Strukturbaums realisiert. Für beide Fenster wird daraufhin die graphische Darstellung aktualisiert, so daß in beiden die Strukturänderung sichtbar wird, siehe Abbildung 4.11(b).

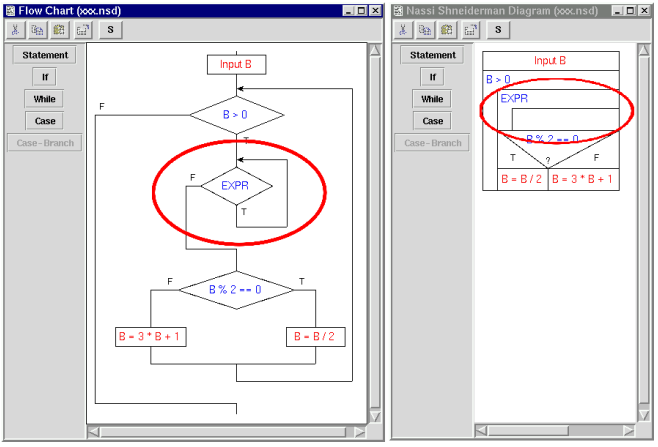
Bei dieser Methode der Implementierung von Sichten werden visuelle Programme durch *eine* Datenstruktur gespeichert, aus der die graphische Darstellung *aller* Sichten erzeugt wird. Es sind also keine Mechanismen zur Aufrechterhaltung der Konsistenz notwendig.

Die in Abbildung 4.11 gezeigte Anwendung ist relativ einfach: Beide Darstellungen lassen sich automatisch anordnen und benötigen keine weiteren Angaben eines Benutzers. Weiterhin benutzen beide Darstellungen dieselben abstrakten Strukturen und enthalten keine Angaben, die nur für eine Sicht gelten.

In komplexeren Anwendungen können zum Beispiel Layoutangaben des Be-



(a) Einfügen einer Schleife in einer Sicht...



(b) ...ändert die Darstellung beider Sichten.

Abbildung 4.11.: Bearbeiten eines strukturierten Programms mit zwei Sichten

nutzers in einer oder in mehreren Sichten verwendet werden. Änderungen, die in einer Sicht durchgeführt werden, können somit nur schwer in anderen Sichten realisiert werden, weil die Angaben des Benutzers fehlen. In LabView [Vose und Williams 1986] wird dieses Problem gelöst, indem bestimmte Defaultwerte für diese Benutzerangaben eingesetzt werden. Das führt dazu, daß neue Programmkonstrukte, die in einer Sicht hinzugefügt werden, sich in der jeweils anderen Sicht "aufeinanderstapeln" und weitere Benutzerinteraktion erfordern. Diese Strategie ließe sich auch in diesem Ansatz einsetzen, wobei durch den Einsatz von Constraints sogar eine überlappungsfreie Darstellung erreicht werden könnte.

In anderen Anwendungen kann es erforderlich werden, für die verschiedenen Sichten unterschiedliche abstrakte Strukturen zu verwenden. Ein Beispiel dafür ist ein Graph, der in verschiedenen Sichten einmal als Graph und ein anderes Mal als Adjazenzmatrix dargestellt wird [vgl. McWhirter und Nutt 1994]. Sollen für die Erzeugung der Darstellungen visuelle Muster verwendet werden, so würden diese Muster für ihre Anwendung unterschiedliche abstrakte Strukturen, einmal einen Graph, ein andermal eine Matrix benötigen.

Für die ditte Art der Anwendung von Fenstern in visuellen Sprachen ist mir derzeit nur ein Anwendungsbeispiel bekannt: UML-Klassendiagramme. Hierbei entwirft ein Benutzer eine Klassenhierarchie und kann für diese Hierarchie beliebig viele Klassendiagramme erzeugen. In den Diagrammen können Klassen mehrfach auftreten, unterschiedlich dargestellt sein oder sogar fehlen.

Auch solche Anwendungen lassen sich mit VLEli realisieren. Man muß dafür aber verschiedene Teilbäume im Strukturbaum für die verschiedenen Klassendiagramme einführen. Bei Änderungen müssen diese Teile des Strukturbaums gegeneinander abgeglichen werden. Diesen Aspekt beleuchtet der folgende Abschnitt.

4.3.4. Model-View

Die im letzten Abschnitt skizzierten komplexeren Anwendungen von Sichten sind dadurch charakterisiert, daß im visuellen Programm Programmobjekte definiert sind, die durch Vertreter in mehreren Fenstern visualisiert werden. Dabei können in jedem Fenster auch mehrere Vertreter für ein Programmobjekt existieren oder den Vertretern können zusätzliche lokale Eigenschaften zugeordnet werden. Um derartige Strukturen zu speichern, muß an einer Programmstelle das Programmobjekt definiert werden. Weiterhin müssen für jeden Vertreter eigene Strukturen im Strukturbaum vorgesehen werden. Da bei jedem Vertreter auch Änderungen am Programmobjekt durchgeführt werden können, müssen bei Editieroperationen Informationen an mehreren Stellen des Strukturbaums miteinander synchronisiert werden. Hierbei muß im Prinzip dasselbe Problem gelöst werden, für daß in Smalltalk das Model-View-Controller-Paradigma [Krasner und Pope 1988] entwickelt wurde.

Beim Model-View-Controller-Paradigma arbeiten drei Klassen zusammen. Die Objekte der Model-Klasse speichern die Informationen des Anwendungsprogramms, die in verschiedenen Sichten dargestellt werden sollen, das sind hier die Programmobjekte. Ändern sich die Informationen, so werden alle Sichten eines Models benachrichtigt. Die Objekte der View-Klasse gehören jeweils zu einem Model-Objekt. Sie sind hier die Vertreter der Programmobjekte. Erhalten die Sichten die Nachricht, daß sich die zugrundeliegenden Informationen geändert haben, so können sie ihre Darstellung aktualisieren. Die Objekte der Controller-Klasse gehören jeweils zu einem View-Objekt und bestimmen das Verhalten bei Eingaben des Benutzers. Hierbei werden insbesondere auch Änderungen an den Informationen durchgeführt, die im Model-Objekt gespeichert sind.

Die Überschrift dieses Abschnitts leitet sich aus der Ähnlichkeit des Problems zum Model-View-Controller-Paradigma her. In der abstrakten Struktur des visuellen Programms werden hierbei Programmkonstrukte benutzt, die Bezüge zu der Model- und der View-Klasse haben. Die Controller-Klasse findet in der im folgenden beschriebenen Lösung kein Äquivalent und wurde deshalb in der Überschrift nicht mit aufgeführt.

Die folgenden Absätze beschreiben *eine* Lösung des Model-View-Problems. Als Beispiel dient hier eine Mini-Sprache, in deren Modell eine Menge von Namen definiert sind. In den Fenstern, die zu einer Sicht gehören, können diese Namen jeweils an beliebige Positionen gesetzt und mit beliebigen Farben eingefärbt werden. Position und Farbe sind dabei unabhängig von anderen Sichten, die Namen werden mit anderen Sichten konsistent gehalten.

Die abstrakte Struktur, mit der das Model-View-Problem in VLEli gelöst werden kann, wird in Abbildung 4.12 skizziert. Im Strukturbaum sind Strukturen für das Modell und für die Sichten enthalten. Im Modell sind Definitionen von Programmobjekten enthalten, denen hier nur eine Eigenschaft, ihr Name zugeordnet ist. Die Definitionen im Modell sind mit den Vertretern der Programmobjekte (ViewOfDef) durch Einträge in der Definitionstabelle verbunden. Den Vertretern sind jeweils weitere Eigenschaften, ihre Farbe und ihre Position zugeordnet. Das Attribut isOld wird zur Herstellung der Konsistenz benötigt und wird weiter unten beschrieben.

Würde man die hier präsentierte Struktur auf die Klassendiagramme übertragen, so würde die Klassenhierarchie dem Modell (Model) und ein Klassendiagramm einer Sicht (View) entsprechen. Das Symbol ViewOfDef hat seine Entsprechung in den Klassen eines Klassendiagramms, die Definition ist ein Konstrukt, daß einer Klasse in der Klassenhierarchie entspricht. Diese bekommt ein Benutzer in der Regel nicht zu “sehen”.

Abbildung 4.12 zeigt die Programmrepräsentation des Modells und zweier Sichten. Die eine Sicht enthält nur eine Definition, nämlich die für B, die andere beide. Der vollständige Strukturbaum enthält alle zur Darstellung der Sichten benötigten Informationen. Im Teilbaum einer Sicht sind jedoch die Namen

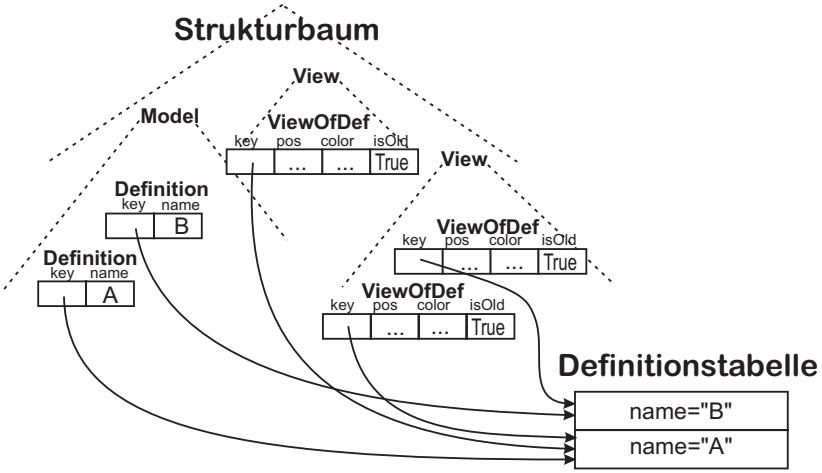


Abbildung 4.12.: Abstrakter Strukturbaum für die Implementierung von Model-View

der Definitionen nicht enthalten. Diese werden durch einen zusätzlichen Attributauswerter von der Definition zur SichtDefinition transportiert, so daß hier keine Inkonsistenzen entstehen können. Auf diesen Aspekt ist bereits Abschnitt 4.3.2 eingegangen.

Editieroperationen, die potentiell die Konsistenz zwischen dem Modell und den Sichten stören können, sind das Einfügen oder Löschen des Definition- oder ViewOfDef-Knotens. Um nach einer Editieroperation die Konsistenz zwischen dem Modell und den Sichten wieder herzustellen, kann ebenfalls der zusätzliche Attributauswerter verwendet werden. Eine wichtige Voraussetzung dafür ist, daß der Attributauswerter die verschiedenen Aktionen des Benutzers eindeutig anhand des Strukturbaums unterscheiden kann.

Es gibt eine Situation, bei der unklar ist, auf welche Weise die Konsistenz wiederhergestellt werden soll: Eine Sichtdefinition, zu der keine Definition existiert, ist in mindestens einer Sicht enthalten. Diese Situation entsteht zum einen, wenn der Benutzer eine neue SichtDefinition in einer Sicht einfügt. In diesem Fall muß eine dazu passende Definition im Modell erzeugt werden. Dieselbe Situation im Strukturbaum entsteht jedoch auch, wenn der Benutzer durch direkte Interaktion mit dem Modell eine Definition entfernt. In diesem Fall müssen alle SichtDefinitionen der gelöschten Definition entfernt werden. Um diese beiden Fälle voneinander zu unterscheiden, kann einer SichtDefinition das zusätzliche Attribut isOld

zugeordnet werden. Das Attribut isOld wird bei der Erzeugung einer SichtDefinition auf False gesetzt. Immer wenn die Konsistenz hergestellt wird, erhalten diese Attribute dann den Wert True. Beim Einfügen einer SichtDefinition ist das Attribut dann noch False, während es nach der Löschung einer Definition bereits den Wert True besitzt.

Der Attributauswerter hat also die benötigten Informationen, um die Konsistenz zwischen dem Modell und den Sichten herzustellen. Dazu muß er aber den Strukturbaum modifizieren. Es müssen die Werte persistenter Attribute geändert, Strukturbaumknoten hinzugefügt und auch entfernt werden können, beispielsweise um nach Löschung einer Definition auch alle SichtDefinitionen zu löschen. Da man während der Attributauswertung im allgemeinen nicht weiß, ob Informationen aus einem zu löschenden Strukturbaumknoten noch benötigt werden, wird hierzu die Technik der verzögerten Auswertung eingesetzt. In VLEli werden hierzu die durchzuführenden Strukturbaumänderungen als Script einer Script-Sprache (Tcl [Welch 1997]) berechnet. Wenn der Attributauswerter vollständig abgearbeitet ist, kann durch Ausführung des erzeugten Scripts der Strukturbaum angepaßt werden.

Eine dazu verwandte, aber weniger flexible Technik benutzen Backlund u.a. [1990] für das LOGGIE-System. Im LOGGIE-System können Attribute gekennzeichnet werden, die bei Erreichen bestimmter vordefinierter Werte zum Ausführen von Prozeduren führen, die weitere Änderungen im Strukturbaum durchführen können.

In diesem Abschnitt wurde gezeigt, wie für das Problem der Konsistenthaltung verschiedener Teilbäume attributierte Grammatiken verwendet werden können. Attributierte Grammatiken sind ein geeignetes Mittel, um die Inkonsistenz festzustellen und Code für die Herstellung der Konsistenz zu erzeugen, weil man die Art der Nutzung verschiedener Sichten auch als eine Spracheigenschaft betrachten kann. Die Definition-Kontexte definieren Programmobjekte, die bei den ViewOfDef-Kontexten angewendet werden. Die beschriebenen Attributberechnungen stellen in diesem Licht betrachtet die Konsistenz einer Definition mit ihren Anwendungsstellen unter der Maßgabe her, daß ein Benutzer strukturelle Änderungen der Definitionen im Kontext ihrer Anwendungen durchführt.

Die Benutzung attributierter Grammatiken erlaubt weiterhin, die erforderlichen Berechnungen in Spezifikationsmodulen zu abstrahieren und diese zu benutzen, um das Sichtkonzept auf einer höheren Ebene zu unterstützen. Dieser Aspekt wird in Arbeiten, die auf den hier präsentierten Grundlagen aufbauen, anhand der UML-Klassendiagramme untersucht.

4.4. Graphische Darstellung

Der vorige Abschnitt hat die Spezifikation der Fenster in VLEli vorgestellt. Einem Fenster ist als wichtigster Bestandteil ein Attributauswerter zugeordnet. Dieser Attributauswerter hat zwei Aufgaben. Zum einen muß er aus dem abstrakten Strukturbaum die graphische Darstellung eines Fensters erzeugen, zum anderen die Editieroperationen bestimmen, die der Benutzer im jeweils nächsten Schritt anwenden kann.

Dieser Abschnitt ist den Attributberechnungen gewidmet, die die graphische Darstellung erzeugen. Auf die Editieroperationen gehe ich im folgenden Abschnitt ein.

Um die graphische Darstellung aus der abstrakten Struktur zu erzeugen, müssen Aufgaben aus zwei Bereichen gelöst werden. Zum ersten müssen die darzustellenden Informationen aus der abstrakten Programmrepräsentation extrahiert werden. Das kann dann zu komplexeren Berechnungen führen, wenn diese Informationen nicht direkt in der Programmrepräsentation gespeichert, sondern aus anderen Informationen des Programms berechnet werden müssen.

Zum zweiten muß die Anordnung der Konstrukte in der graphischen Darstellung berechnet werden. Hierbei müssen aus den manuell festgelegten Layoutangaben des Benutzers die anderen Angaben bestimmt werden. Ziele sind dabei zum einen, das Layout in möglichst großem Umfang durch den Benutzer steuerbar zu machen, zum anderen, nach Programmänderungen möglichst alle erforderlichen Layoutanpassungen automatisch durchzuführen.

Zur Lösung beider Aufgaben eignen sich Attributauswerter. Der erste Bereich hat enge Beziehungen zur semantischen Analyse textueller Programme. Auch hier ist die wesentliche Aufgabe, semantische Informationen über ein Programm zu ermitteln und diese Informationen in anderen Kontexten verfügbar zu machen. Dieses ist das klassische Aufgabenfeld attributierter Grammatiken [Kastens 1991a]. Die erzeugten Attributauswerter werden dabei von anderen Modulen und Generatoren unterstützt, die auch bei der Erzeugung der graphischen Darstellung wertvolle Dienste leisten.

Für die Spezifikation der Layoutberechnungen werden oft Constraint-Netzwerke statt attributierter Grammatiken auf der niedrigsten Ebene verwendet [Bardohl 1998; McWhirter und Nutt 1994; Rekers und Schürr 1996]. Mit Constraint-Netzwerken werden dazu gültige Layouts durch Systeme von Gleichungen und Ungleichungen beschrieben, deren Variable die Positionen und ggf. auch die Größen der Sprachkonstrukte repräsentieren. Nach jeder Programmänderung wird ein Constraint-Solver aufgerufen, der eine gültige Belegung für die Variablen des Constraint-Netzwerks findet und damit die Layoutänderungen der graphischen Darstellung berechnet. Bei einem Constraint-Solver können

dabei Konzepte umgesetzt werden, die möglichst kleine Layout-Änderungen gewährleisten.

In VLEli sind auch Constraint-Netzwerke anwendbar, um das Layout einer graphischen Darstellung zu beschreiben. Constraint-Netzwerke und Attributauswerter ergänzen sich dabei gut. Für die Erzeugung der graphischen Darstellung nach einer Programmänderung kann so in einer ersten Phase die Konsistenz der Layoutparameter sichergestellt werden. Der Attributauswerter berechnet dazu ein (Un-)Gleichungssystem, das die Konsistenzbedingungen beschreibt und übergibt es zusammen mit den relevanten Layoutparametern dem Constraint-Solver. Dieser prüft die Konsistenz und ändert ggf. einige der Layoutparameter, um ein nicht erfülltes Contraint-Netzwerk zu erfüllen. Im Anschluß daran berechnet der Attributauswerter aus den evtl. korrigierten Layoutparametern die graphische Darstellung. Hierzu können auch weitere Layoutberechnungen durchgeführt werden.

Durch das Zusammenwirken der Constraint-Netzwerke und der Attributauswerter lassen sich Layoutkonzepte umsetzen, die sich allein durch nur eines der Werkzeuge schwer realisieren ließen. Beispiele sind Sprachen, bei denen einerseits automatisch Überlappungsfreiheit, Berührung und Schachtelungsbeziehungen aufrechterhalten werden müssen und andererseits (für andere Sprachkonstrukte) Layoutkonzepte realisiert werden sollen, die nicht auf dem Prinzip kleinstmöglicher Änderungen beruhen etwa platzoptimierte Darstellungen.

Darüberhinaus werden die Constraints hier durch eine attributierte Grammatik spezifiziert, wodurch die für attributierte Grammatiken entwickelten Abstraktionstechniken anwendbar sind, um Constraints auf einer hohen Ebene zu beschreiben. Das ist sinnvoll, weil es sehr schwer ist, ein umfangreicheres Constraint-Netzwerk so zu formulieren, daß die vom Constraint-Solver durchgeführten Layoutänderungen den Erwartungen des Sprachentwerfers entsprechen [Griebel 1996]. Auch andere Autoren haben das erkannt und haben eine höhere Spezifikationsebene für Constraints eingeführt [Kamada und Kawai 1991].

Die folgenden Abschnitte gehen auf die Attributberechnungen ein, die für die Erzeugung der graphischen Darstellung eines visuellen Programms benötigt werden. Der erste Abschnitt ist der direkten Berechnung des Layouts durch einen Attributauswerter gewidmet, während der zweite die Umsetzung des constraintbasierten Layouts beschreibt. Der dritte Abschnitt gibt Beispiele für Attributberechnungen, die außer den Layoutberechnungen für die Darstellung eines visuellen Programms noch benötigt werden.

4.4.1. Layoutberechnung durch den Attributauswerter

Der Benutzungskomfort bei der Bearbeitung visueller Programme in Strukturediteren drückt sich unter anderem in den Layoutberechnungen aus. Wird ein visuelles Programm geändert, so soll das Layout der Darstellung möglichst automatisch

an diese Änderung angepaßt werden. Gelingt dies, so kann ein Benutzer ein visuelles Programm schneller und leichter ändern, da die zur Korrektur des Layouts benötigten Änderungen automatisch durchgeführt werden.

Green und Petre [1996] messen die Güte visueller Sprachen (und ihrer Implementierungen) und adaptieren in diesem Zusammenhang den Begriff der "Viskosität" aus der Physik: Die Viskosität mißt den benötigten Aufwand, um eine einzelne Editieroperation durchzuführen. Die Autoren bestimmen die Viskosität von Basic, Prograph und LabView, indem sie jeweils äquivalente Programmänderungen von erfahrenen Benutzern durchführen lassen. Sie ermitteln dabei, daß die Dauer von Änderungen in den drei Sprachen sich bei den Extremen um den Faktor 8 unterscheiden.[2] Ähnlich schlechte Erfahrungen mit LabView berichtet auch Poswig [1994, S. 23]. Sie resultieren daher, daß sich Größenänderungen der LabView-Konstrukte nachträglich nur schwer durchführen lassen und oft eine Neukonstruktion der Verbindungen erfordern.[3]

Grant [1998] generiert Strukturediteren für Sprachen, die sich weitgehend automatisch anordnen lassen. Der Autor hat die Benutzbarkeit der generierten Strukturediteren getestet und hat dazu die Konstruktion verschiedener Programme in einer textuellen und einer visuellen Notation von ML miteinander verglichen. Die Zahlen sind nicht direkt mit denen von Green und Petre [1996] vergleichbar, da einmal die Neukonstruktion, ein andermal der Änderungsaufwand gemessen wurde. Der sich ergebende Geschwindigkeitsvorteil des textuellen Programms ist hier jedoch mit $1,4 - 2,5$ deutlich niedriger und kann zumindest zum Teil als Vorteil der automatischen Anordnung der Sprachkonstrukte gewertet werden.

In den folgenden Absätzen werde ich Layoutalternativen visueller Sprachen an Beispielen vorstellen und jeweils die erforderlichen Attributberechnungen skizzieren. Eine erste Layoutalternative betrifft Sprachkonstrukte, die sich vollständig automatisch anordnen lassen. Diese benötigen keine weiteren Angaben des Benutzers und lassen sich besonders effizient bearbeiten, wie Grant [1998] gezeigt hat.

4.4.1.1. Automatisch berechnetes Layout

Sprachkonstrukte lassen sich automatisch anordnen, wenn sich deren Layouteigenschaften wie Position und Größe eindeutig berechnen lassen. Ein Benutzer kann zwar durch manuelle Angaben das Layout u.U. beeinflussen. Durch derartige Parameter kann das Layout aber nicht inkonsistent werden.

Die vorliegende Implementierung der Nassi-Shneidermann-Diagramme liefert Beispiele für automatisch angeordnete Sprachkonstrukte. Für die Implementierung wurde die Entscheidung getroffen, die Darstellung größenoptimiert zu zeich-

[2] Die genauen Zeiten habe ich in Tabelle 5.3 auf Seite 207 aufgeführt.

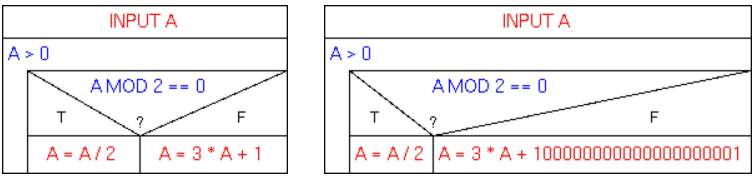
[3] Einige dieser Probleme wurden inzwischen behoben.

nen. Der Benutzer könnte zwar Parameter wie den Zeichensatz der Texte oder die Ausrichtung der Anweisungen ändern. Diese Parameter gehen jedoch höchstens als zusätzliche Hinweise in den Layoutalgorithmus ein, das Layout wird aus diesen automatisch berechnet.

Die Auswirkungen der automatisch durchgeführten Layoutberechnungen auf die Bearbeitung der Nassi-Shneiderman-Programme lassen sich anhand der Abbildung 4.13 nachvollziehen. Wenn man die Zuweisung an A rechts unten in Abbildung 4.13(a) ändert, so wird die Darstellung automatisch vergrößert, um Platz für den geänderten Ausdruck zu schaffen. Dies führt hier zu einer Änderung im If-Konstrukt und dadurch bedingt zu weiteren Layoutanpassungen. Das Ergebnis ist in Abbildung 4.13(b) dargestellt. Andere Editieroperationen, wie zum Beispiel das Einfügen neuer Konstrukte zwischen den anderen, führen in ähnlicher Weise automatisch zu Korrekturen. Dies betrifft auch Verkleinerungen: Wenn man die Änderung in Abbildung 4.13(b) rückgängig macht, so erhält man wieder die kompakte Darstellung aus Abbildung 4.13(a).

Um das Layout für Nassi-Shneiderman-Diagramme durch einen Attributauswerter zu erzeugen, werden Berechnungen entlang der Hierarchie der Sprachkonstrukte benötigt. In einer ersten Phase wird hierzu der minimale Platzbedarf der Sprachkonstrukte von den inneren zu den äußeren Strukturen hin berechnet. Dies entspricht Attributberechnungen, die die minimale Ausdehnung der graphischen Objekte von den Blättern zur Wurzel des Strukturbaums hin bestimmen. Anschließend werden die tatsächlichen Positionen der Sprachkonstrukte berechnet und dabei die Ausrichtung und andere Faktoren berücksichtigt.

Diese Vorgehensweise wird im System VPE verwendet, um visuelle Sprachen zu implementieren, deren Konstrukte sich in ähnlicher Weise automatisch anordnen lassen. Grant [1998] führt außer den Nassi-Shneiderman-Diagrammen noch Spezifikationen für eine visuelle Notation von ML [Cardelli 1983], für IDCCS [Giacalone und Smolka 1988] sowie für mathematische Formeln vor. Auch in anderen



(a) Vor der Änderung

(b) Nach der Änderung

Abbildung 4.13.: Automatische Größenanpassung eines Nassi-Shneiderman-Diagramms

Ansätzen wird diese Technik der Layoutberechnung für die Implementierung visueller Sprachen angewendet [Chabrier u.a. 1988; Lextrait und Zarli 1990]. Sie geht auf Knuth [1979] zurück und wird deshalb oft als *Box-and-Glue*-Methode bezeichnet.

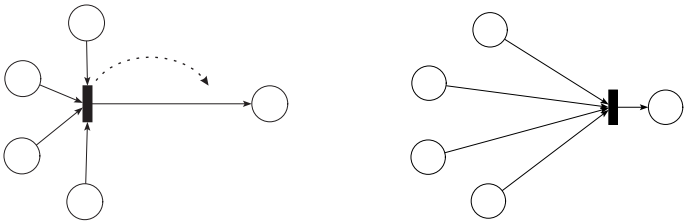
In ähnlicher Weise geht man auch beim Layout textueller Programme zur Durchführung der Aufgabe des Pretty-Printing vor. Auch hierbei muß man beispielsweise zunächst die Breite von Teilen eines Ausdrucks kennen, um zu ermitteln, wie dieser dargestellt werden soll.

Die Eigenschaft, die automatische Anordnung zu unterstützen, läßt sich im weiteren auch den visuellen Mustern zuordnen, deren Erarbeitung Kapitel 3 beschreibt. Alle Muster, bei denen die Relativpositionen eingeschachtelter Objekte eindeutig bestimmt ist, lassen sich automatisch anordnen. Außer dem Formular- und dem Listen-Muster, die bereits bei den Nassi-Shneiderman-Diagrammen auftreten, sind dies das Registerkarten-, das Stapel-, das Tabellen- und das Matrix-Muster. Auch hierbei werden in einer ersten Phase die Größen der Konstrukte von den inneren zu den äußeren Strukturen hin berechnet und anschließend genaue Positionen von außen nach innen bestimmt. Es kommt dabei aber im Strukturbaum zu komplexeren Auswertungsreihenfolgen, weil z.B. Tabellen aus einfacheren Strukturen zusammengesetzt sind.

4.4.1.2. Manuell unterstütztes Layout

Beim automatischen Layout wird eine Anordnung für ein Sprachkonstrukt automatisch berechnet. Das setzt voraus, daß es unter den Layoutalternativen eine "beste" gibt, die durch einen Algorithmus bestimmt werden kann und die die Größe und Position der Sprachkonstrukte festlegt. In einigen Fällen existiert ein derartiger Algorithmus nicht, hat zu lange Laufzeit oder führt nicht zum gewünschten Layout. Sprachen, deren Darstellung auf Graphen beruht, sind ein Beispiel dafür. Für die Sprachimplementierung kann man sich dafür entscheiden, zentrale Layoutparameter, z.B. die Position oder Größe einiger Konstrukte durch den Benutzer änderbar zu machen und diese Angaben für das Layout zu benutzen. Der wesentliche Unterschied zum automatischen Layout ist hierbei, daß durch diese Angaben auch inkonsistente Darstellungen möglich sind.

Durch die Möglichkeit, das Layout der graphischen Darstellung direkt beeinflussen zu können, kann ein Anwender anderen Betrachtern des visuellen Programms außerdem Lesehinweise geben, ohne die das visuelle Programm u.U. nur schwer verständlich ist. Petre und Green [1990] haben beispielsweise herausgefunden, daß "die Plazierung elektrischer Bauteile im Bereich des Computer Aided Design sehr wichtig für ihr Verständnis ist" (übersetzt) und haben dies zum Anlaß genommen, die Möglichkeiten für sekundäre Notationen als Gütekriterium für visuelle Sprachen und ihre Implementierung einzuführen [Green und Petre 1996], siehe Abschnitt 2.1.3.1.



(a) Vor der Änderung (b) Nach der Änderung

Abbildung 4.14.: Automatische Linienführung im Petri-Netz-Editor

Ein Nachteil des manuellen Layouts ist, daß es zu sehr hoher Viskosität der Sprachimplementierung führen kann. Um dies zu verhindern, kann man nur einige der Konstrukte eines visuellen Programms durch den Benutzer positionieren lassen und das Layout anderer Konstrukte daraus berechnen. Ein solches manuell unterstütztes Layout ist beispielsweise geeignet für visuelle Sprachen, die auf Graphen basieren, so auch für Petri-Netze. Eine automatische Positionierung der Knoten durch einen Graph-Layouter wird meist nicht zu den vom Benutzer gewünschten Resultaten führen, so daß die manuelle Festlegung der Knotenpositionen – also hier der Stellen und der Transitionen – sinnvoll ist.

Die Position der Knoten des Graphen kann wiederum verwendet werden, um daraus das Layout der Kanten – also hier der Verbindungen – automatisch zu berechnen. Müßte ein Benutzer auch das Layout der Kanten manuell festlegen, so würde ein solcher Editor kaum nennenswerte Vorteile gegenüber einem normalen “Malprogramm” aufweisen, die “Viskosität” einer solchen Sprache wäre sehr hoch.

Für die Realisierung dieses Layouts werden lediglich dann komplexere Berechnungen benötigt, wenn benutzerdefinierte Angaben in andere Kontexte transportiert werden müssen, um zum automatischen Layout beizutragen. Bei den Petri-Netzen müssen z.B. die Positionen der Stellen und Transitionen in die Kontexte der dort startenden oder endenden Verbindungen transportiert werden. Bei den Verbindungen werden aus diesen und weiteren Angaben die Koordinaten der Verbindungsendpunkte so ausgerechnet, daß diese auf dem Rand der Stellen bzw. der Transitionen enden.

Abbildung 4.14 zeigt ein Beispiel für die Auswirkung auf die Bearbeitung der Petri-Netze. Wird der Übergang wie durch den gestrichelten Pfeil angedeutet verschoben, so werden die verbindenden Pfeile automatisch nachgeführt. Die zur Erzeugung der Darstellung im einzelnen erforderlichen Attributberechnungen sind analog zu denen in Abbildung 4.1(d), Seite 97.

Durch diese Attributberechnungen können sich komplexere Berechnungsstrukturen ergeben, wenn Linien wiederum Endpunkte für weitere Linien sein können. Dies tritt beispielsweise in UML-Klassendiagrammen bei Constraint-Linien (siehe z.B. Abbildung 3.14(b), Seite 85) und bei Kommentaren auf, die mit Linien an beliebige Sprachkonstrukte, auch an Linien und Constraint-Linien angeheftet werden können. Die Grenze der Layoutberechnungen, die so durch zyklenfreie attributierte Grammatiken spezifiziert werden können, wäre überschritten, wenn die Verbindungslinie zwischen Modellelement und Kommentar selbst wieder kommentiert werden könnte.

Die Layoutangaben, die vom Benutzer angegeben werden, müssen bei dieser Art der Layouterzeugung nicht unbedingt die absolute Position oder die Größe von Sprachkonstrukten festlegen. Bei hierarchischen Strukturen wie den Zustandsdiagrammen sollen oft bei Verschiebung eines Konstrukts auch die darin befindlichen weiteren Konstrukte mitverschoben werden. Wird dies in der Sprachimplementierung gewünscht, so werden zweckmäßigerweise nicht die absoluten Koordinaten eines Konstrukts vom Benutzer spezifiziert, sondern nur dessen relative Position zum Ursprung des umgebenden Konstrukts. Der Attributauswerter kann daraus leicht die absoluten Positionen berechnen.

Weitere Varianten sind einsetzbar, beispielsweise um die Stützpunkte einer Polylinie oder die Beschriftung einer Linie automatisch nach dem Verschieben der Linienendpunkte zu adaptieren. Dazu kann das Koordinatensystem der Stützpunkte beispielsweise so gedreht und gestreckt oder gestaucht werden, daß bei Verschiebung der Linienendpunkte die Stützpunkte geeignet mitverschoben werden. Auch dabei müssen mit einem Attributauswerter die durchgeführten Koordinatentransformationen rückgängig gemacht werden, um die absoluten Koordinaten eines Stützpunktes zu ermitteln.

4.4.2. Layoutberechnung mit Constraints

Beim manuellen Layout werden einige Layoutangaben direkt durch den Benutzer gesetzt. Auf diese Weise sind auch Inkonsistenzen in der Darstellung möglich, die einen Benutzer verwirren und die durch zusätzliche Editieroperationen manuell korrigiert werden müssen. Mit Hilfe von Constraints läßt sich eine Verbesserung erzielen. Man beschreibt hierzu gültige Layoutkonfigurationen mit Hilfe von Gleichungen und Ungleichungen, die die gültigen Werte der vom Benutzer manipulierbaren Layoutparameter einschränken. Diese Relationen werden durch einen Constraint-Solver ausgewertet.

Ein wesentlicher Vorteil ergibt sich, wenn die Editieroperationen des Benutzers bewirken, daß einige der Constraints nicht mehr erfüllt sind. Ohne Constraint-Solver müßten solche Editieroperationen entweder abbelehnt oder eine ungültige Darstellung erzeugt werden. Bei der Benutzung von Constraints werden statt des-

sen einige andere Layoutparameter verändert, um damit die Relationen wieder gültig zu machen. Für die Petri-Netze könnte damit beispielsweise spezifiziert werden, daß sich die Stellen und Transitionen nicht gegenseitig überlappen dürfen. Wird eine Stelle oder eine Transition hinzugefügt oder verschoben, so wird das Layout der Darstellung ggf. automatisch so geändert, daß andere Programmkonstrukte dem neu hinzugefügten ausweichen. Dadurch lassen sich Änderungen schneller realisieren, was den Bearbeitungskomfort deutlich verbessert.

Ein wichtiger Aspekt für diese Arbeit ist es, daß das Constraintbasierte Layout mit den Layoutberechnungen kombiniert werden kann, die durch attributierte Grammatiken spezifiziert werden. Für Layoutaufgaben läßt sich so jeweils das geeignetere Spezifikationsmittel verwenden. Abschnitt 4.4.2.2 beleuchtet dieses Thema anhand von Beispielen. Davor beschreibe ich das Prinzip der Layoutberechnung durch Constraints. In Abschnitt 4.4.2.3 gehe auf die Auswahl des Constraint-Solvers Parcon [Griebel u.a. 1997, 1996; Griebel 1996] ein und diskutiere im Anschluß daran die technische Umsetzung der Einbindung von Parcon in die Attributauswerter.

4.4.2.1. Constraints

Die folgenden Absätze zeigen anhand eines Beispiels den Aufbau der Constraint-Netzwerke und geben einen Eindruck davon, in welcher Weise durch Constraints eine Verbesserung des Bearbeitungskomforts visueller Programme erreicht wird. Das Beispiel ist hier das Layout eines einfachen Zustands aus der Sprache der Zustandsdiagramme. Anhand dieses Beispiels wird am Ende des Abschnitts die Layoutberechnung durch Constraints gegen die Layoutberechnung durch Attributauswerter abgegrenzt.

In Abbildung 4.15 ist oben links ein einfacher Zustand eines Zustandsdiagramms abgebildet. Die Darstellung besteht aus zwei graphischen Objekten, einem Rechteck mit abgerundeten Ecken und einem Text. Das Rechteck hat eine Ausdehung, der Text eine Relativposition im Rechteck und ebenfalls eine Ausdehnung. Ein wichtiger Aspekt für die Spezifikation der Constraints ist, daß die Größe des Rechtecks sowie die Position des Textes änderbar sind, während die Ausdehnung des Textes als für den Constraint-Solver nicht änderbar angesehen werden muß. Für die Größe des Rechtecks sowie für die Relativposition des Textes werden persistente Attribute in der Programmrepräsentation eingeführt, deren Werte der Benutzer durch Editieroperationen ändern kann. Die Ausdehnung des Textes kann dagegen anhand der Zeichenfolge und dem Zeichensatz ermittelt werden und geht deshalb nicht in die Programmrepräsentation ein.

Wenn man einfache Zustände wie hier als aus zwei graphischen Objekten bestehend betrachtet, so müßte eine Sprachimplementierung die korrekte Schachtelung der Objekte sicherstellen: die Beschriftung des Zustandes muß sich immer innerhalb des Zustandsrahmens befinden. Das dazu erforderliche Constraint-

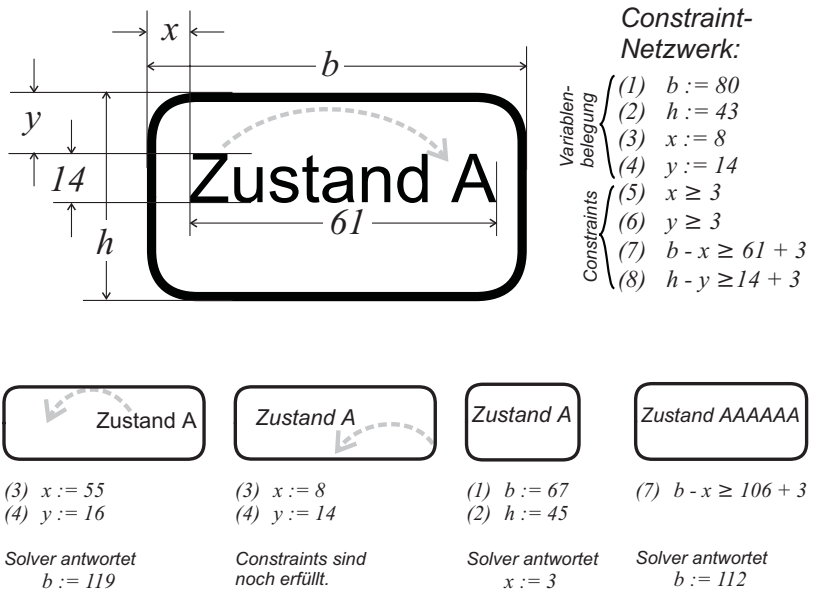


Abbildung 4.15.: Constraint-Netzwerk für Schachtelungsbeziehungen

Netzwerk ist rechts oben im Bild angegeben. Für die veränderlichen Layoutparameter enthält das Constraint-Netzwerk Variable, hier b, h, x und y. Nicht veränderliche Größen sind direkt als Werte im Constraint-Netzwerk eingesetzt, hier der Abstand 3 des Textes zum Rand des Rechtecks sowie die Breite 61 und Höhe 14 des Textes. Die Angaben (1) bis (4) initialisieren die Variablen des Constraint-Netzwerks, die Angaben (5) bis (8) definieren die Constraints, die hier gleichzeitig erfüllt sein müssen und somit konjunktiv verknüpft sind. Das Constraint-Netzwerk hat viele verschiedene Lösungen, ist also unterspezifiziert.

Der untere Teil der Abbildung 4.15 zeigt einige Editieroperationen, die durch grau gestrichelte Pfeile markiert sind. Der Zustand links unten geht aus dem obigen durch Verschiebung des Textes nach rechts hervor. Im Anschluß daran wurde der Text wieder nach links verschoben, der Zustandsrahmen verkleinert und der Text verlängert. Unter den Zuständen ist jeweils das geänderte Constraint-Netzwerk sowie die Reaktion des Constraint-Solvers auf ungültig gewordene Constraints dargestellt.

An den Editieroperationen sieht man, daß der Benutzer beliebige Layoutgrößen ändern kann. Aufgrund der ungerichteten Constraints werden für die restli-

chen Variablen passende Werte gefunden. Weiterhin sieht man anhand der Reaktionen auf die Editieroperationen auch gut das Prinzip der geringstmöglichen Änderung: Anders als bei der Layoutberechnung durch den Attributauswerter wird der Zustandsrahmen nicht automatisch größenoptimiert gezeichnet, sondern nur dann in der Größe angepaßt, wenn das Constraint-Netzwerk ungültig ist.

Durch Hinzufügen weiterer Constraints könnte man erreichen, daß der Zustandsname grundsätzlich in der Mitte des Rechtecks steht. Weiterhin könnte man auch erreichen, daß das Rechteck die kleinstmögliche Größe hat. Dadurch gibt es immer weniger Layout-Freiheiten, bis zuletzt das Layout eines Zustands nur noch durch seine Position bestimmt ist.

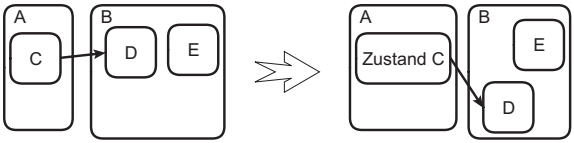
Das Constraint-Netzwerk hat durch die zusätzlichen Anforderungen nur noch eine Lösung, wäre also nicht mehr unterspezifiziert. Anhand der Textgröße und des Abstandes zum Rahmen könnten die Layoutparameter somit auch durch Attributberechnungen ermittelt werden. Das Constraint-Netzwerk für das visuelle Programm könnte somit vereinfacht und die Berechnung der graphischen Darstellung insgesamt beschleunigt werden.

Das ist hier der Fall, weil durch zusätzliche Anforderungen an das Layout die Layoutfreiheiten eleminiert wurden. Bei der Schachtelung einzelner Zustände in einem XOR-Superstate muß ebenfalls korrekte Schachtelung geprüft werden. Dabei ist es aber wenig sinnvoll, das Layout durch zusätzliche Anforderungen eindeutig zu machen. Der Einsatz des Constraint-basierten statt des manuell unterstützen Layouts führt dabei zu effizienter benutzbaren Editoren. Das zeigt genauer der nächste Abschnitt.

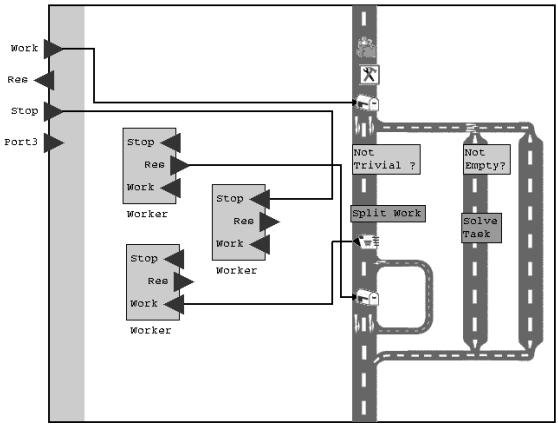
4.4.2.2. Kombinierte Layoutberechnung

Der vorige Abschnitt hat gezeigt, daß das Constraintbasierte Layout sich gut eignet, um Sprachkonstrukte mit Layoutfreiheiten anzuordnen. Die Layoutberechnungen des Attributauswerters sind dagegen effizienter, wenn das Layout eindeutig ist. Das ist beispielsweise oft der Fall, wenn man kompakte Darstellung oder einfache Linienführung bei orthogonalen Linien statt des Prinzips der geringstmöglichen Layoutänderung fordert. In diesem Abschnitt wird gezeigt, daß sich beide Techniken für unterschiedliche Sprachkonstrukte auch gut kombinieren lassen.

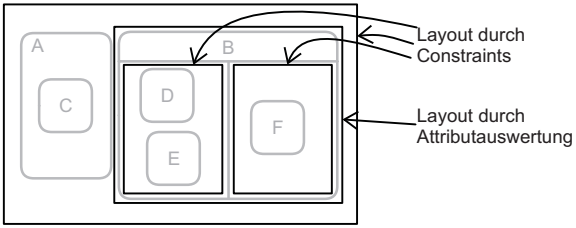
Abbildung 4.16 enthält einige Anwendungsfälle für kombinierte Layoutberechnungen. In der Sprache der Zustandsdiagramme wurde beispielsweise die Anordnung der einfachen Zustände und der Übergänge mit Hilfe des Attributauswerters berechnet. Für die Positionen der Zustände in den XOR-Superstates und für die Größen der XOR-Superstates wurden dagegen persistente Attribute eingeführt und mit Constraints Überlappungsfreiheit und korrekte Schachtelung sichergestellt. Abbildung 4.16(a) zeigt noch einmal den Bearbeitungskomfort, der durch diesen Ansatz erreicht wird. Im Bild wurde lediglich der Name des Zu-



(a) Zustandsdiagramm mit XOR-Superstates



(b) Streets



(c) Zustandsdiagramm mit AND-Superstate

Abbildung 4.16.: Kombination der Layoutverfahren

stands C geändert. Alle Layoutänderungen wurden vom Constraint-Solver veranlaßt.

Abbildung 4.16(b) zeigt ein komplexeres Anwendungsbeispiel in der Sprache Streets. Im Bild wurde die Überlappungsfreiheit der Worker-Prozesse durch Constraints sichergestellt. Die restliche Anordnung hat keine Layoutfreiheiten und wurde durch einen Attributauswerter ermittelt. Insbesondere das automatisch erfolgende Routing der Linien vereinfacht dabei die Benutzung des Editors.

In Abbildung 4.16(c) ist dagegen ein Anwendungsfall dargestellt, in dem sich die Layoutverfahren nicht ganz reibungslos kombinieren lassen. Das Beispiel ist ebenfalls der Sprache der Zustandsdiagramme entnommen. Hier wurden jedoch die AND-Superstates hinzugefügt und entschieden, daß das Layout der AND-Superstates durch den Attributauswerter berechnet werden soll.

Die Komplikation entsteht dadurch, daß durch die Kombination der Layoutverfahren das Constraint-Netzwerk in mehrere Teile zerfällt. Um die Constraints für den äußeren Bildbereich zu ermitteln, muß die Größe des AND-Superstates bekannt sein. Um diese zu berechnen, muß man jedoch die Ausdehung der inneren Constraint-basiert angeordneten Bereiche kennen, müßte also den Constraint-Solver aufrufen.

Durch die Aufteilung der Constraints entstehen unerwünschte Effekte: die im ersten Constraint-Solver-Lauf getroffenen Entscheidungen lassen sich in späteren Läufen nicht mehr rückgängig machen. Im Beispiel heißt das, daß sich die Unterbereiche der AND-Superstates zwar automatisch vergrößern (beispielsweise bei geigneter Umbenennung des Zustands E), jedoch nicht automatisch verkleinern (beispielsweise bei geigneter Umbenennung des Zustands C).

Auch in diesem Anwendungsbeispiel kann man mit nur einem Aufruf des Constraint-Solvers auskommen, wenn man die Größen der Unterbereiche der AND-Superstates als für den Constraint-Solver nicht änderbar betrachtet. Der Vorteil, den man dadurch erzielt, ist ein Effizienzgewinn bei der Erzeugung der graphischen Darstellung. Anstatt je nach Struktur des visuellen Programms den Constraint-Solver sehr oft aufzurufen, reicht nun ein einziger Aufruf. Der Nachteil dieses Verfahrens ist, daß die Größen der Unterbereiche der AND-Superstates hier nicht automatisch angepaßt werden.

4.4.2.3. Auswahl von Parcon

Zur Auswahl des Constraint-Solvers Parcon [Griebel u.a. 1997, 1996; Griebel 1996] haben Schmidt und Schindler [2000] im Rahmen ihrer Diplomarbeit eine Anforderungsanalyse durchgeführt. Wichtige Kriterien waren dabei zum ersten spezielle Anforderungen, die zur Formulierung graphischer Constraints für zweidimensionale Graphiken benötigt werden, zum zweiten die inkrementelle Änderbarkeit der Constraint-Netzwerke zur Reduktion der Kommunikationslast zwi-

schen Constraint-Solver und Attributauswerter und zum dritten die Effizienz des Solvers.

Um graphische Constraints zu formulieren, werden einige spezielle Ausdrucksmittel benötigt, die von Constraint-Solvern für andere Aufgabengebiete nicht unbedingt zur Verfügung gestellt werden. Dazu gehört beispielsweise, daß einzelne Constraints nicht nur durch Konjunktionen, sondern auch durch Disjunktionen miteinander verknüpft werden können. Dies ist beispielsweise für die Nichtüberlappung von Rechtecken notwendig, bei der man formuliert: Rechteck1 ist links, rechts, ober- *oder* unterhalb von Rechteck2. Ein anderer spezieller Gleichungstyp ist die euklidische Entfernung, die beispielsweise bei der Nichtüberlappung von Kreisen benötigt wird. Schließlich ist auch die Unterstützung des least astonishment Prinzips wichtig.

Der Constraint-Solver Parcon ist für derartige Constraints entwickelt worden. Er unterstützt ungerichtete, zyklische Constraints, die Bildung von Hierarchien und das Prinzip der geringstmöglichen Änderung für unterspezifizierte Constraint-Netzwerke.

Ein weiterer wichtiger Punkt ist die Effizienz des Constraint-Solvers. Effizienzprobleme werden am ehesten bei der Auswertung von Disjunktionen sichtbar. Beispielsweise explodiert bei der Sicherstellung der Nichtüberlappung vieler Rechtecke der Lösungsraum. Parcon verfügt hier über “spezielle Optimierungen”, die das Verfahren beschleunigen. Außerdem existiert Parcon auch in einer parallelen Variante, die zur Beschleunigung in diesem Anwendungsfall benutzt werden kann.

Aus einer Reihe von Gründen stand die Auswahl von Parcon schon sehr früh fest. Zum einen fehlen anderen Constraint-Solvern wichtige Eigenschaften: DeltaBlue [Freeman-Benson u.a. 1990] kann beispielsweise keine zyklischen Constraintnetzwerke lösen, Garnet [Myers u.a. 1990] beherrscht ausschließlich gerichtete Constraints. Zum anderen ist Parcon, verglichen mit anderen Solvern, sehr schnell. Ausschlaggebend war schließlich auch, daß Parcon ebenfalls an der Universität Paderborn entwickelt und gewartet wird.

4.4.2.4. Einbindung des Constraint-Solvers

Parcon ist als separates Anwendungsprogramm realisiert. Daher ist die Schnittstelle zwischen Constraint-Solver und generierten Attributauswertern eine bidirektionale Interprozeßkommunikation. Für jedes offene Fenster[4] wird der Constraint-Solver einmal gestartet. Zur Erzeugung der graphischen Darstellung führt der Attributauswerter zunächst einige Layoutberechnungen aus und generiert ein Constraint-Netzwerk, welches textuell in Parcon-Notation [Griebel 1996,

[4] genauer: für jede Sicht

Anhang A] repräsentiert wird. Dieses Constraint-System[5] wird an den Solver übermittelt, der daraufhin mit Layoutänderungen antwortet. Die Layoutänderungen beziehen sich auf die Variablen des Constraint-Netzwerks und bestehen aus deren Name und dem neuen Wert. Diese werden vom Attributauswerter entgegengenommen, der daraufhin weitere Layoutberechnungen durchführen und die graphische Darstellung erzeugen kann.

Beide Werkzeuge, Attributauswerter und Constraint-Solver müssen auf die Layoutinformationen zugreifen, die in der Programmrepräsentation in persistenten Attributen gespeichert werden. Damit die Antworten des Constraint-Solvers in Wertänderungen persistenter Attribute umgesetzt werden können, muß der Name einer Layoutvariable des Constraint-Netzwerks auf die Adresse des diese Layoutinformation speichernden Attributs abbildbar sein. Im einfachsten Fall reicht es, als Name einer Layoutangabe eine textuelle Repräsentation der Adresse des Attributs zu wählen und eine Kennung voranzustellen, die den Attributtyp bezeichnet.

Im einzelnen gestaltet sich der Aufruf des Constraint-Solvers aus der Sicht des Attributauswerters wie folgt:

Berechne die Eingabe des Constraint-Solvers. Das ist ein Constraint-Netzwerk in Parcon-Notation und ein Satz von Anweisungen, die die Werte der Layoutvariablen setzen. Da das Constraint-Netzwerk in textueller Form benötigt wird, ist dies eine Aufgabe, wie sie auch bei der Codeerzeugung in Source-to-Source-Übersetzern anfällt. Hierfür wurde das im Eli-System integrierte Werkzeug PTG [Ptg 2000] benutzt.

Rufe Parcon auf. Parcon prüft daraufhin die spezifizierten Constraints und ändert ggf. einige Layoutparameter. Es sendet Tupel zum Attributauswerter, die aus dem Namen des Layoutparameters und dessen neuem Wert bestehen. Aus dem Namen kann die Adresse eines persistenten Attributs extrahiert werden und in diesem der geänderte Wert gespeichert werden. Aus der Sicht des Attributauswerters hat der Aufruf von Parcon Seiteneffekte: Es ändern sich die Werte einiger persistenter Attribute.

Erzeuge die graphische Darstellung. Dabei werden die neuen Werte der Layoutparameter benötigt. Um sicherzustellen, daß der erzeugte Attributauswerter die Werte erst nach dem Aufruf des Constraint-Solvers ausliest, führt man in der attributierten Grammatik zusätzliche Abhängigkeiten ein.

Die Erzeugung des Formelsystems hat einige zusätzliche Schwierigkeiten, die in Source-To-Source-Übersetzern nicht anfallen. Ein erster Aspekt ist die Eigenschaft eines Constraint-Solvers, die letzte Editieroperation des Benutzers wenn möglich nicht rückgängig zu machen, solange das Constraint-System sich auch

[5] eigentlich die Änderungen

auf andere Weise erfüllen läßt. Dem Solver müssen dazu auch Angaben über die zuletzt vom Benutzer geänderten Layoutangaben übermittelt werden. Diese Angaben sind im Strukturbaum zunächst nicht enthalten. Sie können bereitgestellt werden, indem Änderungen an persistenten Variablen, die in der Folge einer Editieroperation durchgeführt werden, in einer Liste gespeichert werden. Die Liste wird nach Aufruf des Constraint-Solvers zurückgesetzt.

Ein anderer Aspekt ist, daß einige Constraint-Solver, so auch Parcon, inkrementelle Änderungen des Formel-Systems unterstützen. Dadurch wird einerseits die Kommunikationslast zwischen Constraint-Solver und Anwendungsprogramm reduziert. Andererseits ermöglicht dies einem Constraint-Solver, das Formelsystem so zu optimieren, daß nachfolgende Anfragen schneller beantwortet werden können. Ein Beispiel führen Cournarie und Beaudouin-Lafon [1995] für den Constraint-Solver Alien an: Die erste Berechnung eines 50-fach geschachtelten Quadliterals - ein Constraint-Benchmark - erfordert dort 11,0 Sekunden, nachfolgende Berechnungen dauern nur noch 0,1 Sekunden.

Beim Constraint-Solver Parcon spielt es von der Laufzeit her keine Rolle, ob ein Constraint-System zum ersten Mal oder wiederholt ausgewertet wird [Griebel 1996, Abschnitt 6.7]. Trotzdem ist es sinnvoll, die Möglichkeit der inkrementellen Änderungen zu nutzen, da die Schnittstelle von Parcon zum Anwendungsprogramm textuell ist und auf Prozeßkommunikation basiert.

Mit den nichtinkrementellen Attributauswertern, die das Liga-System generiert, können die Änderungen des Formelsystems nicht direkt erzeugt werden. Statt dessen kann man bei der Erstellung der attributierten Grammatik die inkrementelle Interaktion mit dem Constraint-Solver zunächst ignorieren und die Erzeugung des vollständigen Constraint-Systems spezifizieren. Um vollständige Constraint-Netzwerke in Änderungen umzurechnen, wird zwischen dem Constraint-Solver und dem Attributauswerter ein weiteres Modul benötigt, das jeweils das letzte Constraint-System speichert. Hierzu wurde ein assoziatives Array benutzt, das mit Formeln indiziert wird und in dem die Gleichungsnummern gespeichert werden, die Parcon den Formeln zuordnet. Da der Zugriff auf das assoziative Array mit Hilfe einer Hash-Tabelle implementiert wurde, kann man in der Praxis von einer konstanten Zeit für den Zugriff ausgehen. Damit benötigt der Vergleich zweier Constraint-Systeme insgesamt lineare Zeit (in der Anzahl der Formeln).

Würden hier inkrementelle Attributauswerter verwendet, so könnte man auf ein solches Modul verzichten. Statt dessen geschähe der Vergleich der Constraint-Systeme dabei implizit durch die Algorithmen, die geänderte Attributwerte ermitteln. Dieses Verfahren benötigt aber ebenfalls Zeit und erfordert einen gewissen zusätzlichen Verwaltungsaufwand bei den Attributberechnungen und bei der Speicherung des Strukturbaums. Ein weiterer Aspekt ist, daß bei Benutzung nichtinkrementeller Attributauswerter die Layoutänderungen des Constraint-Solvers durch Seiteneffekte realisiert werden können, die durch zusätz-

liche Abhängigkeiten beschrieben sind. Diese Technik ist jedoch bei inkrementellen Attributauswertern nicht anwendbar. Hier müßten die Rückmeldungen des Constraint-Solvers in einer Datenstruktur gespeichert werden. Diese Informationen müßten dann an alle Kontexte mit Layoutattributen verteilt werden.

Beim Aufruf des Constraint-Solvers ist der einzige potentielle Vorteil eines inkrementellen Attributauswerters ein Geschwindigkeitsgewinn bei großen visuellen Programmen. Beim Vergleich der Effizienz ist aber zu berücksichtigen, daß der Solver hier u.a. dazu benötigt wird, um Überlappungsfreiheit zu gewährleisten. Sofern man Rechtecke nicht durch Kreise annähert, kann man ihre Überlappungsfreiheit nur durch Disjunktionen beschreiben. Diese erfordern zur Durchsuchung des Lösungsraums aber einen beträchtlichen Aufwand [Griebel 1996, Abschnitt A.12]. Die Verarbeitungsgeschwindigkeit des Attributauswerters ist demgegenüber vernachlässigbar.

Die verwendete Methode des Aufrufs bedeutet für den Sprachentwickler, daß die inkrementelle Natur von Parcon nicht berücksichtigt zu werden braucht. Da zur Ermittlung der Änderungen des Constraint-Systems zwei aufeinanderfolgende Systeme miteinander verglichen werden, kann für eine graphische Darstellung der Constraint-Solver zunächst aber nur einmal aufgerufen werden. Das ist in der Praxis keine große Einschränkung.

Möchte man sie dennoch umgehen, so müßte man verschiedenen Aufrufen des Constraint-Solvers Namen zugeordnen. Weiterhin müßte man die Seiteneffekte eines Aufrufs des Constraint-Solvers sorgfältiger durch Attributabhängigkeiten modellieren.

4.4.3. Weitere Attributberechnungen

Die Berechnung der Anordnung der graphischen Darstellung ist zwar eine wichtige, aber nicht die einzige Aufgabe bei der Erzeugung der graphischen Darstellung. Eine weitere Aufgabe ist die Ermittlung der Informationen, die in der graphischen Darstellung visualisiert werden sollen. Diese sind ggf. nicht direkt in der Programmrepräsentation gespeichert, sondern werden aus anderen Informationen des Programms ermittelt, um Inkonsistenzen zu vermeiden. Die Attributberechnungen, die dazu erforderlich sind, transportieren oft Informationen von einer Definitionsstelle zu den Anwendungsstellen. Manchmal müssen aber auch Informationen über das Programm ermittelt werden. Für beide Aufgaben lassen sich gut attributierte Grammatiken verwenden. In diesem Abschnitt nenne ich Beispiele dafür.

In der Sprache Forms/3 können Programme zur Manipulation von Vektoren und Matrizen erstellt werden. Dabei werden Positionen innerhalb der Vektoren durch Schraffuren markiert, die an anderer Stelle in Formeln wiederverwendet werden, um einen Bezug zum Inhalt des Vektors an der definierten Stelle herzu-

stellen, siehe Abbildung 4.17. Ändert der Benutzer in einem solchen Programm die Schraffur einer Vektorposition, so ist es wünschenswert, daß die Schraffur an allen Anwendungsstellen dieser Schraffur mitgeändert wird. Das kann erreicht werden, indem etwa die Schraffur nur einmal an der Definitionsstelle gespeichert und vor Erzeugung der graphischen Darstellung von der Definitionsstelle zu den Anwendungsstellen transportiert wird.

Im weiteren könnte ein Sprachentwerfer wünschen, daß die Schraffuren in einem Gültigkeitsbereich automatisch bestimmt werden, wodurch der Benutzer davon entbunden werden würde, selbst eine eindeutige Schraffur zu wählen. Auch diese Aufgabe läßt sich mit attributierten Grammatiken spezifizieren.

Ähnliche Aufgaben fallen auch bei anderen Relationen an, die nicht durch Berührung dargestellt werden. Bei Streets können die Datenflußverbindungen beispielsweise alternativ durch Zahlen gekennzeichnet werden, die an den Enden einer Datenflußbeziehung gemeinsam verwendet werden, siehe z.B. Abbildung 3.15(b) auf Seite 87. Die dabei benutzten Zahlen können durch einen Attributauswerter ermittelt werden.

Ein weiteres Beispiel für Attributberechnungen, deren Ergebnisse in die graphische Darstellung visueller Programme eingeht, liefert die Sprache LabView. Um die Typen der Datenflußlinien festzulegen, wird hier Typinferenz benutzt, siehe Abbildung 4.18. Bei der Konstruktion beider Linien kann erst bei der Verbindung mit der Datenquelle (innerste Konstrukte) entschieden werden, welche Typen den Datenflußlinien zugeordnet werden müssen. Bei den oberen Linien ist dies etwa STRING, ARRAY OF STRING, ARRAY OF ARRAY OF STRING, ..., während sich bei der unteren BOOLEAN, ARRAY OF BOOLEAN, ARRAY OF AR-

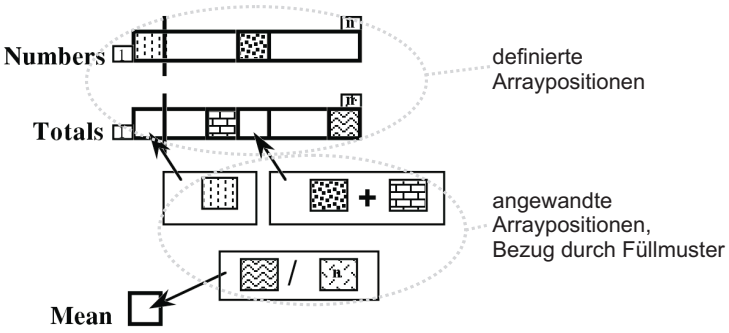


Abbildung 4.17.: Bezug auf Arraypositionen in Forms/3 nach Pandey und Burnett [1993]

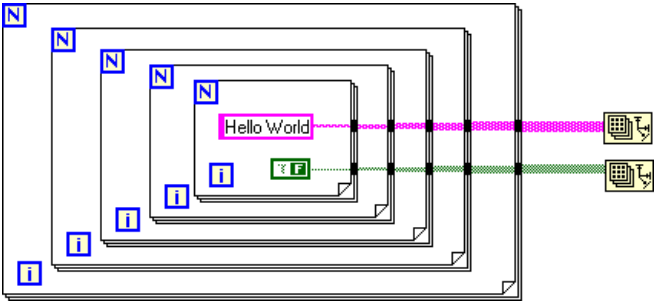


Abbildung 4.18.: Typinferenz in LabView-Programmen

RAY OF BOOLEAN, ...einstellt. Der Datentyp einer Linienverbindung wird in LabView durch Linienstärke und -farbe visualisiert.

Eine Alternative zur Typinferenz bei Labview wäre, die Typen der beteiligten 12 Liniensegmente jeweils einzeln festzulegen, was den Aufwand der Programmkonstruktion enorm vergrößern würde. Auch Burnett u.a. [1995] beschreiben, daß Typinferenz im allgemeinen eine gute Lösung für die Umsetzung der statischen Typisierung in visuellen Sprachen ist. Reps und Teitelbaum [1989a, Abschnitt 7.2] stellen dar, wie Typinferenz in Strukturediteren für textuelle Sprachen mit attributierten Grammatiken spezifiziert werden kann.

Schließlich werden Attributberechnungen nicht nur für die Erzeugung der graphischen Darstellung, sondern auch für die Spezifikation der Editieroperationen benötigt. Wenn komplexere Editieroperationen realisiert werden müssen, treten dabei auch komplexere Attributberechnungen auf. Auf diesen Aspekt geht der folgende Abschnitt ein.

4.5. Editieren visueller Programme

Ein Editierschritt des Strukturediteors besteht aus drei Phasen. In der ersten Phase wird die graphische Darstellung der Fenster aus der Programmrepräsentation erzeugt oder aktualisiert. Anschließend kann ein Benutzer mit der Darstellung interagieren. Dazu müssen die anwendbaren Interaktionen beschrieben werden. Hier wird beispielsweise angegeben, an welchen Stellen der Darstellung sich Sprachkonstrukte einfügen lassen oder welche Sprachkonstrukte sich selektieren und anschließend verschieben oder löschen lassen. Hat ein Benutzer eine Interaktion “eingegeben”, so muß diese in der dritten Phase durch eine Änderung der Programmrepräsentation realisiert werden. Darauf schließt sich wieder die erste Phase an, in der die graphische Darstellung der offenen Fenster aktualisiert wird.

In der graphischen Darstellung eines Fensters sind alle notwendigen Informationen enthalten, um alle drei Teile einer Editieroperation durchzuführen. Sie enthält dazu außer den sichtbaren Teilen noch einige "unsichtbare" Angaben. Diese bestimmen, wie der Benutzer mit der Darstellung interagieren kann und enthalten auch die Informationen, um eine Editieroperation in der Programmrepräsentation zu realisieren. Der vorige Abschnitt hat beschrieben, wie die sichtbaren Teile der Darstellung erzeugt werden. In diesem Abschnitt beschreibe ich die unsichtbaren Teile.

Auch die unsichtbaren Teile der Darstellung werden durch den Attributauswerter berechnet. Dabei sind selten komplexe Berechnungen erforderlich. Dagegen muß oft in einem bestimmten Kontext des Strukturbaums eine bestimmte Prozedur aufgerufen werden. Die benötigten Angaben sind in großen Maße von den im Editorrahmen[6] angebotenen Interaktionen abhängig. Der im Rahmen dieser Arbeit erstellte Editorrahmen basiert auf den Möglichkeiten von Tcl/Tk [Welch 1997]. Er ist so konzipiert, daß möglichst wenig Angaben zu Editieroperationen erforderlich sind. Beispielsweise ist jedem Objekt der graphischen Darstellung ein Kontext des Strukturbaums zugeordnet. Mit Hilfe dieser Information lassen sich alle graphischen Objekte identifizieren, die zu einem Teilbaum gehören. So können Teile des Strukturbaums in der graphischen Darstellung selektiert und dadurch Editieroperationen wie "Löschen", "Ausschneiden", "öffnen von untergeordneten Fenstern", usw. realisiert werden. An wichtigen Stellen hat ein Sprachentwerfer aber weitere Einflußmöglichkeiten, so daß sich auch spezielle Wünsche realisieren lassen.

In den folgenden drei Abschnitten gehe ich auf einige Interaktionstechniken näher ein. Der erste Abschnitt stellt das Konzept der Einfügemarken vor. Diese liefern die benötigten Informationen, um das Einfügen und das Verschieben von Sprachkonstrukten zu realisieren. Der zweite Abschnitt beschreibt, wie die Technik Drag&Drop benutzt werden kann, um Beziehungen zwischen den Konstrukten einer visuellen Sprache zu erzeugen bzw. zu ändern. Der dritte Abschnitt liefert ein Beispiel dafür, wie sich spezielle Editieroperationen in einer visuellen Sprache realisieren lassen.

Auf einige Editieroperationen werde ich in diesem Abschnitt nicht weiter eingehen. Das betrifft z.B. Interaktionen zur Größenänderung oder Dialoge, mit denen die Werte von Attributen editiert werden können. Auch diese werden zur Implementierung visueller Sprachen benötigt. Ihre Umsetzung basiert im wesentlichen auf Konzepten und Methoden, auf die bereits bei anderen Interaktionstechniken eingegangen wird.

[6] Unter "Editorrahmen" verstehe ich die festen, nicht generierten Teile der Sprachimplementierung, das sind i.W. Module zur Implementierung der Strukturediteren.

4.5.1. Einfügemarken

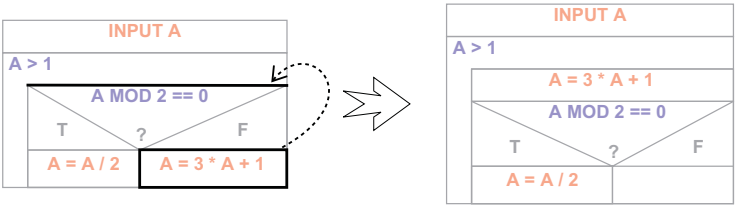
Um “Einfügen” und “Verschieben” zu realisieren, muß ein Benutzer die Stelle festlegen, an die ein neues Sprachkonstrukt eingefügt oder an die ein Konstrukt verschoben werden soll. Hierfür können Einfügemarken wie andere graphische Objekte in die Darstellung integriert werden. Einfügemarken sind zunächst unsichtbar. Verschiebt ein Anwender ein Sprachkonstrukt oder zieht er ein neues Konstrukt aus der Werkzeugleiste (siehe Abschnitt 4.3.2 auf Seite 115) auf den Arbeitsbereich, so wird jeweils die dem Mauscursor am nächsten befindliche Marke sichtbar. Damit kann ein Benutzer die Stelle in der Darstellung auswählen, an die verschoben bzw. in die eingefügt werden soll. In Abbildung 4.19(a) wird so der Einfügekontext durch eine Linie, in Abbildung 4.19(b) durch ein Rechteck markiert.

Einer Einfügemarke ist im weiteren ein Kontext des Strukturbaums zugeordnet. Wählt ein Benutzer für eine Editieroperation eine Marke aus, so bestimmt er damit gleichzeitig auch den Kontext im Strukturbaum, an dem die Operation realisiert werden soll. Über diesen Kontext lassen sich auch die dort anwendbaren Produktionen bestimmen. Diese Information wird verwendet, um nur die legalen Einfügestellen für das neue oder verschobene Konstrukt auswählbar zu machen. Das geht soweit, daß nicht einfügbare Sprachkonstrukte in der Werkzeugleiste automatisch deaktiviert dargestellt werden.

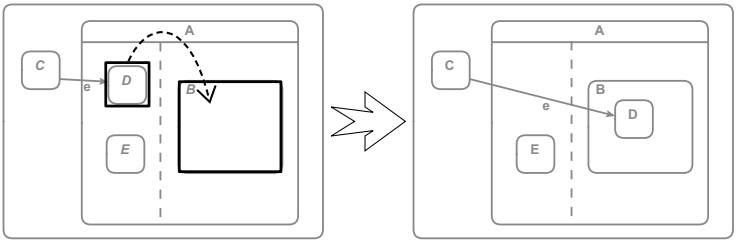
Für verschiedene Layoutkonzepte stehen Einfügemarken mit und ohne Positionswahl zur Verfügung. Die Einfügemarken ohne Positionswahl werden für Sprachkonstrukte verwendet, deren Layout ohne Einflußmöglichkeiten des Benutzers berechnet wird. Hier muß lediglich der Einfügekontext, beispielsweise durch eine Linie markiert werden. Abbildung 4.19(a) zeigt eine Verschiebeoperation in der Sprache der Nassi-Shneiderman-Diagramme. Die ausgewählte Einfügemarke ist dabei durch Aufhellung der restlichen Darstellung hervorgehoben.

Bei den Einfügemarken mit Positionswahl kann der Benutzer zusätzlich die Position spezifizieren, an die ein Konstrukt verschoben bzw. bei der es eingefügt werden soll. Damit läßt sich diese Sorte von Einfügemarken für Sprachkonstrukte anwenden, deren Layout eine vom Benutzer stammende Position erfordert. Abbildung 4.19(b) zeigt eine Verschiebeoperation in der Sprache der Zustandsdiagramme. Der ausgewählte Einfügekontext wird durch ein Rechteck hervorgehoben. Die Einfügeposition ist durch die Koordinaten des Mauscursors beim Loslassen der Maustaste bestimmt.

Ein weiterer Aspekt bei Einfügemarken ist deren Auswahl. Es reicht oft nicht allein, eine Marke anhand der Entfernung zum Mauscursor auszuwählen, weil es vorkommen kann, daß sich mehrere Einfügemarken für unterschiedliche Kontexte an denselben Bildschirmkoordinaten befinden. In Abbildung 4.19(a) sind dies die Einfügemarken nach dem Then- und dem Else-Zweig der bedingten Anweisung sowie die nach der bedingten Anweisung. Eine Lösung dafür ist beispielswei-



(a) Verschiebeoperation bei Einfügemarke ohne Positionswahl



(b) Verschiebeoperation bei Einfügemarke mit Positionswahl

Abbildung 4.19.: Kennzeichnung von Kontexten in Strukturbaum und graphischer Darstellung durch Einfügemarken

se, die Marken mit minimalem Abstand zyklisch nacheinander hervorzuheben, so daß der Benutzer durch Loslassen der Maustaste im richtigen Moment die intendierte Einfüge- bzw. Verschiebeoperation realisieren kann.

Auch Einfügemarken mit Positionswahl können einander teilweise überlappen. In Abbildung 4.19(b) sind dem rechten Teil von Zustand A sowie dem Zustand B jeweils Einfügemarken mit Positionswahl zugeordnet. Damit bei Einfüge- und Verschiebeoperationen eine Marke ausgewählt werden kann, sind den Einfügemarken Prioritäten zugeordnet. Die Einfügemarke für den Zustand B hat eine höhere Priorität als diejenigen, die Zustand A zugeordnet sind. Dies drückt den Sachverhalt aus, daß Zustand B in A enthalten ist und bewirkt hier, daß der innere Kontext bei Einfüge- und Verschiebeoperationen bevorzugt wird.

4.5.2. Drag&Drop

Um nichthierarchische Beziehungen zu editieren, bietet der Editorrahmen verschiedene Interaktionstechniken an. Diejenige, die das Prinzip der direkten Ma-

nipulation [Shneiderman 1983] am besten umsetzt, ist Drag&Drop. Diese Interaktionstechnik ist sowohl innerhalb eines Fensters, als auch zwischen mehreren Fenstern anwendbar. Innerhalb eines Fensters wird sie meistens verwendet, um Sprachkonstrukte zu erzeugen, die durch Linien dargestellt werden und Beziehungen zu anderen Sprachkonstrukten aufweisen. Darauf geht der folgende Abschnitt genauer ein.

Drag&Drop zwischen mehreren Fenstern ist nützlich, um beispielsweise Anwendungen einer Definition einzufügen. Definiert-Benutzt-Beziehungen werden oft durch gemeinsame Verwendung von Merkmalen wie Namen, Farben und Füllmuster statt durch Linien visualisiert. Diese Methode der Darstellung ist nicht darauf angewiesen, daß die in Beziehung stehenden Programmkonstrukte in einem Fenster enthalten sind.

Ein Beispiel dafür enthält Abschnitt 4.6.1.2. In der Abfragesprache QBE werden die benutzten Datenbanktabellen durch Ausgabe ihrer Namen und ihrer Struktur gekennzeichnet. Um in der QBE-Abfrage aus Abbildung 4.23 (Seite 155) eine weitere Abfragetabelle (Anwendung) zu einer Tabelle einzufügen, kann die Tabellendefinition aus dem oberen Fenster in eines der unteren Fenster gezogen werden.

4.5.2.1. Drag&Drop innerhalb eines Fensters

Bei der Drag&Drop-Interaktion werden zwei verschiedene Stellen in der graphischen Darstellung angegeben, beispielsweise um zwischen diesen Stellen eine Linie zu ziehen. Bei der Berechnung der graphischen Darstellung werden die Stellen, an denen Drag&Drop angewendet werden kann, durch Drag-Marken bzw. durch Drop-Marken gekennzeichnet. Diesen Marken sind wie bei den Einfügemarken die jeweils zugehörigen Kontexte im Strukturbaum zugeordnet. Zieht der Benutzer eine Drag-Marke auf eine Drop-Marke, so wird eine spezifizierbare Prozedur aufgerufen. Diese Prozedur kann je nach den Anforderungen einer visuellen Sprache Konstrukte einfügen und Beziehungen zwischen den angegebenen Programmstellen erzeugen.

Ein Beispiel für die Anwendung dieser Technik ergibt sich bei den Petri-Netzen. Um die Verbindungen zwischen den Stellen und den Transitionen zu definieren, sind zwei Strategien möglich, die in den folgenden Absätzen beschrieben werden. Die erste benötigt drei Interaktionen, um eine Linie zu definieren, die zweite lediglich eine.

Bei der ersten Strategie wird zunächst eine neue Verbindung in die Darstellung eines Petri-Netzes eingefügt. Die dann noch unverbundenen Enden der Linie werden anschließend einzeln durch Drag&Drop mit Stellen (oder Transitionen) des Petri-Netzes verbunden. Abbildung 4.20 verdeutlicht dies am Beispiel der Petri-Netze. In der graphischen Darstellung werden an den Enden von Linien jeweils Drag-Marken plaziert, an den Stellen und Transitionen des Petri-Netzes Drop-Marken. Die Drag- und Drop-Marken enthalten wie die Einfügemarken

die zur Realisierung der Editieroperation notwendigen Informationen und sind wie sie zunächst unsichtbar. Während der Interaktion werden sie sichtbar, um eine Rückmeldung an den Benutzer zu geben. Das zweite und das dritte Teilbild von Abbildung 4.20 enthält sichtbar gewordene Drag- und Drop-Marken. Die gestrichtelten grauen Pfeile kennzeichnen die ausgeführte Drag&Drop-Operation.

Die Editieroperation des Benutzers wird realisiert, indem der Eintrag in der Definitionstabelle, der zur Stelle (bzw. zur Transition) gehört, in einem Attribut der Verbindung gespeichert wird. Dadurch wird eine Beziehungen zwischen diesen zwei Programmstellen erzeugt. Die zu der graphischen Darstellung gehörende Programmrepräsentation ist jeweils rechts von denFenstern in Abbildung 4.20 skizziert.

Bei der zweiten Strategie wird die Drag&Drop-Interaktion direkt zum Ziehen einer Linie benutzt, ohne vorher eine Verbindung zu erzeugen. Man kürzt dadurch einige Interaktionen ab und kann in der Werkzeugleiste auf die Verbindungsproduktion verzichten. Die Drag- und Drop-Marken werden dazu bei den Stellen und Transitionen erzeugt: Jede Stelle und jede Transition erhält sowohl eine Drag-Marke als auch eine Drop-Marke. Hat ein Benutzer eine Stelle zu einer Transition (oder umgekehrt) gezogen, so wird diese Editieroperation realisiert, indem zunächst eine neue Verbindung erzeugt wird. Dieser Verbindung werden anschließend die Definitionstabellen-Einträge der verbundenen Programmkonstrukte zugeordnet.

Beide Strategien kommen ohne Moduswechsel aus, d.h. die einzelnen Interationen können beliebig mit anderen Interaktionen kombiniert werden. Die erste Methode hat den Vorteil, daß sie für verschiedene Linientypen funktioniert und auch die nachträgliche Bearbeitung einer Linie unterstützt. Hierzu wäre bei der zweiten Technik ein spezieller Modus je Linientyp erforderlich. Die zweite Technik führt dagegen zu einfacher benutzbaren Editoren.

Ein anderer Aspekt bei den Beziehungen einer visuellen Sprache sind Gültigkeitsregeln. Bei Petri-Netzen müssen Verbindungen beispielsweise jeweils eine Stelle und eine Transition miteinander verbinden. Verbindungen zwischen Stellen oder zwischen Transitionen sind nicht erlaubt. Ein Strukturedítor kann entweder durch eine Fehlermeldung auf illegale Verbindungen aufmerksam machen oder solche Verbindungen gar nicht erst zulassen. Beide Varianten werden von VLEli unterstützt.

Anders als bei den hierarchischen Sprachkonstrukten werden solche Gültigkeitsregeln aber nicht durch eine Grammatik bestimmt. Statt dessen werden die Gültigkeitsregeln durch die Berechnungen einer attributierten Grammatik sichergestellt. Bei den Petri-Netzen müssen dazu Informationen über die verbundenen Programmkonstrukte beschafft und in den Kontext der Verbindung transportiert werden. Dort kann zum einen eine Fehlermeldung erzeugt werden.

Zum anderen können den Drop-Marken auch Namen und den Drag-Marken

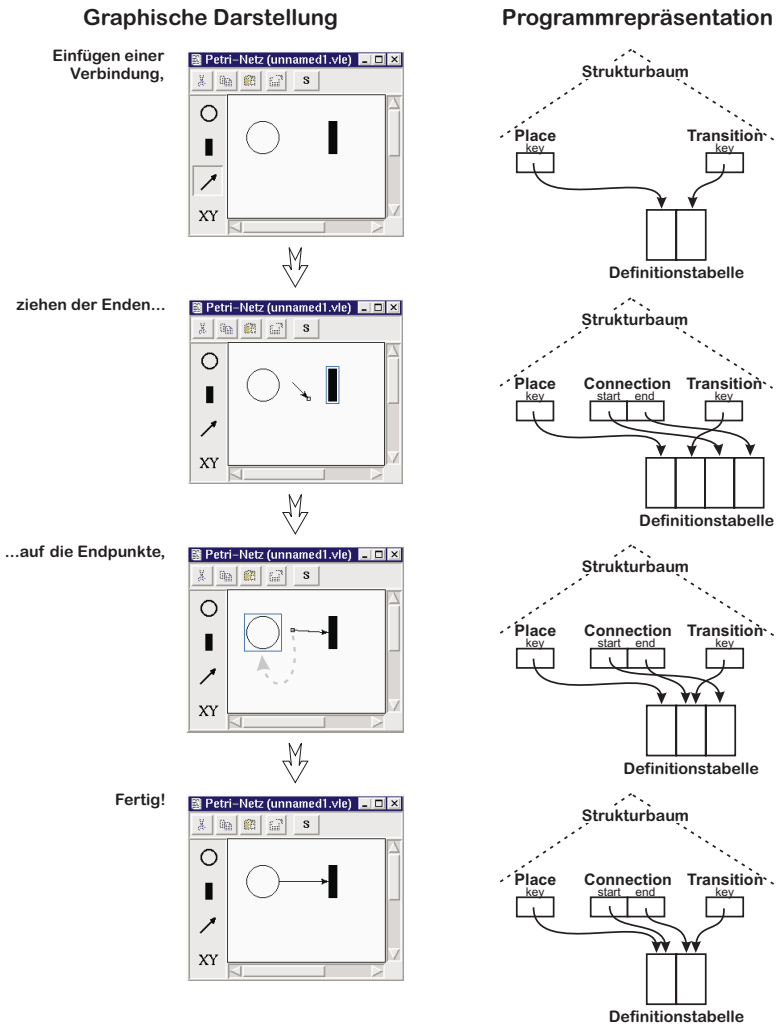


Abbildung 4.20.: Erzeugung von Verbindungen mit Drag&Drop in drei Schritten

Mengen von Namen zugewiesen werden. Zieht man dann eine Drag-Marke, so werden lediglich die erlaubten Drop-Marken angezeigt, das sind diejenigen, deren Namen in der Menge erlaubter Namen enthalten ist.

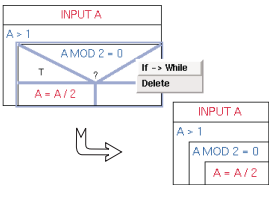
Mit den Berechnungen der attributierten Grammatik kann für die Drag-Marken der Linienendpunkte jeweils eine passende Menge berechnet werden. So hat die Drag-Marke für das linke Pfeilende im zweiten Teilbild von Abbildung 4.20 noch die Menge {Stelle, Transition}, während es im dritten Teilbild nur noch die Menge {Stelle} hat. So wird sichergestellt, daß keine illegalen Verbindungen erzeugt werden können.

4.5.3. Spezielle Editieroperationen

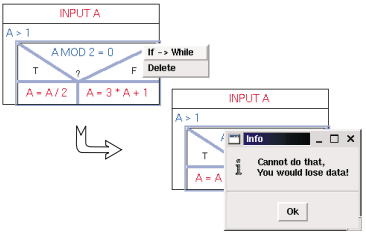
Die soeben vorgestellte Methode, mit Hilfe von Attributberechnungen die Anwendbarkeit von Editieroperationen zu prüfen, läßt sich auch allgemeiner anwenden. Um beispielsweise eine Editieroperation anzubieten, die eine If-Anweisung bei den Nassi-Shneiderman-Diagrammen in eine Schleife ändert, muß man etwa

```
SYMBOL Stmt: count: int;
SYMBOL Stmt COMPUTE
    SYNT.count = 1;
END;

CLASS SYMBOL CountStmts: count: int;
CLASS SYMBOL CountStmts COMPUTE
    SYNT.count =
        CONSTITUENTS Stmt.count
        WITH (int, ADD, IDENTICAL, ZERO);
END;

SYMBOL IfTrue INHERITS CountStmts END;
SYMBOL IfFalse INHERITS CountStmts END;

SYMBOL IfStmt: iftowhileok, iftowhilebranch: int;
SYMBOL IfStmt COMPUTE
    SYNT.iftowhileok =
        OR(EQ(CONSTITUENT IfTrue.count, 0),
           EQ(CONSTITUENT IfFalse.count, 0));

    SYNT.iftowhilebranch =
        IF (EQ(CONSTITUENT IfTrue.count, 0), 2, 1);

    IF (THIS.iftowhileok,
        VLContextedit(_currn, "If -> While",
            VLLIST3("::Change", "pWhileStmt",
                THIS.iftowhilebranch)),
        VLContextedit(_currn, "If -> While",
            VLLIST2("::CDlg::Information",
                "Cannot do that,\nYou would lose data!")));
END;

SYMBOL WhileStmt
COMPUTE
    VLContextedit(_currn, "While > If",
        VLLIST3("::Change","pIfStmt", 1));
END;
```

(a) Spezifikation



(b) Erfolgreiche Anwendung



(c) Warnmeldung

Abbildung 4.21.: If-To-While Transformation für Nassi-Shneiderman-Diagramme

sicherstellen, daß dabei keine Information verlorengeht. Dazu prüft man, ob der Then-Teil oder der Else-Teil der Anweisung leer ist. Ist dies der Fall, so kann beispielsweise ein Eintrag im Kontextmenü angeboten werden, mit dem diese Transformation durchgeführt werden kann. Abbildung 4.21(a) zeigt den Lido-Code, der diese Prüfung spezifiziert und die Einträge im Kontextmenü erzeugt. Daneben sind zwei Anwendungssituationen dargestellt. Bei der ersten wird die Transformation durchgeführt, bei der zweiten erscheint eine Warnmeldung.

Die If-To-While-Transformation ist eine tiefgreifende Änderung der Programmstruktur. Mir ist kein Indiz dafür bekannt, daß sie bei der Bearbeitung von Programmen von größerer Bedeutung ist. Trotzdem ist sie eine Editieroperation, die oft als Gegenbeispiel für Strukturediteren verwendet wird. Die Argumentation dabei ist, daß es in einem herkömmlichen Texteditor ein leichtes ist, das Wortsymbol `IF` durch das Wortsymbol `WHILE` auszutauschen, während bei Strukturediteren dazu in der Regel eine sehr viel längere Folge von Interaktionen benötigt wird [Minör 1992]. Dieses Gegenbeispiel läßt sich nicht direkt auf visuelle Strukturediteren übertragen, weil sich die graphische Darstellung der bedingten Anweisung und die der Schleife in einer visuellen Sprache wie den Nassi-Shneiderman-Diagrammen sehr viel weiter unterscheiden. Weiterhin wird die If-To-While-Transformation, z.B. von [Arefi u.a. 1990] dazu benutzt, die Ausdruckskraft von Generatoren für Strukturediteren zu demonstrieren.

Um die Baum-Transformation hier durchzuführen, wurde eine Tcl-Prozedur geschrieben und mit dem Eintrag im Kontextmenü verbunden. Die benötigte Tcl-Prozedur enthält 15 Codezeilen und ist nicht in Abbildung 4.21(a) enthalten. Die If-To-While-Transformation wurde also *nicht* durch Ausdrucksmittel von VLEli realisiert, vielmehr wurde der generierte Code an einer definierten Schnittstelle erweitert. Der Editorrahmen stellt bei vielen wichtigen Editieroperationen derartige Möglichkeiten zur Verfügung. Weitere Anwendungsbeispiele sind Koordinatentransformationen bei Einfüge- oder Verschiebeoperationen sowie die automatische Erzeugung von Linien bei der Drag&Drop-Interaktion.

4.6. Analyse- und Weiterverarbeitung

Bei der Implementierung einer visuellen Sprache werden neben einem spezialisierten Editor auch Analyse und Übersetzungswerkzeuge benötigt. Diese Werkzeuge dienen nicht nur der Weiterverarbeitung, sondern können darüberhinaus auch die Änderung visueller Programme erleichtern, beispielsweise indem Sie die Konsistenz zwischen Definitionen und ihren Anwendungen sicherstellen.

Die dabei anfallenden Aufgaben stellen sich zu einem großen Teil auch bei der Analyse und Weiterverarbeitung *textueller* Programme. Für diese Aufgabe sind umfangreiche, integrierte Werkzeugsammlungen entworfen und implementiert worden. Viele dieser Werkzeuge sind auch für die Analyse und Weiterverarbei-

tung *visueller* Sprachen anwendbar und erleichtern diese erheblich. Die Anwendbarkeit dieser Werkzeuge ist letztlich auch ein Grund dafür, daß visuelle Programme in dieser Arbeit durch Strukturbäume repräsentiert werden.

Der erste Abschnitt beschreibt Analysen, die jeweils zwischen zwei Editieroperationen durchgeführt werden. Diese werden zur Prüfung, aber auch zur automatischen Änderung visueller Programme benutzt. In diesem Abschnitt wird unter anderem auch gezeigt, daß wichtige Teile der Spezifikation der Analyse sich auch unverändert für eine textuelle Notation eines visuellen Programms durchführen lassen. Der zweite Abschnitt ist der Spezifikation von Übersetzungsaufgaben gewidmet.

4.6.1. Konsistenzprüfungen

Abschnitt 4.5.3 hat am Beispiel der If-To-While-Transformation gezeigt, daß Editieroperationen auch durch Prozeduren realisiert werden können, die ein Sprachentwerfer selbst schreibt oder aus einer Bibliothek wiederverwendet. Mit dieser Technik lassen sich sehr weitgehend alle Arten von lokalen Modifikationen durchführen.

Diese Methode hat jedoch auch ihre Grenzen, z.B. wenn sich eine Editieroperation potentiell auf viele andere, im Strukturbaum weit entfernte Kontexte auswirkt. Diese Situation tritt zum Beispiel auf, wenn eine Sprache Definiert-Benutzt-Beziehungen enthält und bestimmte Konsistenzbedingungen zwischen Definition und Anwendung durch den Strukturediteor geprüft oder automatisch erfüllt werden sollen. Um die Konsistenz nach Änderung einer Definition durch entsprechende Änderungen an allen Anwendungsstellen aufrechtzuerhalten, müßten Baumberechnungen im größeren Maßstab formuliert werden – eine Aufgabe, die durch Einsatz eines aus Spezifikationen hergestellten Attributauswerters einfacher gelöst werden kann.

Konsistenzprüfungen sind Analyseaufgaben, für die Attributauswerter eingesetzt werden können. Sollen Prüfungen nach jeder Editieroperation automatisch stattfinden, so können die Berechnungen durch einen speziellen Attributauswerter, dem *Attributauswerter für Konsistenzprüfungen* durchgeführt werden. Diese Berechnungen werden dann jeweils vor der Erzeugung der graphischen Darstellung der Fenster ausgeführt. Sie können außer für Konsistenzprüfungen noch benutzt werden, um Informationen zwischen den Teilen eines visuellen Programms auszutauschen, die in verschiedenen Fenstern dargestellt sind. Auf diesen Aspekt ist bereits in Abschnitt 4.3.2 eingegangen worden. Schließlich können mit dem Attributauswerter für Konsistenzprüfungen nach einer Programmänderung auch weitere Änderungen durchgeführt werden, so daß Konsistenzfehler auch automatisch korrigiert werden können.

Im folgenden zeige ich zwei Anwendungsbeispiele dieses Attributauswerters.

Im ersten Beispiel werden die Verbindungen eines Petri-Netzes geprüft. Dies ist ein einführendes Beispiel das zeigt, wie einfach die Spezifikation von Konsistenzprüfungen ist. Es zeigt insbesondere auch, daß sich die Spezifikationen in der attributierten Grammatik nur geringfügig von entsprechenden Spezifikationen für textuelle Sprachen unterscheiden.

Im zweiten Beispiel wird die Konsistenz zwischen den Tabellendefinitionen und den Anwendungen von Tabellen in QBE aufrechterhalten. Das führt hier zu einem effizient benutzbaren Strukturediteur für QBE. Das Beispiel ist von allgemeinerem Interesse, weil ähnliche Aufgaben auch in vielen anderen visuellen Sprachen vorkommen.

4.6.1.1. Prüfung der Verbindungen bei Petri-Netzen

Die Verbindungen bei den Petri-Netzen dürfen immer nur Stellen und Transitionen miteinander verbinden, nicht Stellen mit Stellen oder Transitionen mit Transitionen. VLEli unterstützt zwei Ansätze, um diese Konsistenzbedingung aufrechtzuerhalten. Zum einen können illegale Verbindungen bereits durch die Spezifikation der Editieroperationen verhindert werden[7], zum anderen können Fehlermeldungen für sie erzeugt werden.

Beide Varianten dieser Aufgabe erfordern zur Lösung strukturell ähnliche Berechnungen: Zunächst muß im Kontext der Stellen und der Transitionen die Art des Programmobjekts in der Definitionstabelle gesetzt werden. Im Kontext der Verbindung kann diese Information gelesen werden. Daraus ist ersichtlich, welche Sprachkonstrukte verbunden werden und es kann ggf. eine Fehlermeldung erzeugt werden. Die Reihenfolge der Berechnungen wird dabei über zusätzliche Attributabhängigkeiten ausgedrückt.

Bemerkenswert ist dabei, daß diese Analyseaufgabe für eine textuelle Notation der Petri-Netze zu einer sehr ähnlichen Spezifikation führt. Abbildung 4.22 zeigt die Spezifikation dieser Analyseaufgabe für beide Sprachvarianten und kennzeichnet Gemeinsamkeiten und Unterschiede. Der gesamte Ablauf der Analyse ist bei beiden Varianten identisch – das ist das Ergebnis des Entwurfs der Programmrepräsentation. Unterschiede ergeben sich lediglich bei der Berechnung der Einträge in die Definitionstabelle und bei der Ausgabe der Fehler.

Um die Einträge in die Definitionstabelle zu ermitteln, müssen bei der textuellen Variante zunächst die Beziehungen zwischen den Sprachkonstrukten durch eine Namensanalyse ermittelt werden. Bei der visuellen Variante ist dies nicht erforderlich, da die Beziehungen nicht durch Bezeichner, sondern direkt durch Einträge in die Definitionstabelle repräsentiert werden. Bei den Fehlermeldungen werden in beiden Sprachvarianten unterschiedliche Informationen benötigt. Im

[7] Siehe dazu Abschnitt 4.5.2.1

TextAnaPrep.lido[1]:

```
/* Perform Name Analysis */

CLASS SYMBOL IdentOcc: Sym: int;
CLASS SYMBOL IdentOcc COMPUTE
   SYNT.Sym = TERM;
END;

SYMBOL DefId INHERITS IdentOcc, IdDefScope, Unique COMPUTE
   IF (NOT(THIS.Unique),
      message(ERROR, "Multiple Definition", 0, COORDREF));
END;

SYMBOL UseId INHERITS IdentOcc, IdUseEnv, ChkIdUse
END;

/* Copy Definition Table Keys */

SYMBOL State, Transition: Key: DefTableKey;
SYMBOL Connection: StartKey, EndKey: DefTableKey;

SYMBOL State COMPUTE
   SYNT.Key = CONSTITUENT DefId.Key;
END;

SYMBOL Transition COMPUTE
   SYNT.Key = CONSTITUENT DefId.Key;
END;

RULE: Connection ::= 'CONNECTION' UseId '->' UseId ';'
COMPUTE
   Connection.StartKey = UseId[1].Key;
   Connection.EndKey = UseId[2].Key;
END;

/* Produce Error Message */

SYMBOL Connection COMPUTE
   IF (THIS.illegalConnection,
      message(ERROR, "IllegalConnection", 0, COORDREF));
END;
```

(a) Unterschiedliche Spezifikation textuelle...

VisAnaPrep.check.lido[1]:

```
/* Extract Keys from persistent Attributes */

SYMBOL State, Transition: Key: DefTableKey;
SYMBOL Connection: StartKey, EndKey: DefTableKey;

SYMBOL State COMPUTE
   SYNT.Key = KEYOF(THIS.persKey);
END;

SYMBOL Transition COMPUTE
   SYNT.Key = KEYOF(THIS.persKey);
END;

SYMBOL Connection COMPUTE
   SYNT.StartKey = KEYOF(THIS.persFrom);
   SYNT.EndKey = KEYOF(THIS.persTo);
END;

/* Produce Error Message */

SYMBOL Connection COMPUTE
   IF (THIS.illegalConnection,
      message(ERROR, "IllegalConnection", 0, _currn, "petri"));
END;
```

(b) ... und visuelle Variante

IllegalConnections.head[1]:

```
#define UNSET 0
#define STATE 1
#define TRANSITION 2
```

IllegalConnections.pdl[2]:

```
Kind: int;
```

IllegalConnections.check.lido[3]:

```
/* Set Kind Property */

SYMBOL State COMPUTE
   SYNT.KindOk = ResetKind(THIS.Key, STATE);
END;

SYMBOL Transition COMPUTE
   SYNT.KindOk = ResetKind(THIS.Key, TRANSITION);
END;

/* Establish Precondition: All Kind Properties are set */

SYMBOL PetriNet COMPUTE
   SYNT.KindOk = CONSTITUENTS (Transition.KindOk, State.KindOk);
END;

/* Extract Kind Property and check for error */

SYMBOL Connection: fromKind, toKind: int;
SYMBOL Connection: illegalConnection: int;

SYMBOL Connection COMPUTE
   SYNT.fromKind = GetKind(THIS.StartKey, UNSET)
      <- INCLUDING PetriNet.KindOk;
   SYNT.toKind = GetKind(THIS.EndKey, UNSET)
      <- INCLUDING PetriNet.KindOk;

   SYNT.illegalConnection =
      AND(EQ(THIS.fromKind, THIS.toKind), NE(THIS.fromKind, UNSET));
END;
```

(c) Gemeinsame Analyseteile

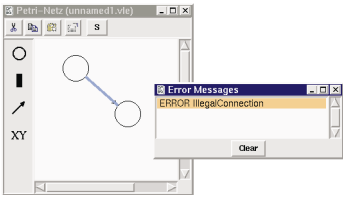
Input.petri[1]:

```
PETRINET
  STATE A;
  STATE B;

  CONNECTION A -> B;
END
```

Output[2]:

```
"Input.petri", line 5:3 ERROR: IllegalConnection
```

(d) Erzeugte Meldung: textuelle...



(e) ... und visuelle Variante

Abbildung 4.22.: Gemeinsamkeiten und Unterschiede der Spezifikation einfacher Analyseaufgaben für die visuelle und textuelle Notation der Petri-Netze

textuellen Fall sind dies die Zeile und die Spalte in der analysierten Textdatei. Diese werden zusammen mit der Fehlermeldung ausgegeben.

Im Strukturediteur wird für die Erzeugung einer Fehlermeldung der Name einer Sicht und ein Zeiger auf das fehlerhafte Konstrukt im Strukturbaum benötigt. Damit kann die generierte Entwicklungsumgebung ein Fenster mit Fehlermeldungen erzeugen, siehe Abbildung 4.22(e). Wählt der Benutzer eine Fehlermeldung an, so wird ein Fenster mit dem fehlerhaften Konstrukt geöffnet oder in den Vordergrund gebracht und darin das fehlerhafte Konstrukt markiert. Das ist möglich, da anhand des Kontextes im Strukturbaum der zugehörige Teil der Darstellung ermittelt werden kann.

4.6.1.2. Konsistenz zwischen Tabellendefinition und Abfragen bei QBE

Datenbankdefinitionssprachen bestehen oft aus Konstrukten für die Definition von Datenbanktabellen und Konstrukten für die Spezifikation von Sichten. Sichten sind in diesem Zusammenhang Datenbankabfragen, auf die ein Anwendungsprogramm wie auf reguläre Datenbanktabellen zugreifen kann. Dadurch wird Datenunabhängigkeit erreicht, weil Anwendungsprogramme nicht zwischen Tabellen und Abfragen zu unterscheiden brauchen.

Abbildung 4.23 zeigt eine Datenbankdefinition in einer zu diesem Zweck erfundenen Datenbankdefinitionssprache. Das Bild zeigt im obersten Fenster zwei Datenbanktabellen, Adressen und Bestellungen. Diese werden durch Tabellen definiert, deren Spalten die Felder der Datenbank und deren Einträge die Typen der Felder sind. Als Konstrukt für die Definition von Sichten wurde eine Abfragesprache realisiert, die auf der Sprache *Query by Example* (QBE) beruht.

Die Sprache QBE [Zloof 1975] ist eine Datenbankabfragesprache, bei der der Benutzer zur Definition einer Abfrage eine Tabelle ausfüllt, deren Struktur mit der Datenbanktabelle übereinstimmt, die der Abfrage zugrundeliegt. Um eine bestimmte Abfrage zu formulieren, schreibt er Beispieldaten in die Tabelle. Das Datenbanksystem antwortet darauf mit allen Einträgen der Datenbank, die auf die angegebenen Beispieldaten passen.

In Abbildung 4.23 sind zusätzlich zu den Datenbanktabellen Adressen und Bestellungen noch einige Sichten definiert. Die Sicht BestellAdressen spezifiziert ein inneres Kreuzprodukt (*inner join*) dieser Tabellen über das Feld AdrId (siehe mittleres Fenster). Darauf basiert eine weitere Sicht, die die Bestellungen der Kunden mit dem Namen Müller enthält (unterstes Fenster).

Das für diesen Abschnitt wesentliche Merkmal dieser visuellen Datenbankdefinitionssprache sind die Konsistenzbedingungen zwischen den Einträgen einer Datenbanktabelle oder einer Sicht und ihren Anwendungen in weiteren Sichten. Der Benutzer einer solchen Sprache könnte etwa an der Definition der Tabelle Adressen etwas ändern, z.B. das Feld PLZ entfernen. Durch eine solche Änderung würde die Abfrage BestellAdressen mit der Tabellendefinition inkonsistent: alle

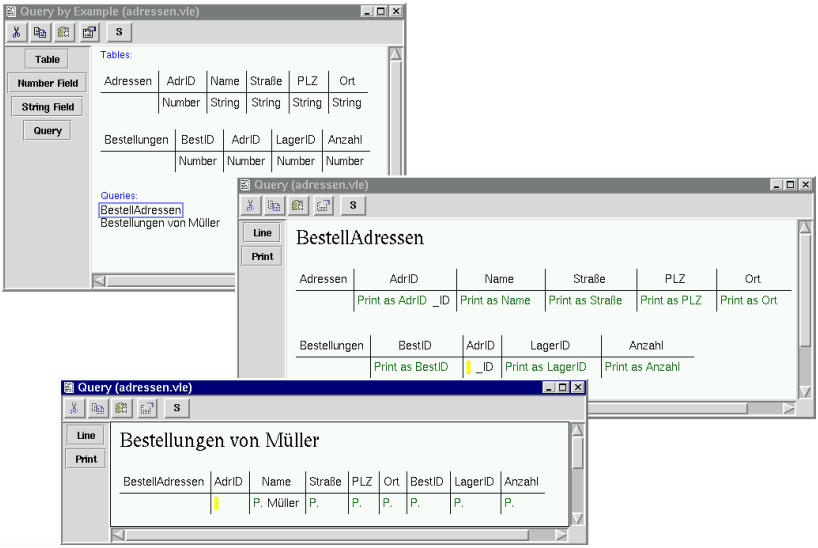


Abbildung 4.23.: Datenbanktabellen und ihre Anwendung in QBE-Abfragen

Daten, die das Feld PLZ betreffen, müßten ebenfalls gelöscht werden. Das führt zu weiteren Änderungen in der Abfrage Bestellungen Müller. In der Beispielimplementierung dieser Sprache werden solcherlei konsistenzerhaltende Änderungen automatisch durchgeführt. So lassen sich Datenbankänderungen relativ leicht realisieren. Die Benutzung des Systems wird einfacher, die Viskosität [Green und Petre 1996] nimmt ab.

Diese Funktionalität wurde ebenfalls mit Hilfe des Attributauswerters für Konsistenzprüfungen realisiert. Hierzu geht man in zwei Schritten vor: Zunächst muß die Inkonsistenz zwischen Definitions- und Anwendungsstelle erkannt werden. Anstatt einer Fehlermeldung muß jedoch im zweiten Schritt eine Änderung der abstrakten Struktur durchgeführt werden.

Die Erkennung der Inkonsistenz zwischen einer visuellen Definition und Anwendung, ist demselben Problem für textuelle Sprachen ähnlich. Anstelle der Tabellendefinition und der Felder stelle man sich beispielsweise eine Prozedur und ihre formalen Parameter vor. Die Rolle einer Abfragetabelle in einer QBE-Abfrage entspricht dann einem Prozeduraufruf, bei dem die aktuellen Parameter an die formalen nicht durch die Position in der Parameterliste, sondern über die Namen der formalen Parameter gebunden werden. Abbildung 4.24 skizziert den Strukturbaum und die den Symbolen zugeordneten Einträge in die Definitionstabelle für

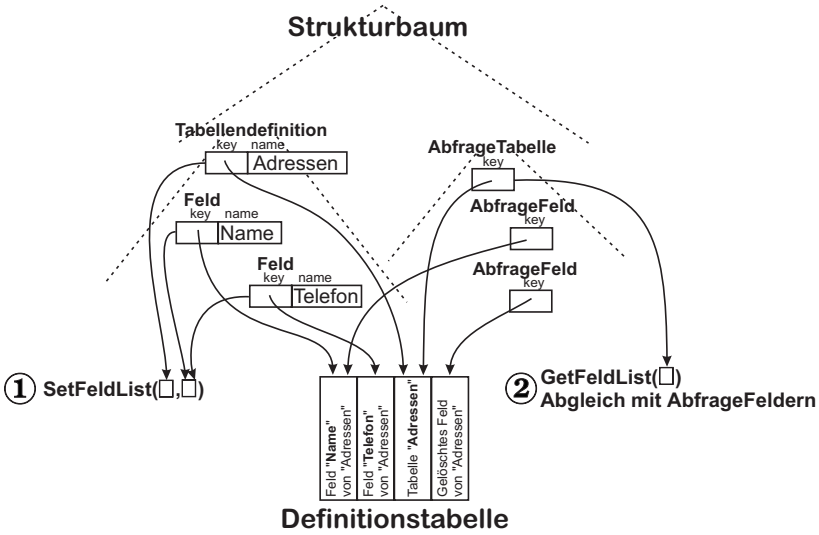


Abbildung 4.24.: Programmrepräsentation für das Konsistenzproblem

das Analyseproblem. Die gezeigte Struktur ergibt sich in ähnlicher Form auch bei der Analyse des textuellen Prozeduraufrufs, nachdem die Namensanalyse durchgeführt ist.

Im Falle der Konsistenz sind die AbfrageFelder in einer Abfrage mit den Feldern der zugehörigen Tabellendefinition über einen Eintrag in die Definitionstabelle verbunden, wie dies in Abbildung 4.24 für das Feld “Name” der Fall ist. Um die Konsistenz zu prüfen, kann man eine Liste der Definitionstabellen-Einträge der Felder für eine Tabelle erzeugen und diese in die Kontexte der AbfrageTabellen transportieren. Über einen Vergleich der existierenden AbfrageFelder mit den Einträgen in der Liste kann man ermitteln, welche Felder in der Tabelle gelöscht wurden und welche neu hinzugekommen sind. Im Bild tritt das unterste AbfrageFeld nicht mehr auf und ist in der Tabellendefinition entfernt worden. Weiterhin enthält die Liste den Definitionstabellen-Eintrag des Feldes “Telefon”, das nicht bei den AbfrageFeldern vorkommt und deshalb neu erzeugt worden sein muß.

Um die Konsistenz wieder herzustellen, müßte im Strukturbaum aus Abbildung 4.24 also ein AbfrageFeld gelöscht, ein neues erzeugt und diesem der Schlüssel des Feldes “Telefon” zugeordnet werden. Hierfür sind strukturelle Änderungen in der AbfrageTabelle erforderlich, die anhand der abstrakten Struktur der AbfrageTabelle formuliert werden können. Um diese Änderungen zu realisieren,

wird Code in einer Scriptsprache erzeugt und nach Abarbeitung des Attributauswerters ausgeführt, siehe dazu Abschnitt 4.3.4.

Wie bereits der Vergleich mit Prozeduraufrufen für textuelle Sprachen nahelegt, sind die automatisch durchgeführten strukturellen Änderungen, die die Konsistenz zwischen einer Definition und ihren Anwendungen aufrechterhalten, auch für andere Sprachen, beispielsweise Streets und Prograph anwendbar. In Streets enthält die Schnittstelle einer Task bzw. einer Prozedur einige, als Dreiecke dargestellte Ports. Diese Ports können bei der Anwendung mit anderen Ports verbunden werden. Wird in einer Task ein Port aus der Schnittstelle gelöscht, so kann dieser an den Anwendungsstellen automatisch entfernt werden. In Prograph enthalten Datenflußgraphen Verbindungspunkte, die bei der Anwendung des Graphen mit anderen Knoten verbunden werden. Auch hier werden bei Änderungen an der Schnittstelle eines Datenflußgraphen die Anwendungsstellen automatisch konsistent gehalten.

Darüberhinaus ist die Technik, automatisch Strukturänderungen anhand von Analyseergebnissen durchzuführen, auch für andere Aufgaben anwendbar. Ein einfaches Beispiel ist die automatische Vergabe eindeutiger Namen für neue Definitionen. Ein anderes Beispiel sind Strukturänderungen, die anhand der Ergebnisse der Typanalyse durchgeführt werden. Reps und Teitelbaum [1989b] wenden diese Technik beispielsweise für die Implementierung von Strukturediteren für textuelle Sprachen an.

Die Durchführung von Analysen stellt nach der Erzeugung der graphischen Darstellung einen zweiten Anwendungsbereich für Attributauswerter in dieser Arbeit dar. Die automatische Durchführung struktureller Änderungen im Rahmen der Analyse ist dabei von besonderer Bedeutung, da dadurch die Benutzung einer visuellen Sprache wesentlich vereinfacht wird. Auch die in diesen Bereich fallenden Aufgaben lassen sich durch Symbolattributierung und Vererbung so umsetzen, daß sie durch Erben von Symbolrollen und Überschreiben von Attributberechnungen angewendet werden können. Um dies zu zeigen, wurde die Aufgabe der Konsistenzerhaltung zwischen Definition und Anwendung durch ein Spezifikationsmodul abstrahiert und dieses zur Implementierung der Sprachen Streets und QBE angewendet [Schmidt und Schindler 2000].

4.6.2. Übersetzung visueller Programme

Die Übersetzung ist das Haupt-Anwendungsgebiet attributierter Grammatiken. Die Übersetzung visueller Programme unterscheidet sich dabei nicht wesentlich von der textueller Programme. Für die Übersetzungsaufgabe lassen sich hier attributierte Grammatiken und die anderen Werkzeuge und Bibliotheken des Eli-Systems benutzen. Die Übersetzung bietet, verglichen mit der Übersetzung textueller Programme nur wenig neues. Anzumerken ist hier lediglich, daß eine we-

sentliche Aufgabe wegfällt – das ist die Namensanalyse. Darauf ist aber der vorige Abschnitt bereits eingegangen.

Abbildung 4.25 zeigt ein Nassi-Shneiderman-Programm und seine Übersetzung in ein C++-Programm. Um diese Transformationsaufgabe zu lösen, wurde außer attributierten Grammatiken das Werkzeug PTG verwendet, um den Zieltext zu beschreiben. Weiterhin wurde das ELI-Modul "Indent" benutzt, um den generierten Quelltext einzurücken.

Außer der Spezifikation des Übersetzers gibt es einige weitere Themenbereiche, die die Übersetzung visueller Programme betreffen. Der erste Punkt betrifft die Erzeugung von Source-To-Source-Übersetzern. Implementiert man wie hier eine Entwicklungsumgebung, so besteht oft der Wunsch, nach der Übersetzung ein weiteres, oft externes Übersetzungswerkzeug aufzurufen. Dies erfolgt idealerweise transparent für den Benutzer. Dabei eventuell auftretende Fehlermeldungen müssen zu den Kontexten des visuellen Programms zurückverfolgt werden.

Dieser Aspekt wurde im Zusammenhang mit der Erstellung des Streets-Übersetzers [Kastens und Jung 1998] untersucht. Man kann die für die Abbildung der Fehler benötigten Informationen mit Hilfe der Ausgabeverarbeitung des Werkzeugs PTG erhalten, wenn man bei den Ausgaben die Zeile und Spalte mitführt und in den PTG-Mustern zusätzlich die Kontexte des Strukturbaums übergibt.

Ein anderer Punkt betrifft Strukturediteren. Ihre Benutzung ist manchmal unbequem, z.B. wenn man Ausdrücke mit ihnen bearbeitet. Dabei müssten ihre zahlreichen Bestandteile wie verschiedene Operatoren und Literale mit der Maus auf

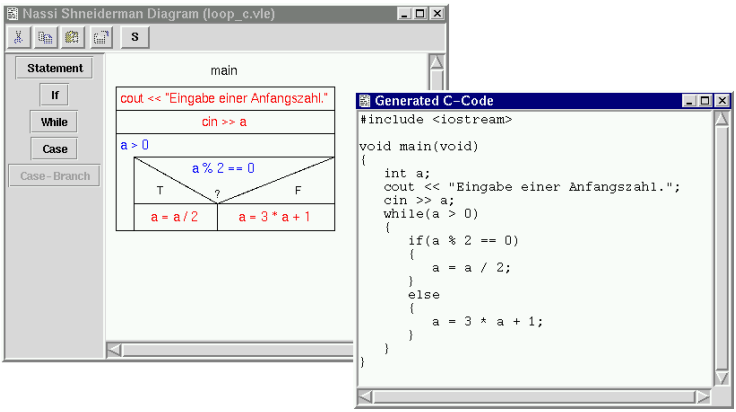


Abbildung 4.25.: Übersetzung eines Nassi-Shneiderman-Programms in ein C++-Programm

die Arbeitsfläche gezogen werden. Dazwischen müßten die Werte der Literale mit der Tastatur spezifiziert werden. Das kann man vermeiden, wenn man Ausdrücke als Texte in (persistenten) Attributen speichert. Da dabei kein Strukturbaum für den Ausdruck für Analysen und Transformationen zugänglich ist, müssen diese Texte passend für die Zielsprache vorliegen. Um dieses Problem zu lösen, kann man die eingegebenen Texte mit Hilfe eines Zerteilers (engl. *parser*) in Strukturbäume transformieren, die als berechnete Teilbäume in die attributierte Grammatik eingebunden und weiterverarbeitet werden können. Dazu muß man einen Zerteiler generieren, der alternativ verschiedene Teilgrammatiken bearbeiten kann. Dieses Problem hat u.a. Geisser [1998] im Rahmen seiner Diplomarbeit gelöst.

4.7. Literaturbezug

In diesem Kapitel habe ich eine Methode konzipiert, mit der Entwicklungsumgebungen für visuelle Sprachen generiert werden können. Die Methode wurde durch das VLEli-System praktisch umgesetzt. In diesem Abschnitt beschreibe ich wichtige Unterschiede zwischen dem VLEli-System und anderen Generatoren für visuelle Sprachen. Einen Überblick über die Ansätze habe ich in Abschnitt 2.3 gegeben.

Eine wichtige Grundlage für die Sprachimplementierung im VLEli-System ist die Repräsentation der Programme visueller Sprachen. Dazu werden Strukturbäume verwendet, die um zusätzliche Datenstrukturen für die Repräsentation von Beziehungen ergänzt werden. Die in dieser Arbeit verwendete Art der Ergänzung ermöglicht es, Werkzeuge und Methoden der Implementierung textueller Sprachen auch für visuelle einzusetzen. Dies sind etwa Generatoren für attributierte Grammatiken, mit denen vom VLEli-System Werkzeuge zur Analyse und Weiterverarbeitung der visuellen Programme erzeugt werden können.

Dies ist ein wesentlicher Unterschied zu anderen Ansätzen, die den Versuch der Wiederverwendung von Werkzeugen nicht unternehmen. In anderen Ansätzen wird so meist keine Werkzeugunterstützung für die Analyse und Weiterverarbeitung visueller Programme geboten. In den Ansätzen von Zhang und Zhang [1998], Golin und Magliery [1993] und Chok und Marriott [1998] können etwa lediglich Anknüpfungen spezifiziert werden, die bei der Erkennung von Sprachkonstrukten durch den visuellen Parser angestoßen werden. Lediglich beim Ansatz von Crimi u.a. [1990] können zur Weiterverarbeitung einfache Attributberechnungen spezifiziert werden, die aber nicht ausreichen, um komplexere Analyseaufgaben zu formulieren.

Im VLEli-System wird ein Großteil der sprachabhängigen Teile der erzeugten Entwicklungsumgebungen durch Attributauswerter implementiert. Das betrifft nicht nur die Layoutberechnungen, die Erzeugung der graphischen Darstellung

und die Editieroperationen, sondern darüberhinaus auch die Konsistenthaltung unterschiedlicher Sichten auf das visuelle Programm und die Durchführung von Strukturmodifikationen. Einerseits lassen sich so für all diese Aufgaben auch Informationen über das visuelle Programm benutzen, die ebenfalls mit Attributauswertern extrahiert werden können. Dadurch eignet sich VLEli für eine große Menge visueller Sprachen. Andererseits sind mit den Generatoren für attributierte Grammatiken auch effektive Techniken zur Modularisierung der Spezifikationen [Kastens und Waite 1994] für visuelle Sprachen einsetzbar. Im folgenden Kapitel zeige ich, daß damit flexibel kombinierbare Module bereitgestellt werden können, mit denen visuelle Sprachen auf hoher Ebene und in einfacher Weise implementiert werden können.

Attributierte Grammatiken werden auch in einigen anderen Ansätzen für die Implementierung visueller Sprachen benutzt. Der Ansatz, Teile von Spezifikationen zu modularisieren, ist jedoch nicht weit verbreitet. Von den mir bekannten Generatoren unterstützt lediglich das LOGGIE-System [Backlund u.a. 1990] ein entfernt ähnliches Konzept. Hierbei werden die Berechnungen einzelner Attribute einer attributierten Grammatik in einer Bibliothek abgelegt. Diese können so für die Implementierung anderer visueller Sprachen wiederverwendet werden. Da diese Methode aber lediglich einzelne Aspekte einer Spezifikation kapseln kann, kann im LOGGIE-System keine hohe Abstraktionsebene erreicht werden.

Um benutzerfreundliche Editoren für visuelle Sprachen zu generieren, werden im VLEli-System unter anderem zwei verschiedene Techniken der Layouterzeugung umgesetzt: Attributberechnungen eignen sich gut, um etwa platzoptimierte graphische Darstellungen zu erzeugen während Constraint-Solver bei Darstellungen mit Layoutfreiheiten zu besseren Ergebnissen führen. Ein wichtiger Aspekt dieser Arbeit ist, daß beide Verfahren alternativ benutzt werden können und sich sogar für die Implementierung visueller Sprachen kombinieren lassen. Dies ist bei anderen Ansätzen nicht der Fall: hier wird jeweils lediglich eine Methode der Layouterzeugung unterstützt.

Auch die Integration von Constraint-Solver und Attributauswerter ist als im Bereich der Implementierung visueller Sprachen neuartig anzusehen. Lediglich Barford und Vander Zanden [1989] beschreiben die Einbindung von Constraint-Solvern in Attributauswerter, implementieren damit aber Visualisierungssysteme. Im LOGGIE-System [Backlund u.a. 1990] werden zwar auch Constraint-Solver und Attributauswerter benutzt, der Constraint-Solver ist hier aber in eine über dem Attributauswerter liegenden Darstellungsebene gekapselt und allein für die Layoutberechnung verantwortlich.

5. Entwicklung und Einsatz von Spezifikationsmodulen

Ein Spezifikationsmodul ist ein Bibliotheksmodul, das Spezifikationen für verschiedene Werkzeuge, insbesondere auch Teile attributierter Grammatiken kapseln kann. Die Modularisierung der Aufgaben, die durch attributierte Grammatiken gelöst werden, wird dabei in hohem Maße durch die Abstraktionstechniken unterstützt, die das Liga-System bereitstellt. Im Bereich der Implementierung textueller Sprachen können so wichtige Aufgaben, etwa die Namens- und die Typanalyse in einfacher Weise gelöst werden [Kastens und Waite 1994]. Durch die Abstraktionstechniken des Liga-Systems sind die Module flexibel einsetzbar, weil sie unabhängig von den Grammatikproduktionen der zu implementierenden Sprache sind. Sie lassen sich gut miteinander kombinieren, weil ein Sprachkonstrukt Attributberechnungen mehrerer Module gleichzeitig erben kann. Weiterhin lassen sich einzelne Berechnungen eines Moduls überschreiben, so daß die Spezifikationsmodule einen weiten Anwendungsbereich haben, gleichzeitig aber einfach anzuwenden sind.

Im weiteren werden Qualitätsmerkmale der Spezifikationsmodule in hohem Maße bei ihrem Entwurf festgelegt. Die Spezifikationsmodule, die im Eli-System die Implementierung textueller Sprachen vereinfachen, bauen auf einer allgemeinen Strukturierung der Aufgabe der Sprachimplementierung auf [Kastens 1990, 1991a; Waite und Goos 1984]. Eine solche Strukturierung leitet einen Sprachentwerfer bei der Sprachimplementierung und hilft ihm, Teilaufgaben zu identifizieren. Für diese Teilaufgaben findet er in der Spezifikationsmodul-Bibliothek Lösungsalternativen, die Standardlösungen der Sprachimplementierung kapseln. Diese Lösungen kann er, weitgehend ohne Spezifikationen einer niedrigeren Ebene zu benutzen, zu vollständigen Sprachimplementierungen integrieren.

Die Spezifikationsmodule werden so zu Sprachkonstrukten einer problemorientierten Spezifikationssprache, deren Konzepte unmittelbar auf das Denkmodell des Sprachentwerfers passen. Das macht sie auch für die Anwendung durch Nicht-Spezialisten geeignet. Da die Spezifikationsmodule Lösungen auf einer sehr hohen Ebene anbieten, sind die resultierenden Spezifikationen kurz und leicht verständlich. Weiterhin werden für eine Aufgabe meist mehrere Lösungswege mit

ähnlichen Schnittstellen angeboten. So lassen sich Sprachänderungen und Designexperimente leicht durchführen.

In dieser Arbeit wurden Spezifikationsmodule zur Implementierung visueller Sprachen erarbeitet und damit die Abstraktionstechnik der Spezifikationsmodule in einem neuen Bereich angewendet. Dieses Kapitel diskutiert interessante Aspekte ihrer Entwicklung und Anwendung. Dabei stellt sich heraus, daß sich wichtige Eigenschaften auch für den visuellen Anwendungsbereich einstellen. Indem wie bei textuellen Sprachen die Abstraktionstechniken des Liga-Systems mit einem Denkmodell kombiniert werden, das einem Sprachentwickler als Leitfaden dient, werden Einfachheit, Flexibilität und eine hohe Spezifikationsebene erreicht und gleichzeitig ein breites Spektrum von Sprachstilen unterstützt.

Wichtige Vorarbeiten dazu sind in den vorangehenden Kapiteln diskutiert worden: In Kapitel 3 wurde die Spezifikationsmodulbibliothek entworfen. Hierzu wurden visuelle Muster eingeführt, die einen Sprachentwickler beim Entwurf einer visuellen Sprache leiten und die ihm helfen, Spezifikationsmodule für ihre Implementierung auszuwählen. Kapitel 4 hat das Werkzeugsystem VLEli vorgestellt, das u.a. mit dem Ziel entwickelt wurde, die Abstraktionstechniken des Liga-Systems für die Implementierung visueller Sprachen anwendbar zu machen. Dadurch sind auch die technischen Voraussetzungen gegeben, um Spezifikationsmodule für die Implementierung visueller Sprachen zu entwickeln.

Dieses Kapitel ist in zwei Teile gegliedert. Der erste Teil diskutiert die den Spezifikationsmodulen zugrundeliegenden Konzepte. Neben den Effekten, die die Abstraktionstechniken des Liga-Systems auf die Anwendung der Spezifikationsmodule haben, sind dies unter anderem die Überlegungen, die die Kombinierbarkeit der Spezifikationsmodule sichern. Der zweite Teil evaluiert den Ansatz der Spezifikation visueller Sprachen durch Module. Dazu gebe ich einen Überblick über die Modulbibliothek und diskutiere Eigenschaften ihrer Anwendung anhand einiger Beispielsprachen.

5.1. Konzepte

Im Bereich visueller Sprachen haben die Spezifikationsmodule u.a. die Aufgabe, die graphische Darstellung für bestimmte strukturelle Abstraktionen zu erzeugen und die Editieroperationen zu realisieren. Die Spezifikationsmodule unterscheiden sich dabei in der zugrundeliegenden strukturellen Abstraktion – z.B. Folgen von Anweisungen oder Mengen von Zuständen – und im Aufbau der graphischen Darstellung. Diesem Aspekt wurde bei ihrem Entwurf Rechnung getragen; er hat in Kapitel 3 zum Begriff der visuellen Muster geführt.

Ein visuelles Muster legt gewisse Eigenschaften eines Sprachkonstrukts fest, läßt aber viele Freiheiten. Diese Freiheiten werden z.T. durch Parameter der Spezifikationsmodule umgesetzt, manche müssen aber für die Implementierung auch

festgelegt werden. In einem Spezifikationsmodul werden so gewisse Entwurfsentscheidungen getroffen, die die unterstützten graphischen Darstellungen verschiedentlich einschränken und deren Verhalten bei Benutzerinteraktion beeinflussen. Dadurch wird letztendlich auch festgelegt, wie einfach die Anwendung eines Spezifikationsmoduls ist.

Eine wichtige Eigenschaft der Spezifikationsmodule ist es deshalb, daß mehrere Module für verschiedene Ausprägungen von Entwurfsentscheidungen zur Verfügung gestellt werden können. So kann für einfache Anwendungsfälle eine einfache Spezifikation erreicht werden, ohne die Ausdruckskraft für den komplexen Fall einzuschränken.

Um effizient verschiedene Spezifikationsmodule für eine breite Palette visueller Muster bereitzustellen, wird eine mehrstufige Spezifikationstechnik eingesetzt. Dies hat gegenüber einer einstufigen Spezifikation den Vorteil, daß grundlegende Methoden und Werkzeuge wie der Aufruf des Constraint-Solvers nicht wieder und wieder für verschiedene Spezifikationsmodule implementiert werden müssen. Weiterhin ist der Implementierungsaufwand für ein Spezifikationsmodul vergleichweise kleiner, weil für seine Umsetzung auf eine allgemeinere Spezifikationstechnik aufgesetzt werden kann. Ein sich dadurch ergebender Vorteil ist, daß ein umfangreicher Satz von Ausdrucksmitteln auf einer hohen Ebene eingesetzt werden kann, um eine visuelle Sprache zu implementieren.

Dieser Abschnitt diskutiert die Konzepte und Methoden, die den Spezifikationsmodulen zugrundeliegen. Im ersten Unterabschnitt wird dazu die Anwendung der Spezifikationsmodule an einem kleinen Beispiel vorgeführt und gezeigt, wie leicht sich Sprachänderungen und Modifikationen der graphischen Darstellung realisieren lassen.

Abschnitt 5.1.2 diskutiert Konzepte, die sicherstellen, daß sich die Spezifikationsmodule miteinander kombinieren lassen. In diesem Zusammenhang werden die Layoutberechnungen geplant und der Nutzen der Mehrfachvererbung demonstriert. Danach gehe ich auf die visuelle Notation der Symbolrollendiagramme ein. Symbolrollendiagramme charakterisieren Anwendungssituationen eines Spezifikationsmoduls. Anhand dieser Diagramme können Spezifikationsmodule korrekt angewendet werden, ohne komplexere Zusammenhänge aus der Implementierung der Module zu kennen.

Abschnitt 5.1.4 geht schließlich auf die dynamischen Zeichnungen ein, die Teile der graphischen Darstellung einer visuellen Sprache beschreiben. Dynamische Zeichnungen können vergrößert werden, um andere graphische Darstellungen in ihnen einzufügen. Vergleichbar mit entsprechenden Werkzeugen zur Erzeugung von Zieltexten bei der Übersetzung textueller Sprachen sind sie ein wichtiges Beschreibungsmittel für visuelle Sprachen in dieser Arbeit.

5.1.1. Anwendung der Spezifikationsmodule

In diesem Abschnitt wird eine einfache, aber vollständige Anwendung der Spezifikationsmodule diskutiert. Im ersten Teil zeige ich das hohe Niveau, das durch Spezifikationsmodule bei der Sprachimplementierung erreicht wird. Die einfache visuelle Sprache wird durch lediglich 20 Zeilen Spezifikation implementiert, von denen nur 8 für die Anwendung der Spezifikationsmodule benötigt werden. Aus diesen Spezifikationen wird eine Entwicklungsumgebung für die Beispielsprache implementiert, die aus ca. 25 000 Zeilen C-Code und ca. 8000 Zeilen Tcl-Code besteht.

Der zweite Teil zeigt, wie einfach sich Änderungen der graphischen Darstellung und Spracherweiterungen realisieren lassen.

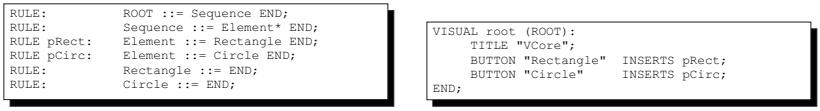
5.1.1.1. Die Sprache VCore

Visuelle Programme der Beispielsprache VCore bestehen (zunächst) lediglich aus einer einfachen Folge zweier Konstrukte. Diese werden der Einfachheit halber durch senkrecht untereinanderstehende geometrische Figuren, durch einen Kreis bzw. ein Quadrat visualisiert. Der Leser mag sich hier Folgen in anderen Sprachen vorstellen, z.B. Folgen von Ports in Streets oder Folgen von Anweisungen in Nassi-Shneiderman-Diagrammen.

Zum Entwurf der Sprache kann man Bezug auf das Listen-Muster nehmen. Außer der Anordnung der Elemente der Folge werden dadurch auch wichtige Editieroperationen spezifiziert. Weiterhin lassen sich dadurch Qualitätsaussagen zur Sprache VCore machen: Beim Listen-Muster sind alle Listenelemente gleichzeitig sichtbar. Dadurch ist seine graphische Darstellung übersichtlich, solange der Platzverbrauch für die einzelnen Elemente nicht zu groß wird. Ein alternatives Muster zur Beschreibung einer Folge wäre das Stapel-Muster, siehe Abschnitt 3.4.3.2.

Der erste Schritt der Sprachimplementierung ist die Definition ihrer Struktur und die Spezifikation der verfügbaren Fenster, siehe Abbildung 5.1. Der Aufbau der Programme einer visuellen Sprache wird in VLEli durch eine kontextfreie Grammatik festgelegt, siehe Abschnitt 4.2. Auch die Spezifikation der Fenstertypen ist im letzten Kapitel diskutiert worden. An dieser Stelle soll nicht weiter auf diese Thematik eingegangen werden. Die Spezifikationen sind hier aufgeführt, weil sie das Verständnis der darauf aufbauenden Spezifikationen erleichtern.

Der nächste Schritt der Sprachimplementierung definiert die graphische Darstellung der Fenster. Der Sprachentwickler kann hierzu Spezifikationsmodule für die Sprachkonstrukte anwenden. Für das Listen-Muster stehen zwei verschiedene Implementierungen zur Verfügung. Das Spezifikationsmodul SimpleList kann einfache Listen implementieren, in der die Elemente horizontal oder vertikal angeordnet sind. Das Modul RecursiveList beschreibt Listen dagegen durch rekursive



```
RULE:          ROOT ::= Sequence END;
RULE:          Sequence ::= Element* END;
RULE pRect:    Element ::= Rectangle END;
RULE pCirc:    Element ::= Circle END;
RULE:          Rectangle ::= END;
RULE:          Circle ::= END;
```

(a) Struktur der VCore-Programme

```
VISUAL root (ROOT):
     TITLE "VCore";
     BUTTON "Rectangle"  INSERTS pRect;
     BUTTON "Circle"     INSERTS pCirc;
END;
```

(b) Fenstertypen

Abbildung 5.1.: Grundlegende Spezifikation der VCore-Entwicklungsumgebung

Anwendung dynamischer Zeichnungen.[1] Es kann somit auch komplexere Darstellungen erzeugen, bei denen für Folgenelemente zusätzliche Zeichnungsteile benötigt werden.

Die Listendarstellung soll hier zunächst durch das Spezifikationsmodul SimpleList erzeugt werden. Für die Darstellung der Listenelemente kann das Spezifikationsmodul Terminal benutzt werden, mit dem sich beliebige, nicht weiter strukturierte Zeichnungselemente erzeugen lassen.

Um die Spezifikationen eines Spezifikationsmoduls mit den Konstrukten einer Sprache zu assoziieren, exportieren die Module *Symbolrollen*. Symbolrollen sind Stellvertreter für Sprachkonstrukte im Spezifikationsmodul. Ihnen sind Attributberechnungen zugeordnet, die gemeinsam die graphische Darstellung eines Sprachkonstrukts erzeugen und die Editieroperationen definieren. Um das Spezifikationsmodul SimpleList zu benutzen, muß ein Sprachentwerfer beispielsweise drei Symbolrollen zur Kenntnis nehmen: Die Symbolrolle ListNode kennzeichnet die Sprachkonstrukte, die aufgereiht werden sollen, die Symbolrolle ListContext markiert den Kontext, in dem neue Listenelemente hinzugefügt werden können und die Symbolrolle SimpleList repräsentiert die vollständige Folge.

Die Berechnungen, die durch diese Symbolrollen gekapselt werden, kann ein Sprachentwerfer an Symbole seiner Sprache *vererben*. Abbildung 5.2(a) zeigt dies für die Symbolrollen des SimpleList-Moduls, Abbildung 5.2(b) für die Elemente der Folge. Die Symbole `Sequence`, `Element`, `Rectangle` und `Circle` kommen dabei direkt in der Grammatik (siehe Abb. 5.1(a)) vor, das Symbol `_List_Elements` wird implizit durch das Konstrukt `Element*` erzeugt.

Im einfachen Fall reicht die einfache Vererbung bereits aus, um ein Spezifikationsmodul anzuwenden. Hierbei werden für zahlreiche Eigenschaften eines Sprachkonstrukts Standardwerte gewählt, die zur Anpassung bestimmter Eigenschaften überschrieben werden können. Abbildung 5.2(b) zeigt dies für das Rechteck-Symbol: Das Spezifikationsmodul Terminal zeichnet normalerweise

[1] Siehe Abschnitt 5.1.4

```
SYMBOL Sequence INHERITS VP_SimpleList END;
SYMBOL _List_Element INHERITS VP_ListContext END;
SYMBOL Element INHERITS VP_ListNode END;
```

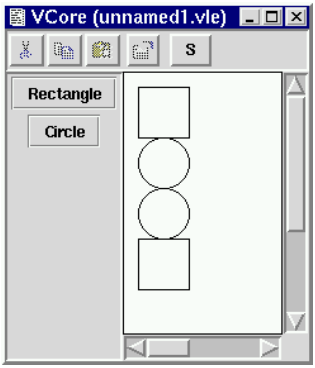
(a) Anwendung des SimpleList-Moduls

```
SYMBOL Rectangle INHERITS VP_Terminal
COMPUTE
   THIS.VP_OptFigure = "rectangle";
END;

SYMBOL Circle INHERITS VP_Terminal END;
```

(b) Spezifikation der elementaren Sprachkonstrukte

```
$/Visual/VisPattern.gnrc+instance=root:inst
```

(c) Importieren der Spezifikationsmodul-Bibliothek



(d) Generierter VCore-Editor

Abbildung 5.2.: Spezifikation der graphischen Darstellung von VCore

einen Kreis. Durch Überschreiben der Berechnung des OptFigure-Attributs kann jedoch auch ein Rechteck erzeugt werden.[2]

Abbildung 5.2(c) zeigt den letzten Teil der VCore-Spezifikation: Die hier angegebene Zeile instanziiert alle Spezifikationsmodule, die zur Erzeugung der graphischen Darstellung eines Fensters vorhanden sind. Das ist möglich, da die Module außer der Angabe des Fenstertyps (hier root, siehe Abb. 5.1(b)) keine weiteren generischen Parameter benötigen. In Abbildung 5.2(d) ist der generierte Editor mit einem Beispielprogramm abgebildet.

5.1.1.2. Änderungen der graphischen Darstellung

Um mit den Spezifikationsmodulen SimpleList und Terminal verschiedenartige graphische Darstellungen zu erzeugen, verfügen die Module über eine ganze Reihe verschiedener Parameter. Beim Modul Terminal lassen sich beispielsweise Form, Farbe, Größe und Linienstärke beeinflussen und auch mehrere Zeichnungselemente verwenden. Bei der Liste können optional ein Rahmen, eine Grundlinie oder Trennungslinien hinzugefügt werden. Weiterhin kann der Abstand und die Ausrichtung der Elemente sowie die Ausbreitungsrichtung der Liste eingestellt werden.

[2] Um die Symbolrollen und Attribute von denen anderer Modulbibliotheken zu unterscheiden, wurde jeweils der Präfix VP_ vor den Namen verwendet, der in den Beschreibungen zugunsten der Lesbarkeit weggelassen wurde.

Mit Hilfe von Attributberechnungen stellen die Spezifikationsmodule Standardeinstellungen für ihre Parameter zur Verfügung. Diese kann ein Sprachentwerfer übernehmen und so ein Spezifikationsmodul auf einfache Weise anwenden. Das zeigt beispielsweise Abbildung 5.2(a): hier ist nur die Nennung der Symbolrollen erforderlich. Der Sprachentwerfer kann aber auch einige der Einstellungen durch andere Angaben überschreiben, um andere Darstellungen zu erzeugen. Technisch gesehen überschreibt er damit einige der Attributberechnungen einer Symbolrolle und erbt nur noch die anderen, nicht überschriebenen Berechnungen.

Abbildung 5.3 zeigt 5 Darstellungsalternativen für die Sprache VCore und gibt damit einen Eindruck von den vielfältigen Einsatzmöglichkeiten der Spezifikationsmodule. Die Abbildungen 5.3(a) und 5.3(b) wurden dabei durch die Spezifikationsmodule SimpleList und Terminal erzeugt. Für Abb. 5.3(c) wurde statt Terminal die Symbolrolle IconTerminal des Terminal-Moduls verwendet. IconTerminal unterstützt die Verwendung von Bildern und ermittelt automatisch deren Größe. Die restlichen beiden Abbildungen benutzen das Modul RecursiveList statt SimpleList. Alle Darstellungsalternativen verwenden dieselbe strukturelle Abstraktion und stellen dasselbe visuelle Programm dar, das auch Abbildung 5.2(d) zeigt.

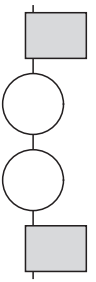
In der Darstellung aus Abbildung 5.3(a) wurde eine schwarze Grundlinie hinzugefügt, ein Abstand von 10 Pixeln zwischen den Listenelementen gewählt und für die horizontale Ausrichtung der Listenelemente Referenzpunkte benutzt. Zusätzlich wurden die Recktecke grau gefüllt und ein wenig verkleinert. Die Spezifikation für diese Darstellung (in der Abbildung links) macht deutlich, daß hierfür lediglich einige Attribute der Symbolrollen zu überschreiben sind. Da die Einstellungen weitgehend orthogonal zueinander sind, kann der Sprachentwerfer mit ihnen experimentieren, bis sich die von ihm gewünschte Darstellung ergibt.

In Abbildung 5.3(b) wurde ein Rahmen mit abgerundeten Ecken für die Liste hinzugefügt und es wurden gestrichelte Trennlinien benutzt. So entsteht eine graphische Darstellung, die an die AND-Superstates der Zustandsdiagramme erinnert. In diesem Beispiel wurden zusätzliche Attributberechnungen hinzugefügt, mit denen die Größe der Listenelemente anhand ihrer Position in der Liste bestimmt wird. Die Darstellung in Abbildung 5.3(c) verwendet ebenfalls Rahmen, hier jedoch ohne äußere Begrenzungslinie. Der Rahmen der äußeren Liste ist grau, die Rahmen für die Elemente weiß gefüllt.

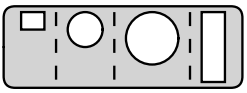
Die Darstellungen, die sich mit dem SimpleList-Modul erzeugen lassen, gehen schon recht weit, haben aber gewisse Einschränkungen, die aus ihrer einfachen Spezifikation und Implementierung resultieren. Um komplexere Darstellungen zu erzeugen, kann man das Modul SimpleList gegen das Modul RecursiveList austauschen. Zur Umstellung braucht hierbei lediglich die Symbolrolle SimpleList gegen RecursiveList ausgetauscht zu werden, wie dies Abbildung 5.3(e) skizziert.

In Abbildung 5.3 sind zwei Anwendungsbeispiele für das Modul RecursiveList enthalten. Das Beispiel aus Abb. 5.3(d) verwendet eine diagonale Ausbreitungs-

```
SYMBOL Sequence INHERITS VP_SimpleList COMPUTE
   THIS.VP_OptElementDistance = 10;
   THIS.VP_OptAlignInWidth = VisAlignBuoy;
   THIS.VP_OptBaselineStyle = "-outline black";
END;

SYMBOL Rectangle INHERITS VP_Terminal COMPUTE
   THIS.VP_OptSize = VisIntVector(40,30);
   THIS.VP_OptFigure = "rectangle";
   THIS.VP_OptFigureStyle="-fill lightgrey";
   THIS.VP_OptAlignmentPoint = VisIntVector(5,5);
END;

SYMBOL Circle INHERITS VP_Terminal COMPUTE
   THIS.VP_OptAlignmentPoint = VisIntVector(20,20);
END;
```

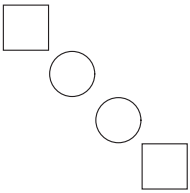


(a)

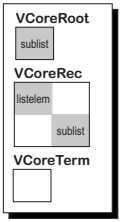


(b)

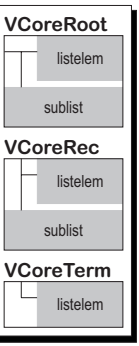
(c)

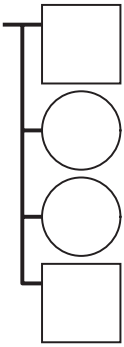


(d)




```
SYMBOL Sequence: simple: int;
SYMBOL Sequence INHERITS VP_RecursiveList COMPUTE
   THIS.simple = LT(THIS.VP_ListElementCount, 2);
   THIS.VP_OblRoot = IF (THIS.simple, ADDROF(VCoreRoot1),
                         ADDROF(VCoreRoot));
   THIS.VP_OblRec = IF (THIS.simple,ADDROF(VCoreRec1),
                        ADDROF(VCoreRec));
   THIS.VP_OblTerm = IF (THIS.simple, ADDROF(VCoreTerm1),
                         ADDROF(VCoreTerm));
END;

SYMBOL _List_Element INHERITS VP_ListContext END;
SYMBOL Element INHERITS VP_ListNode END;

SYMBOL Rectangle INHERITS VP_Terminal COMPUTE
   THIS.VP_OptFigure = "rectangle";
   THIS.VP_OptAlignmentPoint = VisIntVector(0, 10);
END;
SYMBOL Circle INHERITS VP_Terminal COMPUTE
   THIS.VP_OptAlignmentPoint = VisIntVector(0, 20);
END;
```





(e)

Abbildung 5.3.: Alternative graphische Darstellungen für VCore (einige mit Teilen der Spezifikation)

richtung der Liste, in Abb 5.3(e) werden Folgen durch einen kleinen, vertikal angeordneten Baum dargestellt.

Um die Darstellungen zu erzeugen, werden beim Modul RecursiveList dynamische Zeichnungen angegeben. In den Beispielen sind die verwendeten Zeichnungen jeweils neben den visuellen Programmen skizziert. Die Zeichnungen enthalten Platzhalter für die Listenelemente und evtl. zusätzlich benötigte graphische Elemente. Sie werden entsprechend dem Aufbau der darzustellenden Folge ineinander eingesetzt, wie dies auch schon in Abbildung 3.12 (Seite 82) verdeutlicht wurde.

5.1.1.3. Sprachänderungen

Ein Sprachentwerfer muß oft nicht nur mit der graphischen Darstellung einer visuellen Sprache experimentieren, auch die Struktur einer neuen Sprache steht u.U. noch nicht endgültig fest und wird weiterentwickelt. In diesem Zusammenhang ist es wichtig, daß sich Sprachänderungen in einfacher Weise realisieren lassen, möglichst ohne Änderungen an bereits existierenden Spezifikationen zu erfordern.

Ein hohes Maß an Unabhängigkeit der verschiedenen Teile der Spezifikation ist hier auf die Verwendung der Symbolattributierungen des Liga-Systems zurückzuführen - sie bewirken dasselbe auch im Aufgabenbereich der Analyse textueller Sprachen. Darauf bin ich bereits in Abschnitt 4.1.2 eingegangen. Hinzu kommt hier noch, daß sich die Spezifikationsmodule aufgrund ihrer Schnittstellen gut kombinieren lassen.

An dieser Stelle soll ein konkretes Beispiel für die Durchführung einer Sprachänderung gezeigt werden. Die Sprache VCore soll dazu so erweitert werden, daß sie auch hierarchische Folgen unterstützt, d.h. eine Folge kann nun auch als Element in einer Folge auftreten. Zur Realisierung dieser Änderung kann der Grammatik von VCore (siehe Abb. 5.1(a)) eine weitere Produktion hinzugefügt werden:

```
RULE pList:  Element ::= Sequence END;
```

Um diese Produktion anwenden zu können, muß ein zusätzliches Bedienelement in das Editorfenster eingefügt werden. Dazu ist eine zusätzliche Zeile in der Spezifikation aus Abbildung 5.1(b) erforderlich:

```
BUTTON "Seq"  INSERTS pList;
```

Bei der Spezifikation der graphischen Darstellung ist keine Änderung notwendig. Abbildung 5.4 enthält zwei Beispiele für Darstellungen, die aus den Spezifikationen für Abbildung 5.3 durch Hinzufügen der beiden soeben angegebenen Zeilen erzeugt wurden. Auch die Spezifikationen der anderen Darstellungsalternativen sind mit den Änderungen lauffähig. Hierbei sind jedoch zusätzliche Änderungen erforderlich, weil hier die Hierarchie durch die Art der Darstellung nicht erkennbar ist.

5.1.2. Kombination der Spezifikationsmodule

Wie im vorigen Abschnitt gezeigt, lassen sich die Spezifikationsmodule miteinander kombinieren, um eine visuelle Sprache zu implementieren. Für die Sprachimplementierung ist dabei die Flexibilität von entscheidender Bedeutung.

In den Beispielen aus dem vorigen Abschnitt wurde dies bereits deutlich: beim SimpleList-Modul kann die graphische Darstellung der Elemente nicht nur durch das Terminal-Modul, sondern auch durch andere Spezifikationsmodule erzeugt werden. Beim Beispiel aus dem vorigen Abschnitt hat sich dies positiv ausgewirkt: Für die beschriebene Sprachänderung war keine Anpassung bei den Anwendungen der Spezifikationsmodule erforderlich, weil die graphische Darstellung der Elemente auch wiederum vom SimpleList-Modul erzeugt werden kann.

Damit die Berechnungen verschiedener Spezifikationsmodule miteinander kooperieren können, muß eine Modulschnittstelle vereinbart werden. Im ersten Unterabschnitt wird dazu festgelegt, welche Layoutinformationen in welchen Attributen gespeichert werden und wie auf diese Informationen zugegriffen werden darf. In zweiten und dritten Unterabschnitt werden die Festlegungen illustriert, indem exemplarisch Layoutberechnungen für verschiedene Module realisiert werden. Abschließend zeige ich, wie die Module, die Informationen durch Linien visualisieren, mit den Berechnungen der anderen Module zusammenpassen.

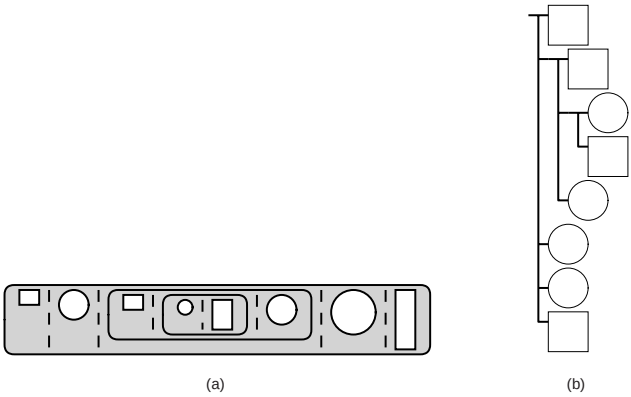


Abbildung 5.4.: VCore-Programme mit Hierarchien

5.1.2.1. Schnittstelle für geschachtelte Darstellungen

Eine der Aufgaben der Spezifikationsmodule ist es, die Anordnung der graphischen Darstellung zu bestimmen. Anders als bei den anderen Aufgaben sind dazu Berechnungen erforderlich, die sich über weite Teile des Strukturbaums für ein visuelles Programm erstrecken. Eine einzelne Anwendung eines Spezifikationsmoduls steuert dabei i.a. lediglich einen Teil der Berechnungen bei. Alle Layoutberechnungen müssen aber an ihren Schnittstellen zusammenpassen, damit sich eine konsistente Darstellung ergibt.

Um die Layoutberechnungen verschiedener Spezifikationsmodule miteinander zu kombinieren, muß ein Modul jeweils die Informationen zur Verfügung stellen, die von anderen Modulen benötigt werden. Weiterhin sollten alle Module (aus einer Gruppe) dasselbe Verhalten an ihren Schnittstellen[3] aufweisen, damit diese Module flexibel gegeneinander ausgetauscht werden können. Das kann erreicht werden, indem die Abhängigkeiten der Layoutberechnung für eine Gruppe kombinierbarer Module in groben Zügen festgelegt wird.

Als Grundgerüst für eine solche Festlegung bietet sich die Box-and-Glue-Methode (siehe Abschnitt 4.4.1.1 auf Seite 127) an. Dabei wird das Layout einer geschachtelten Darstellung berechnet, indem zunächst die Größe der graphischen Strukturen von den inneren zu den umgebenden Strukturen bestimmt und anschließend deren Positionen von den äußeren zu den inneren Strukturen hin berechnet wird. Das eignet sich insbesondere für visuelle Sprachen, bei denen Hierarchien durch ineinander geschachtelte rechteckige Bereiche visualisiert werden. Mit einer solchen Vorgehensweise lassen sich z.B. alle in Kapitel 3 untersuchten Sprachen implementieren.

Wichtig ist im weiteren, daß die Box-and-Glue-Methode erweitert wird, um auch die Anordnung graphischer Objekte durch Constraints in den Berechnungsablauf zu integrieren. Auf diese Thematik bin ich bereits in Abschnitt 4.4.2.2 eingegangen. Wenn man die durch Constraints änderbaren Layoutparameter in gewisser Weise beschränkt, kann man ein Constraint-Netzwerk durch Attributberechnungen von den Blättern zur Wurzel hin berechnen und dieses einmalig (für eine graphische Darstellung) durch einen Constraint-Solver lösen lassen.

Schließlich kann auch das Layout für Linien in dieses Verfahren der Layoutberechnung integriert werden. Dabei wird im Kontext der Linien die Position der verbundenen Zeichnungsteile benötigt, so daß man sich im allgemeinen damit begnügen muß, daß der Verlauf der Linien bei der Anordnung anderer Teile der Darstellung unberücksichtigt bleibt. Dies ist kein allzu großes Problem, weil dies zum einen für die Implementierung praktisch eingesetzter Sprachen ausreicht. Zum

[3] "Verhalten" bedeutet im Zusammenhang mit Symbolberechnungen einer attributierten Grammatik, daß bestimmte Informationen in bestimmten Attributen verfügbar sind und daß zwischen den Berechnungen, die diese Informationen ermitteln, bestimmte festgelegte Abhängigkeitsmuster eingehalten werden.

anderen gelten die Festlegungen lediglich für das Schnittstellenverhalten der Spezifikationsmodule. Für die Implementierung eines Moduls für das Graph-Muster könnte durchaus auch die Linienführung der Kanten bei der Positionsbestimmung der Knoten mitberücksichtigt werden.

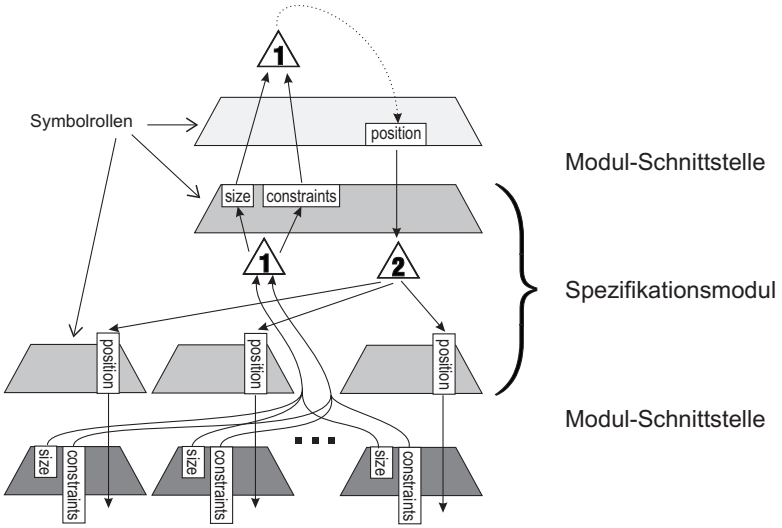
Im folgenden beschreibe ich die Schnittstelle der Spezifikationsmodule, die hier durch Attribute und durch Abhängigkeiten zwischen diesen Attributen gegeben ist. Der allgemeine Fall ist in Abbildung 5.5(a) skizziert. In der Abbildung ist in der Mitte ein Spezifikationsmodul zu sehen. Es hat nach Oben und Unten in der Darstellung Schnittstellen zu anderen Spezifikationsmodulen. Innerhalb des Moduls sind trapezförmig einige Symbolrollen eingezeichnet, die wiederum einige benannte Attribute enthalten. Die Pfeile in der Abbildung kennzeichnen Abhängigkeiten zwischen den Attributen. Die Nummern kennzeichnen die beiden Phasen des Box-And-Glue-Layoutverfahrens.

Die Schnittstelle zwischen zwei Spezifikationsmodulen wird durch die drei Attribute mit den Namen size, constraints und position gebildet. Das Attribut size enthält Layoutinformationen, die im umgebenden Kontext benötigt werden. Das ist die Größe dieser Bereiche, kann aber auch weitere Informationen wie z.B. die Relativposition einer horizontalen und vertikalen Grundlinie oder die Form dieser Bereiche umfassen.[4] Das Attribut constraints der Modulschnittstelle beschreibt das Constraint-Netzwerk, das die Legalität des Layouts für den betrachteten Teilbaum beschreibt.

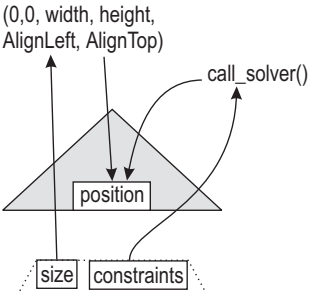
Das Attribut position beschreibt schließlich den für die graphische Darstellung eines Spezifikationsmoduls verfügbaren Platz, das ist die absolute Position der linken oberen Ecke und die Ausdehnung dieses Bereichs, sowie die gewünschte Ausrichtung. Dieses Attribut wird – anders als die beiden anderen, von der Wurzel zu den Blättern des Strukturbaums hin, von einem Spezifikationsmodul jeweils für die in der Darstellung enthaltenen Bereiche anderer Module berechnet. Es ist in Abbildung 5.5(a) deshalb jeweils oberhalb der Schnittstelle eingezeichnet.

Die Layoutberechnung wird in zwei Phasen ausgeführt, welche den Phasen des Box-and-Glue-Layouts entsprechen. In der ersten Phase werden jedoch zusätzlich zu den Layoutinformationen die Constraint-Netzwerke der Unterbäume von den Blättern zur Wurzel hin berechnet. Nach Abschluß der ersten Phase wird dann der Constraint-Solver gestartet, wobei implizit ggf. einige der Layoutattribute geändert werden. Der Abschluß der Prüfung der Constraints wird gekennzeichnet, indem das Attribut position des Wurzelknotens auf den Ursprung des Koordinatensystems gesetzt wird, siehe Abbildung 5.5(b). Dies ist für die korrekte Berechnungsreihenfolge beim Constraintbasierten Layout wichtig: wenn eine Berechnung von der absoluten Position abhängt, so kann davon ausgegangen werden, daß der Constraint-Solver das Layout bereits korrigiert hat.

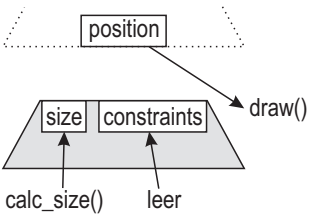
[4] Der intuitive Name size wurde einem allgemeineren Namen wie layoutinfo vorgezogen.



(a) Schnittstelle bei hierarchischen Strukturen

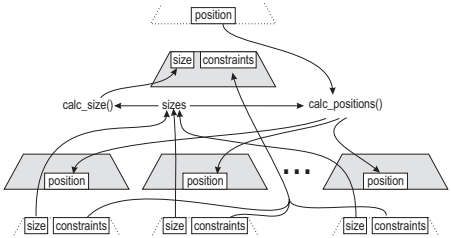


(b) Attributberechnungen an der Wurzel



(c) Attributberechnungen bei Terminalen

Abbildung 5.5.: Schnittstelle für geschachtelte Darstellungen



(a) Attributabhängigkeiten für die Box-and-Glue-Methode

```
calc_sizes(sizes, elemdistance)
{  n := length(sizes);
   totalwidth = max(sizes[i].width);
   totalheight = sum(sizes[i].height) + (n-1) * elemdistance;
   return (totalwidth, totalheight);
}

calc_positions(sizes, position, alignment, elemdistance)
{  n := length(sizes);
   px := calc_x_alignment(position, totalwidth);
   sonposition[0].y := calc_y_alignment(position, totalheight);
   for i := 1 .. n-1
         sonpos[i].y := sonpos[i-1].y + sizes[i-1].height +
                        elemdistance;
   for i := 0 .. n-1
         sonposition[i].x := px;
         sonposition[i].width = totalwidth;
         sonposition[i].height = sizes[i].height;
         sonposition[i].align_x = alignment;
         sonposition[i].align_y = AlignTop;
   return sonposition;
}
```

(b) Pseudocode für das SimpleList-Modul (vereinfacht)

Abbildung 5.6.: Umsetzung des Box-And-Glue-Verfahrens, mit Beispiel

Die folgenden Abschnitten konkretisieren dieses allgemeine Verfahren für die unterstützten Layoutverfahren.

5.1.2.2. Box-and-Glue-Layout

Die Festlegungen der Modulschnittstelle sind zu einem Teil aus den Berechnungsschritten hervorgegangen, mit denen die Box-and-Glue-Methode der Layoutberechnung implementiert wird. Es ist deshalb relativ leicht, auf Basis dieser Festlegungen Spezifikationsmodule zu definieren, die diese Methode der Layoutberechnung benutzen.

Abbildung 5.6(a) enthält eine für das Box-and-Glue-Verfahren angepaßte Version von Abb. 5.5(a). Das abgebildete Spezifikationsmodul hat dasselbe Schnittstellenverhalten wie das allgemeine, führt jedoch die Abhängigkeiten innerhalb des Moduls genauer aus. Um das Layout einer graphischen Darstellung festzulegen, müssen noch die Prozeduren calc_size und calc_positions ausformuliert werden. Hierzu sind je nach Layoutaufgabe in den Spezifikationsmodulen unterschiedliche Berechnungen erforderlich.

Abbildung 5.6(b) enthält ein, in Pseudocode angegebenes Beispiel für diese Prozeduren. Hier wird die Anordnung für eine vereinfachte Version des SimpleList-Moduls berechnet. Die Vereinfachung besteht darin, daß einige der bei SimpleList änderbaren Layoutparameter hier festgelegt sind. Das ist keine wesentliche Einschränkung, macht jedoch die Prozeduren in Abbildung 5.6(b) kürzer und einfacher.

Einige der Layoutparameter wurden jedoch beibehalten. Sie werden durch die in den Prozeduren auftretenden Größen elementdistance und alignment beschrieben. Für diese Parameter hält das Modul Standardwerte bereit, die bei der Benutzung des Moduls überschrieben werden können. Die Prozedur calc_size berechnet die Gesamtgröße der Liste, indem sie die maximale Breite der Listenelemente berechnet und ihre Höhen aufsummiert. Das Ergebnis beschreibt die Größe der Liste bei vertikaler Ausrichtung ihrer Elemente. Die Prozedur calc_positions berechnet die Positionen der Listenelemente so, daß ein Listenelement jeweils unter seinem Vorgänger liegt. Die relative X-Position wird dabei in Abhängigkeit von der gewünschten Ausrichtung berechnet.

In der Modul-Schnittstelle existiert zusätzlich das Attribut constraints. Beim Box-And-Glue-Layout treten keine neuen Constraints auf. Deshalb werden hier lediglich die Constraint-Netzwerke der Teilbäume kombiniert und das resultierende Gesamtnetzwerk weitergegeben.

5.1.2.3. Constraintbasiertes Layout

Beim constraintbasierten Layout ist dieser Teil der Modulschnittstelle von sehr viel größerer Bedeutung. Auch hier werden die Constraints für die Unterbäume zusammengefaßt. Hinzu kommen jedoch weitere Constraints, die beispielsweise die Überlappung der Unterbereiche ausschließen oder auf korrekte Schachtelung im Rahmen der äußeren Darstellung prüfen. Die Abhängigkeiten der dazu erforderlichen Attributberechnungen beschreibt Abbildung 5.7.

Das Constraint-Netzwerk wird durch die Prozedur calc_constraints berechnet. Die Prozedur erhält als Parameter die Größe und Relativposition der eingeschachtelten Bereiche (Pfeile von unten) sowie die Größe des umgebenden Konstrukts (Pfeil von oben, z.B. die Ausdehnung des Mengenrahmens). Die Relativpositionen und die Größe des umgebenden Konstrukts sind dabei Werte, die der Benutzer direkt durch Interaktion mit der graphischen Darstellung ändern kann. Diese Angaben werden deshalb in persistenten Attributen gespeichert. Sie sind in der Ab-

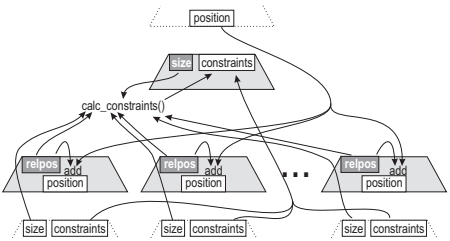


Abbildung 5.7.: Umsetzung des Constraintbasierten Layouts

bildung invers dargestellt. Im Gegensatz zu den berechneten Attributen können die persistenten Attribute im Constraint-Netzwerk durch Variablen repräsentiert werden; der Constraint-Solver kann also nur die Werte dieser Attribute ändern.

Die berechneten Constraints werden im folgenden zusammen mit den anderen Constraints weitergeleitet und im Wurzelkontext schließlich an den Constraint-Solver übergeben. Wenn die Position in der oberen Modulschnittstelle berechnet ist, so wurde der Constraint-Solver aufgerufen und hat ggf. die Werte der persistenten Attribute geändert. Die Positionen eingeschachtelter Bereiche können nun jeweils aus der Relativposition und der Position des umgebenden Kontextes berechnet werden.

Eine Besonderheit ist hierbei zu beachten: Geht man wie oben beschrieben vor, so muß das Constraint-Netzwerk im allgemeinen Fall so definiert werden, daß die Größe des umgebenden Kontextes als konstanter Wert in das Constraint-Netzwerk eingesetzt wird. Ursache dafür ist, daß der Wert dieses Attributs benötigt wird, bevor der Constraint-Solver aufgerufen wird. Das Attribut darf also nicht vom Aufruf des Constraint-Solvers abhängen, was der Fall wäre, wenn sich dessen Wert durch den Aufruf des Constraint-Solvers ändern könnte. Auf diese Problematik bin ich in Abschnitt 4.4.2.2 eingegangen.

Eine Ausnahme hiervon ist jedoch möglich. Wird auch das umgebende Konstrukt constraintbasiert angeordnet, so kann von der allgemeinen Modulschnittstelle abweichend die Größe des umgebenden Konstrukts auch als Variable in das Constraint-Netzwerk beider Module eingesetzt werden. Dadurch werden wechselseitige Abhängigkeiten zwischen den Constraint-Netzwerken dieser Module erzeugt, was sich durch komfortablere Layoutanpassungen auswirkt. Ein Beispiel dazu habe ich in Abbildung 4.16(a) (Seite 135) gezeigt. Das im Rahmen dieser Arbeit implementierte Mengen-Modul unterstützt beide Alternativen.

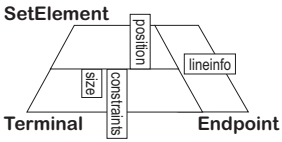
5.1.2.4. Linien

In den graphischen Darstellungen visueller Programme werden außer geschachtelten Strukturen auch oft Linien benutzt, beispielsweise um in datenflußorientierten visuellen Sprachen den Informationsfluß zwischen den Operationen zu beschreiben oder um die Übergänge in Automaten oder Zustandsdiagrammen zu definieren.

Linien verbinden dabei zwei oder mehr Sprachkonstrukte. Ändert der Benutzer die Position eines Sprachkonstrukts, so erwartet er, daß die Linie automatisch an diese Änderung angepaßt wird. Um diese Anforderung zu erfüllen, müssen Informationen über die verbundenen Programmkonstrukte zum Kontext der Linienverbindung transportiert und dort in die Layoutberechnung einbezogen werden.

Ein Teil der Informationen, die benötigt werden, um den Verlauf einer Linie zu berechnen, wird dabei von anderen Spezifikationsmodulen berechnet. Dies

```
SYMBOL Transition
INHERITS VP_SetElement
END;
```




```
SYMBOL Transition
INHERITS VP_Endpoint
COMPUTE
   THIS.VP_OptToTheEnd = 0;
END;
```

```
SYMBOL Transition
INHERITS VP_Terminal
COMPUTE
   THIS.VP_OptSize = VisIntVector(10,40);
   THIS.VP_OptFigure = "rectangle";
   THIS.VP_OptFigureStyle = "-fill black";
END;
```

Abbildung 5.8.: Einsatz von Mehrfachvererbung am Beispiel der Petrinetze

sind beispielsweise die Position und Größe der verbundenen Sprachkonstrukte. Um diese Informationen in den Kontext der Linien zu transportieren, müssen im Kontext der verbindbaren Programmkonstrukte zusätzliche Attributberechnungen ausgeführt werden.

Diese zusätzlichen Attributberechnungen lassen sich ebenfalls in einer Symbolrolle kapseln. Diese Symbolrolle kann einem Sprachkonstrukt *zusätzlich* zu anderen Symbolrollen vererbt werden, um es zu einem potentiellen Endpunkt einer Linie zu machen. Diese Situation läßt sich am Beispiel der Petri-Netze erklären: Das Sprachkonstrukt Transition erbt hierbei drei Symbolrollen: die Symbolrolle Terminal erzeugt seine graphische Darstellung. Es liefert Informationen über die Größe und benötigt Informationen über die Position, an der die Darstellung erscheinen soll. Diese Informationen werden von der Symbolrolle SetElement geliefert. SetElement spezifiziert die Transition als Element einer Menge, deren Darstellung in einem Mengenrahmen enthalten ist und die sich nicht mit den Darstellungen anderer Elemente überschneidet. Die Symbolrolle Endpoint markiert die Transition schließlich als potentiellen Endpunkt von Übergängen und benötigt dazu Informationen über Größe, Form und Position.

Diese Situation ist in Abbildung 5.8 graphisch dargestellt. Das Grammatiksymbol Transition erbt die o.a. drei Symbolrollen, wobei jede dieser Rollen Attributberechnungen kapselt. Diese Attributberechnungen können, wie im Bild gezeigt, überschrieben werden, um das Standardlayout und Standardverhalten anzupassen. Bei der Symbolrolle Endpoint bestimmt so beispielsweise der Wert des Attributs ToTheEnd, ob die Linie zum Mittelpunkt oder zum Rand der Transition gezeichnet werden soll. Ein weiteres Attribut bestimmt die äußere Form der Transition, welche benötigt wird, wenn eine Linie bis zum Rand der Darstellung gezogen werden soll. In Abb. 5.8 wurde hier der Defaultwert der Symbolrolle Endpoint übernommen. Diese Informationen und weitere Informationen fließen

bei der Symbolrolle Endpoint neben Position und Größe in die Informationen ein, die im Attribut lineinfo zusammengefaßt werden

Ein weiterer Ansatzpunkt, der zu Interaktionen zwischen Modulen für Linien und Modulen für ineinander geschachtelte Darstellungen führt, ist die Beschriftung von Linien. Im allgemeinen Fall sind hier nicht nur Texte, sondern allgemeiner auch andere graphische Darstellungen sinnvoll. Ein Beispiel liefert die Sprache der UML-Klassendiagramme, bei der an den Linienenden auch komplexere Darstellungen möglich sind, siehe beispielsweise die Assoziation zwischen Class1 und Class4 in Abbildung 3.14(b) (Seite 85).

Um allgemeine Beschriftungen zu implementieren, kann eine Symbolrolle zur Verfügung gestellt werden, die für die Module, die hierarchische Strukturen visualisieren, an Stelle der Wurzel verwendet werden kann. Das erlaubt es, die Beschriftung einer Linie durch Anwendung des Terminalmoduls, ggf. aber auch durch andere Module wie das Formularmodul oder das Listen-Modul zu erzeugen.

In Abbildung 5.9, die schematisch die Attributabhängigkeiten für Linien-

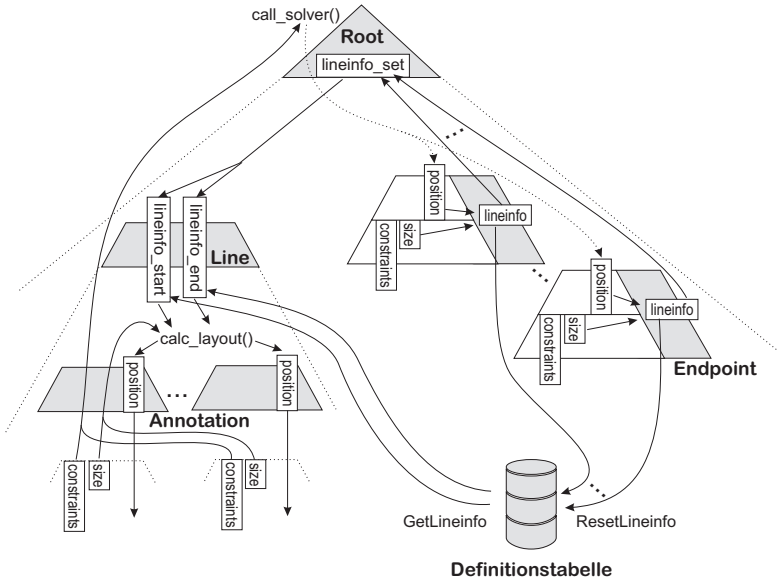


Abbildung 5.9.: Attributabhängigkeiten von Spezifikationsmodulen für Linien

Module wiedergibt, hat diese Symbolrolle den Namen Annotation. Sie wird einer Linienbeschriftung zusätzlich zu anderen Symbolrollen vererbt. Die Berechnungen von Annotation interagieren dabei mit den Layoutberechnungen des Linien-Moduls. Diese ermitteln aus der Größe der Beschriftungen sowie aus den Positionen der Linienendpunkte die Positionen der Beschriftungen.

An den Beschriftungen fallen im allgemeinen Fall zusätzlich Constraint-Netzwerke an, die die Gültigkeit des Layouts innerhalb der Linienbeschriftung beschreiben. Diese Constraint-Netzwerke müssen in den Wurzelkontext des Strukturbaums transportiert werden, um dort in den Aufruf des Constraint-Solvers einzugehen. Ggf. kann ein Linienmodul diese Constraint-Netzwerke ergänzen, um zusätzliche Layoutbedingungen für die Linienbeschriftung festzulegen, etwa "darf nicht weiter als x von der Linie entfernt sein".

In Abbildung 5.9 ist weiterhin der Transport der lineinfo-Attribute zu den Linien dargestellt. Die Informationen werden dazu in die Definitionstabelle eingetragen und im Kontext einer Linie ausgelesen. Die Abhängigkeiten im Strukturbaum steuern lediglich den Ablauf der Berechnungsreihenfolge. Um Informationen über die Definitionstabelle zu transportieren, werden im Kontext der Endpunkte und im Kontext der Linien Referenzen auf Einträge der Definitionstabelle gespeichert. Die Editieroperationen der Linien werden auf Änderungen dieser Attribute zurückgeführt. Dies zeigt beispielsweise Abbildung 4.20 (Seite 148) genauer.

5.1.3. Symbolrollendiagramme

In den letzten Abschnitten habe ich gezeigt, daß Spezifikationsmodule angewendet werden, indem Symbolrollen an die Grammatiksymbole einer Sprache vererbt werden. Dabei sind bestimmte Randbedingungen einzuhalten, damit die Berechnungen der Symbolrollen in der gewünschten Weise miteinander verzahnt werden. Ein Beispiel bildet das Spezifikationsmodul SimpleList. Zu seiner Benutzung müssen die Symbolrollen SimpleList, ListContext und ListNode so vererbt werden, daß

- ListContext eindeutig im Unterbaum unter SimpleList auftritt und einen Listenkontext repräsentiert,
- ListNode beliebig oft, aber ohne weitere Kettenproduktionen im Unterbaum unter ListContext auftritt,
- und eindeutig unterhalb von ListNode eine Symbolrolle angewendet wird, die die Modulschnittstelle für geschachtelte Darstellungen aus Abschnitt 5.1.2.1 implementiert.

Diese Beschreibung ist textuell, wenn überhaupt, so nur schwer verständlich und ist an dieser Stelle nur aufgeführt, um eine visuelle Notation für derartige

Strukturbedingungen zu motivieren. Dieser habe ich den Namen *Symbolrollendiagramme* gegeben, da die Diagramme die Abhängigkeiten der Symbolrollen eines Spezifikationsmoduls beschreiben. Mit diesen Diagrammen können die Konsistenzbedingungen, die für die Anwendung der Spezifikationsmodule beachtet werden müssen, auf einfache Art dokumentiert werden. Das erleichtert es einem Sprachentwerfer, die zur Lösung einer Visualisierungsaufgabe passenden Symbolrollen zu identifizieren und richtig anzuwenden.

Ein weiterer Aspekt, der sich gut anhand der Symbolrollendiagramme diskutieren läßt, ist die Art der Randbedingungen, die ein Sprachentwerfer bei der Anwendung der Spezifikationsmodule berücksichtigen muß. Im weiteren können die Diagramme auch eine Grundlage für eine visuelle Spezifikation visueller Sprachen bilden.

Die Notation der Symbolrollendiagramme hat Bezüge zu den UML-Klassendiagrammen [Rational Software Corporation u.a. 1997] und zu einer graphischen Notation für Strukturbäume [Kastens 1990]. Aus den UML-Klassendiagrammen wurde die Notation der Symbolrollen und der Assoziationen entlehnt, aus der Darstellung der Strukturbäume wurde die Notation für Teilbäume und Layoutkonventionen übernommen.

Abbildung 5.10 zeigt zwei Beispiele für solche Diagramme. Die Knoten repräsentieren die Symbolrollen eines Spezifikationsmoduls, hier des Spezifikationsmoduls SimpleList. Attribute, die überschrieben werden können, um die graphische Darstellung eines Spezifikationsmoduls oder dessen Verhalten beim Editieren zu beeinflussen, sind dabei optional im unteren Teil einer Symbolrolle angegeben. Dies ist hier nur für eines der beiden Diagramme geschehen.

Um das Spezifikationsmodul SimpleList nach dem Symbolrollendiagramm aus Abbildung 5.10(a) anzuwenden, müssen die angegebenen drei Symbolrollen an Grammatiksymbole vererbt werden. Für diese Symbolrollen gelten bestimmte Bedingungen, die im folgenden anhand der Abbildung erläutert werden.[5]

Die Kanten in Abbildung 5.10 visualisieren die Beziehungen, die zwischen den Grammatiksymbolen einer Sprache bestehen müssen, um das Spezifikationsmodul anzuwenden. Durch die Linie zwischen SimpleList und ListContext in Abbildung 5.10(a) wird angedeutet, daß das Grammatiksymbol, das die Rolle ListContext erbt, in jedem Strukturbaum eindeutig im Unterbaum unter dem Symbol, das die Rolle SimpleList erbt, identifizierbar sein muß. Die Punkte bedeuten dabei, daß zwischen den Symbolrollen weitere Kettenproduktionen vorhanden sein dürfen. Einschränkungen dieser Art sind relativ häufig und werden in der Sprache Lido durch Anwendung des CONSTITUENT-Konstrukts induziert.

In ähnlicher Weise bedeutet der angedeutete Unterbaum zwischen ListContext und ListNode, daß Symbole, die die Rolle ListNode erben, beliebig oft im Unter-

[5] Der Leser kann sich bei Bedarf vergewissern, daß die Bedingungen in der Anwendung aus Abbildung 5.1 und 5.2 eingehalten wurden.

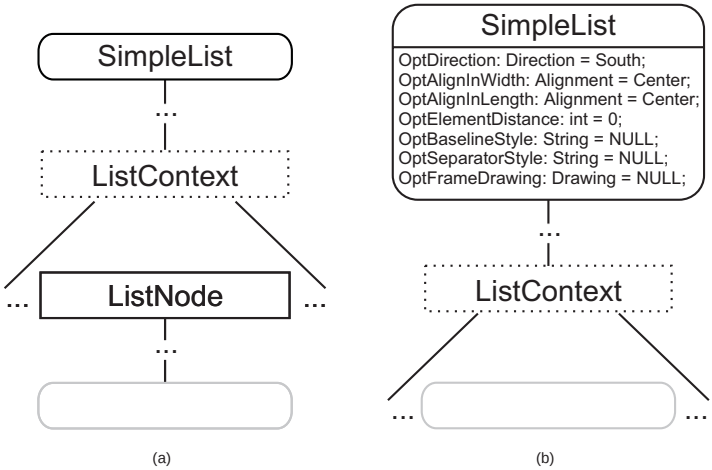


Abbildung 5.10.: Symbolrollendiagramme für das Spezifikationsmodul SimpleList

baum von Symbolen vorkommen dürfen, die die Rolle ListContext erben. Diese müssen hier aber direkte Söhne dieses Kontextes sein, ansonsten wären auch hier Punkte eingezeichnet.

Neben den Kanten gibt es noch verschiedene Darstellungsattribute bei den Symbolrollen. Hat eine Symbolrolle abgerundete Ecken, wie z.B. SimpleList, so implementiert sie die Schnittstelle für geschachtelte Darstellungen aus Abschnitt 5.1.2.1. In ähnlicher Weise bedeutet ein grau umrandetes Rechteck ohne Namen, daß an dieser Stelle eine Symbolrolle eines anderen Moduls verwendet werden muß, die eine solche Schnittstelle zur Verfügung stellt. Dies kann wie im Bild im Unterbaum einer Symbolrolle erforderlich sein. Eine andere Darstellungsalternative impliziert Mehrfachvererbung, siehe Knotentyp (n-iv) in der Zusammenstellung der Bildelemente (Abbildung 5.11).

Eine weitere Art der Darstellung einer Symbolrolle sind gestrichelte Rahmen, wie bei ListContext. Dies signalisiert, daß die Symbolrolle nur an Grammatiksymbole vererbt werden darf, die eine Folge repräsentieren. Solche Symbole werden implizit spezifiziert, indem in der Grammatik einem Symbol ein Wiederholungszeichen, z.B. ein Stern nachgestellt wird. Die dadurch eingeführten Symbole haben den Präfix `_List_`.

Abbildung 5.10 enthält zwei Symbolrollendiagramme, die unterschiedliche Anwendungssituationen für das Spezifikationsmodul SimpleList beschreiben. Das Diagramm in Abb. 5.10(a) kennzeichnet die häufigere Anwendungssituation. Unter manchen Umständen ist es jedoch möglich, auf die Symbolrolle ListNode zu verzichten und in der Programmrepräsentation auf eine Kettenproduktion zu verzichten. Diese Situation ist in Abbildung 5.10(b) dargestellt. Sie eignet sich insbesondere für Listen, die nur eine Verfeinerungsalternative für das Listenelement enthalten.

Bei vielen Spezifikationsmodulen wird die Anwendung nicht nur durch ein einziges, sondern durch eine Menge von Symbolrollendiagrammen beschrieben. Die unterscheidenden Merkmale der verschiedenen Diagramme reichen von leicht unterschiedlichen Ausprägungen der strukturellen Abstraktion bis zu unterschiedlichen Editieroperationen für ein Spezifikationsmodul.

Für einen Sprachentwerfer ist es oft einfach, sich bei einem Spezifikationsmodul für ein Symbolrollendiagramm zu entscheiden, weil den verschiedenen Diagrammen jeweils charakteristische Eigenschaften zugeschrieben werden können. Diese gehen soweit, daß in manchen Situationen auch eine automatische Auswahl des "richtigen" Symbolrollendiagramms möglich wäre.

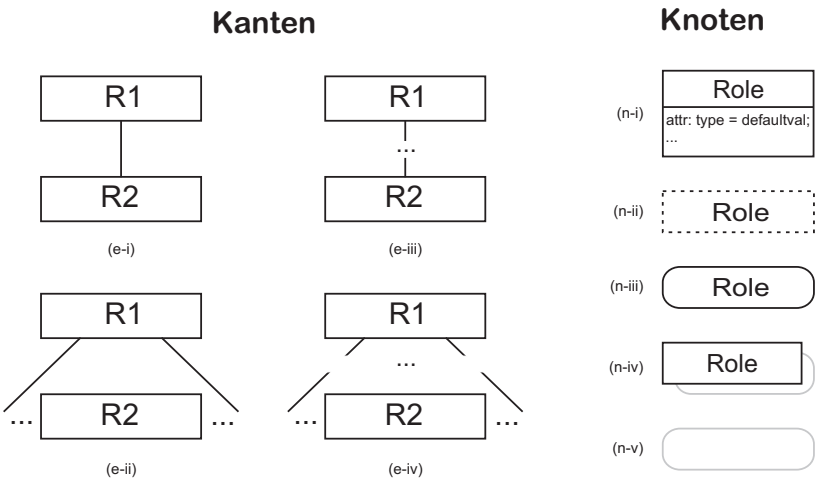


Abbildung 5.11.: Graphische Elemente der Symbolrollendiagramme

5.1.4. Dynamische Zeichnungen

Eine Zeichnung ist ein Teil der graphischen Darstellung eines visuellen Programms. Eine *dynamische* Zeichnung ist eine Spezifikation für eine Zeichnung. Dynamische Zeichnungen bestehen aus graphischen Objekten und aus Platzhaltern, in die andere Zeichnungen eingesetzt werden können. Die wesentliche Eigenschaft der Spezifikation ist die Festlegung des Verhaltens der Zeichnung, wenn der Platzhalter nicht genügend Platz für die eingesetzte Zeichnung bietet und dieser deshalb vergrößert werden muß.

Dynamische Zeichnungen traten als erstes bei der Implementierung des Formular-Musters auf – ein Formular *ist* im Wesentlichen eine dynamische Zeichnung. Seine Tupelelemente beschreiben Zeichnungen, die in die Platzhalter der dynamischen Zeichnung eingefügt werden müssen, um die visuelle Darstellung des Formulars zu erhalten.

Im weiteren Verlauf der Konzeption hat sich jedoch gezeigt, daß dynamische Zeichnungen auch bei anderen Spezifikationsmodulen sinnvoll anwendbar sind. Sie lassen sich verwenden, um Rahmen bei fast allen Spezifikationsmodulen zu erzeugen und um Folgen beim RecursiveList-Modul zu visualisieren. Sie stellen damit eine wichtige Methode dar, um bestimmte Eigenschaften einer visuellen Sprache zu spezifizieren.

Allgemein betrachtet ist eine dynamische Zeichnung eine Funktion, die ein Tupel von Zeichnungen auf eine neue Zeichnung abbildet. Die neue Zeichnung besteht aus den anderen Zeichnungen und enthält ggf. zusätzliche Zeichnungselemente. Im Rahmen ihrer Diplomarbeit haben Schmidt und Schindler [2000] zwei Beschreibungsmodelle für dynamische Zeichnungen implementiert. Beiden Beschreibungsmodellen ist gemeinsam, daß Zeichnungen in Platzhalter eingesetzt werden. Unterschiede bestehen bei den zusätzlichen Zeichnungselementen. Bei den *dynamischen Vektorgraphik-Zeichnungen* sind dies beliebige graphische Primitive (Linien, Kreise, Recktecke), bei den *dynamischen Kachelzeichnungen* dagegen Bitmap-Graphiken einer gemeinsamen Größe. Die dynamischen Kachelzeichnungen können deshalb lediglich in Stufen vergrößert werden.

Die Grundidee zu den dynamischen Zeichnungen stammt aus dem VPE-System [Grant 1998]. Die dortige Spezifikation der Zeichnungen ist jedoch nicht deklarativ, das Verhalten definierter Zeichnungen manchmal schlecht vorhersehbar.

Gemeinsamkeiten bestehen auch zu den graphischen Komponenten bei Unidraw [Vlissides und Linton 1990], die zur Implementierung spezialisierter graphischer Editoren eingesetzt werden. In Unidraw wird die Verbindbarkeit von Komponenten einer graphischen Darstellung durch Verbindungspunkte spezifiziert, die übereinander gelegt werden können. Eine Komponente kann dabei mehrere Verbindungspunkte enthalten. Ihre Positionierung innerhalb der Komponente

wird durch Techniken beschrieben, die man sich als Federn vorstellen kann, die sich zwischen den Verbindungspunkten befinden.

Die Idee zu den Kachelzeichnungen stammt letztendlich aus dem Agentsheets-System [Repenning und Fahlen 1993; Repenning und Sumner 1995]. In Agentsheets werden Kachelzeichnungen benutzt, um miteinander kommunizierende Agenten darzustellen. Dabei ergibt sich in der Regel ein komplexeres Bild, welches aus einzelnen, zueinander passenden Kacheln zusammengesetzt ist. Diese Technik wurde auch in Streets [Kastens und Jung 1998] benutzt, um den Kontrollfluß paralleler Prozesse durch die Metapher "Straße" zu visualisieren. Dabei erforderte der Detailreichtum der Straße den Einsatz vorgefertigter Bildelemente, die im Editor zu einem Ganzen zusammengesetzt werden.

Schließlich werden die Beschreibungselemente, die zur Spezifikation der dynamischen Zeichnungen verwendet werden, auch im Bereich der Codegenerierung für textuelle Sprachen angewendet. Mit der Spezifikationssprache des Werkzeugs PTG [Ptg 2000] kann man Zieltexte durch Textmuster beschreiben, die aus festen Texten und Einfügestellen bestehen. An den Einfügestellen können Informationen textuell eingefügt oder beliebige andere Textmuster angewendet werden.

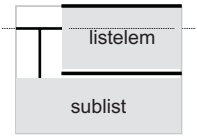
Obwohl sowohl die Spezifikation der Vektorgraphik-Zeichnungen, als auch die der Kachelzeichnungen hier durch textuelle Spezifikationen erfolgt, läßt sich ihr Informationsgehalt gut graphisch beschreiben. In Abbildung 5.12 wird der Aufbau dynamischer Zeichnungen anhand der Baumdarstellung der Sprache VCore (siehe Abb. 5.3(e)) veranschaulicht. Die Teilbilder (a) bis (c) beschreiben in einzelnen Schritten den Aufbau der Zeichnung VCoreRoot, das Teilbild (d) enthält die Spezifikation der anderen beiden Zeichnungen. Die Teilbilder enthalten jeweils rechts die textuelle Spezifikation und links die entsprechende graphische Repräsentation.

In Abbildung 5.12(a) sind zunächst die grundlegenden Vektorgraphik-Teile der Zeichnung spezifiziert. Der graue Rahmen veranschaulicht dabei lediglich den Rand der Zeichnung und gehört nicht zur Vektorgraphik. Die gestrichelte Linie markiert die Grundlinie der Zeichnung. Das ist wichtig, damit hierarchische Listen korrekt dargestellt werden, siehe Abbildung 5.4(b). In der textuellen Spezifikation hat die Grundlinie einer Achse jeweils die Koordinate 0.

Abbildung 5.12(b) ergänzt die Zeichnung um Platzhalter für graphische Darstellungen. Die Platzhalter sind in der graphischen Darstellung durch graue Rechtecke visualisiert, die einen Namen und optional die Spezifikation einer Grundlinie enthalten. Die Namen der Platzhalter werden benötigt, damit sich andere Teile der Spezifikation einer visuellen Sprache auf die Platzhalter beziehen können. Beim RecursiveList-Modul werden beispielsweise in Platzhalter mit dem Namen listelement die Elemente der zu visualisierenden Liste und in Platzhalter mit dem Namen sublist rekursiv andere dynamische Zeichnungen eingesetzt, siehe dazu auch Abbildung 3.12 (Seite 82).

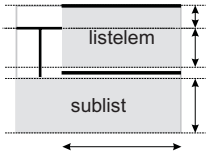
```
figure(2, "line", "-width 2"); point(0, 0); point(20, 0);
figure(2, "line", "-width 2"); point(10, 0); point(10, 10);
```

(a) Grundlegende graphische Elemente der Zeichnungen



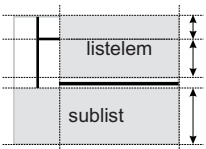

```
insertionLine(20, -6, 30, -6, NULL);
container("listelement", 20, -6, 30, 6, VisAlignLeft, visAlignBuoy(0));
insertionLine(20, 8, 30, 8, "listelement");
container("sublist", 0, 10, 30, 18, VisAlignLeft, VisAlignTop);
```

(b) Hinzunahme der Platzhalter und Einfügemarken

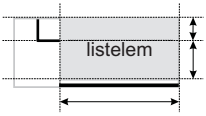



```
stretchX(22, 28);
stretchY(-4, -2);
stretchY(2, 4);
stretchY(14, 16);
```

(c) Hinzunahme der Dehnungsbereiche




```
figure(2, "line", "-width 2"); point(10, 0); point(20, 0);
figure(2, "line", "-width 2"); point(10, -6); point(10, 10);
container("listelement", 20, -6, 30, 6, VisAlignLeft, visAlignBuoy(0));
insertionLine(20, 8, 30, 8, "listelement");
container("sublist", 0, 10, 30, 18, VisAlignLeft, VisAlignTop);
stretchX(22, 28);
stretchY(-4, -2);
stretchY(2, 4);
stretchY(14, 16);
```




```
figure(3, "line", "-width 2"); point(10, -6); point(10, 0); point(20, 0);
point(0,0);
container("listelement", 20, -6, 30, 6, VisAlignLeft, visAlignBuoy(0));
insertionLine(20, 8, 30, 8, "listelement");
stretchX(22, 28);
stretchY(-4, -2);
stretchY(2, 4);
```

(d) Die anderen beiden Zeichnungen für VCore

Abbildung 5.12.: Spezifikation dynamischer Zeichnungen

Zusammen mit den Platzhaltern werden in Abbildung 5.12(b) Einfügemarken spezifiziert, das sind die horizontalen schwarzen Linien. Einfügemarken werden in einer dynamischen Zeichnung nur dann benötigt, wenn die Zeichnung im RecursiveList-Modul angewendet werden soll. Die Einfügemarken dienen dazu, eine Einfügeposition in der graphischen Darstellung mit einem Einfügekontext im Strukturbaum zu assoziieren. Eine ausführlichere Beschreibung enthält Abschnitt 4.5.1. Der Sprachentwerfer muß hier aber lediglich die Koordinaten der Einfügemarken festlegen.

Der dritte Teil der Spezifikation legt das dynamische Verhalten der Zeichnungen fest. Hierzu muß spezifiziert werden, in welcher Weise die Zeichnung verändert werden soll, wenn eine graphische Darstellung in einen Platzhalter eingefügt wird. Das Ziel war es, das dynamische Verhalten möglichst intuitiv und einfach zu spezifizieren, dabei aber Ausdrucksstärke und effiziente Realisierbarkeit der Spezifikationstechnik zu erhalten.

Um das dynamische Verhalten einer Zeichnung festzulegen, wird diese in ihren minimalen Ausmaßen spezifiziert und ihr Expansionsverhalten durch Dehnungsintervalle beschrieben. Dehnungsintervalle geben an, welche Teile der Darstellung linear gestreckt werden können, um dadurch (u.a.) die Platzhalter zu vergrößern. Die Dehnungsintervalle werden durch Pfeile und gestrichelte Linien visualisiert (siehe Abbildung 5.12(c)). Sie dürfen sich auf einer Achse nicht überlappen. Weiterhin muß jeder Container sich in jeder Richtung mindestens einmal mit einem Dehnungsintervall überlappen. Im Bild hat der Container listelement auf der vertikalen Achse zwei Dehnungsintervalle, um das Expansionsverhalten für die Ausrichtung anhand der Grundlinie korrekt zu beschreiben.

Das Expansionsverhalten, das durch Dehnungsintervalle erzielt wird, ist besonders intuitiv, solange die Grenzen der Intervalle nicht von schräg verlaufenden Linien geschnitten werden. Um die Ausdrucksstärke zu vergrößern, können Dehnungsintervalle priorisiert werden und es können Intervalle spezifiziert werden, die sich im gleichen Maße strecken sollen. Damit kann ggf. die Isometrie in bestimmten Bildteilen aufrechterhalten werden.

Um dynamische Zeichnungen zu spezifizieren, wurde zunächst eine Spezifikationsnotation umgesetzt, die in die Wirtssprache C++ eingebettet ist, d.h. die textuellen Spezifikationen in Abbildung 5.12 sind Methodenaufrufe einer C++-Klasse. Das erlaubt es, die Beschreibungsmodelle für dynamische Zeichnungen zu evaluieren, ohne die visuelle Spezifikationssprache, die die graphischen Repräsentationen in Abb 5.12 andeuten, vollständig zu entwickeln und zu implementieren.

In einer visuellen Entwicklungsumgebung für visuelle Sprachen sollte es dagegen unbedingt eine visuelle Spezifikationssprache für dynamische Zeichnungen geben. Dafür sprechen wichtige Argumente: Die Beschreibung der grundlegenden Vektorgraphik-Zeichnung ist am einfachsten graphisch durchführbar. Gleiches gilt für die anderen Bestandteile der Zeichnungen, da diese ebenfalls Koor-

dinatenangaben benötigen. In einer Entwicklungsumgebung können außerdem Konsistenzprüfungen für eine spezifizierte Zeichnung durchgeführt werden. Weiterhin können dynamische Zeichnungen so leichter und schneller entwickelt, getestet und bei Sprachänderungen modifiziert werden. Grade das Experimentieren mit Designalternativen einer Sprache kann dies erforderlich machen. Schließlich können auch einige der Berechnungen, die jetzt erst zur Laufzeit durchgeführt werden, in die Übersetzung der Spezifikation verlagert werden.

5.2. Evaluierung

Schon für die Implementierung textueller Sprachen wurden mit Erfolg Lösungen für häufig auftretende Teilprobleme durch Spezifikationsmodule abstrahiert. In dieser Arbeit will ich zeigen, daß sich diese Technik auch gut eignet, um visuelle Sprachen zu implementieren. Durch diese Vorgehensweise wird die Sprachimplementierung einfach und es wird eine sehr hohe Spezifikationsebene erreicht. Das führt unter anderem dazu, daß sich die Spezifikationen einfach ändern lassen, beispielsweise um Sprachänderungen oder Designexperimente durchzuführen. Schließlich basieren die Spezifikationsmodule auf einer grundlegenden Spezifikationstechnik. Dies führt dazu, daß sich dieser Ansatz für eine große Menge visueller Sprachen anwenden läßt. Im Rahmen der Beispielsprache VCore wurden bereits diese Qualitätseigenschaften exemplarisch für ein Spezifikationsmodul gezeigt.

Um diese Spezifikationstechnik näher zu untersuchen, wurden Spezifikationsmodule für die wichtigsten visuellen Muster erarbeitet und einige Beispielsprachen damit implementiert. Der erste Abschnitt beschreibt kurz die implementierten Spezifikationsmodule und nennt ihre wesentlichen Eigenschaften. Der zweite Abschnitt diskutiert meßbare Eigenschaften dieser Spezifikationstechnik anhand der Implementierung einiger Beispielssprachen.

5.2.1. Überblick über die Spezifikationsmodulbibliothek

Die Implementierung der Spezifikationsmodule wurde im Rahmen der gemeinsamen Diplomarbeit von Schmidt und Schindler [2000] durchgeführt. Die Zielstellung war dabei, einen initialen Satz von Spezifikationsmodulen für die wichtigsten visuellen Muster zu erarbeiten. Dieses Ziel wurde sehr gut erfüllt: die implementierten Spezifikationsmodule ermöglichen es bereits, zahlreiche visuelle Sprachen damit zu implementieren. Ihr Nutzen wurde an so unterschiedlichen Sprachen wie QBE [Zloof 1975], Zustandsdiagrammen [Harel 1988], Streets [Kastens und Jung 1998], Nassi-Shneiderman-Diagrammen [Nassi und Shneiderman 1973], Petri-Netzen [Petri 1962] sowie an einigen Beispielsprachen gezeigt. Die

Spezifikationsmodule lassen sich hervorragend kombinieren. Sie bieten ein breites Spektrum verschiedener Darstellungsmöglichkeiten, so daß sie vielfältig eingesetzt werden können, um eine Sprache zu entwerfen und mit dem Design der Sprachkonstrukte zu experimentieren.

Abbildung 5.13 liefert einen graphischen Überblick über die Spezifikationsmodulbibliothek. Für die implementierten Module ist im Bild jeweils ein charakterisierendes Anwendungsbeispiel enthalten. Diese Abbildung wird im folgenden als Leitfaden dienen, um die Spezifikationsmodule vorzustellen und ihre wichtigsten Eigenschaften zu erläutern.

5.2.1.1. Das SimpleList- und das RecursiveList-Modul

Die Spezifikationsmodule, die in Abb. 5.13 links dargestellt sind, visualisieren Folgen. Beim SimpleList- und RecursiveList-Modul wird die Reihenfolge der Elemente der Folge in eine Ordnung der graphischen Darstellung der Listenelemente umgesetzt, wie es beim Listen-Muster (Seite 81) beschrieben wurde. Bei beiden Modulen wird das Layout der Darstellung durch die Box-And-Glue-Methode direkt durch den Attributauswerter berechnet, beide Module implementieren die Attributschnittstelle für geschachtelte Darstellungen. Wie bereits in Abschnitt 5.1.1 anhand von Beispielen gezeigt, werden beim RecursiveList-Modul dynamische Zeichnungen zur Beschreibung der Liste verwendet, wodurch gegenüber dem SimpleList-Modul andere Ausbreitungsrichtungen und zusätzliche Zeichnungsteile für die graphischen Elemente ermöglicht werden.

Die Anwendung der Module wird durch ihre Symbolrollendiagramme be-

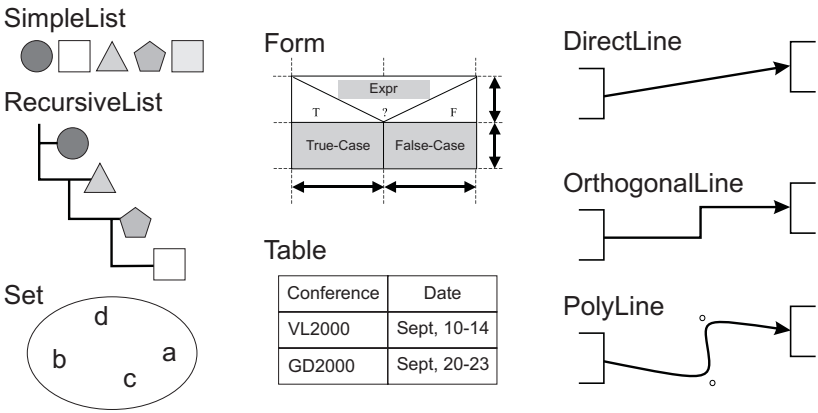


Abbildung 5.13.: Überblick über die Spezifikationsmodule

schrieben. Das Symbolrollendiagramm für das SimpleList-Modul wurde bereits in Abschnitt 5.1.3 vorgestellt, für das RecursiveList-Modul ist es in Abbildung 5.14(a) wiedergegeben. Wie man sieht, gleicht ein großer Teil der Symbolrollen auf der rechten Seite denen, die auch im SimpleList-Modul verwendet werden. Dadurch ist ein Wechsel des Spezifikationsmoduls relativ leicht. Beide Konstellationen des SimpleList-Moduls sind auch bei RecursiveList anwendbar, auch wenn hier lediglich eine Variante dargestellt ist.

Mit Hilfe der links dargestellten Symbolrolle Container können beim RecursiveList-Modul zusätzliche graphische Darstellungen wie beim Form-Modul in die Listen-Darstellung eingebracht werden. Dies ist beispielsweise erforderlich, um den Default-Zweig der Fallunterscheidung bei den Nassi-Shneiderman-Diagrammen zu erzeugen. Die beim Form-Modul diskutierten Variationsmöglichkeiten für Container sind auch hier anwendbar.

5.2.1.2. Das Set-Modul

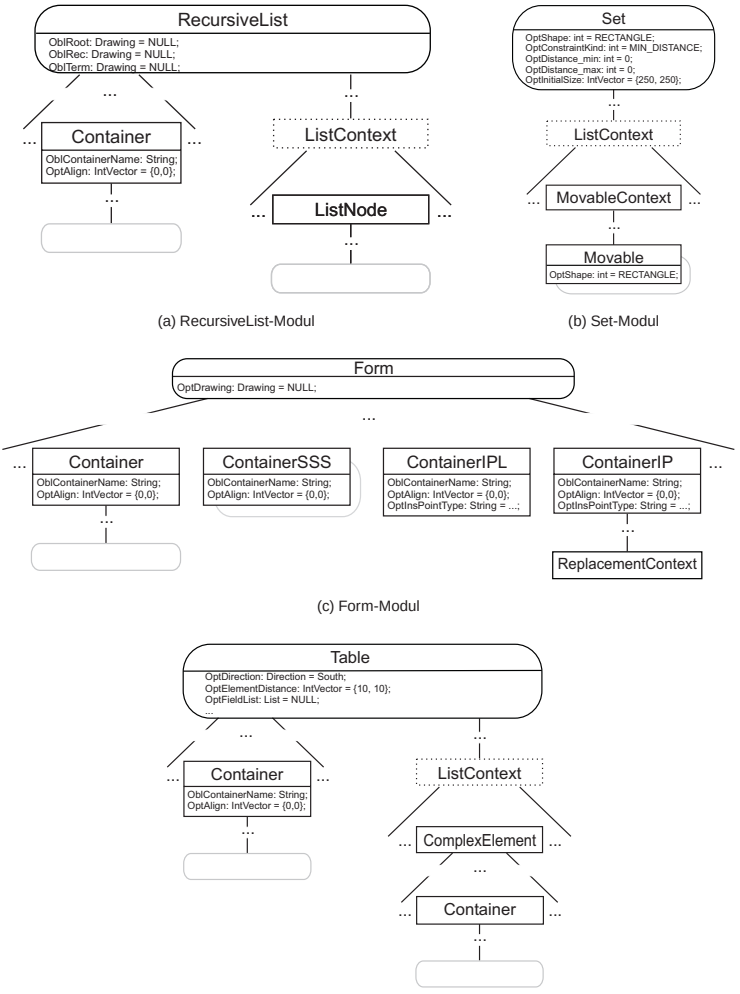
Das Set-Modul implementiert das Mengen-Muster (Seite 83). Die hier zugrundeliegende abstrakte Struktur zwar ebenfalls eine Folge, hier kommt es jedoch ausschließlich auf Enthaltensein und nicht auf die Reihenfolge der Elemente an. Das Modul führt die Layoutberechnungen mit Hilfe eines Constraint-Solvers durch und kann so die korrekte Schachtelung und die Nichtüberlappung der graphischen Darstellungen der Elemente unterstützen. Im weiteren können durch Überschreiben einer Attributberechnung auch Constraints generiert werden, die einen Maximalabstand der Elemente sicherstellen. Damit kann ggf. auch Berührung der graphischen Darstellungen der Elemente erzielt werden.

Um das Modul anzuwenden, sind vier Symbolrollen, wie im Diagramm aus Abbildung 5.14(b) angegeben, an Grammatiksymbole zu vererben. Dabei kann in der Symbolrolle Movable die Form jedes Elementtyps und in der Symbolrolle Set die äußere Form der Menge spezifiziert werden. Diese Informationen werden zur Generierung des Constraint-Netzwerks benötigt.

5.2.1.3. Das Form-Modul

Das Form-Modul setzt das Formular-Muster um. Es visualisiert Tupel, die im einfachsten Fall die rechte Seite einer Grammatikproduktion sind und benutzt dazu eine dynamische Zeichnung. In der Zeichnung sind die Layoutberechnungen gekapselt, die automatisch eine platzoptimierte Darstellung für das Tupel nach der Box-and-Glue-Methode berechnen.

Um die Zeichnung anzuwenden, muß ihr Name angegeben werden und es muß festgelegt werden, in welche Platzhalter der dynamischen Zeichnung die Tupelelemente jeweils eingesetzt werden sollen. Es werden zwei Anwendungssituationen unterstützt, die für unterschiedliche Tupelelemente kombiniert werden



(a) RecursiveList-Modul

(b) Set-Modul

(c) Form-Modul

(d) Table-Modul

Abbildung 5.14.: Symbolrollendiagramme für die Spezifikationsmodule (Teil 1)

können. Beide Situationen sind gemeinsam in Abbildung 5.14(c) skizziert. Sie unterscheiden sich, je nachdem ob es für ein Tupelelement eine oder mehrere Produktionsalternativen gibt.

Gibt es mehrere Alternativen, so werden Editieroperationen benötigt, mit denen ein Benutzer zwischen den Alternativen auswählen kann. Um diese zu erzeugen, kann man die Symbolrolle ContainerIPL verwenden. Diese muß, damit die Editieroperationen korrekt erzeugt werden, dem Grammatiksymbol vererbt werden, für das mehrere Produktionsalternativen existieren. Da das nicht immer praktisch ist, können die Berechnungen alternativ auch in ContainerIP und ReplacementContext aufgeteilt werden.

Wenn es lediglich eine Produktionsalternative gibt, so werden keine Editieroperationen benötigt. Auch hier gibt es wiederum zwei Möglichkeiten: Die Symbolrolle ContainerSSS[6] ist für Mehrfachvererbung konzipiert und ermöglicht es, gegenüber der Symbolrolle Container eine Kettenproduktion einzusparen.

Die Symbolrollen für Container repräsentieren das allgemeinere Konzept "Einsetzen in eine dynamische Zeichnung", das außer im Form-Modul auch noch im RecursiveList und im Table-Modul verwendet wird. Man kann sich die Symbolrollen auch als separates Spezifikationsmodul vorstellen, das in den hier vorgestellten Modulen lediglich benutzt wird.

5.2.1.4. Das Table-Modul

Das Table-Modul setzt das Tabellen-Muster um. Im Rahmen dieser Arbeit wurde es evaluiert, indem die Datenbank-Abfragesprache QBE [Zloof 1975] implementiert wurde. Das Modul visualisiert eine Folge von Tupeln und berücksichtigt gegenüber einer Kombination des SimpleList- mit dem Form-Modul die Gleichartigkeit der Tupel, indem es die Elemente in gleicher Weise als Spalte oder Zeile einer Tabelle ausrichtet.

Auch im Symbolrollendiagramm (siehe Abb. 5.14(d)) läßt sich die Verwandschaft zur Kombination dieser Module erkennen: man muß lediglich die Symbolrolle Table gegen SimpleList und die Rolle ComplexElement gegen Form austauschen. Entsprechend wird durch die Symbolrolle Container auf der rechten Seite ein Tabellenelement, durch ComplexElement eine Tabellenzeile und durch ListContext die ganze Tabelle repräsentiert. Die Symbolrolle Table koordiniert die Berechnungen des Spezifikationsmoduls und generiert zusammen mit der Symbolrolle Container auf der linken Seite die Tabellenüberschriften.

Bei der Anwendung des Moduls muß die Struktur der Tabelle spezifiziert werden, indem die Attributberechnung für Opt_FieldList überschrieben wird. Dabei werden symbolische Namen für die Tabellenspalten vergeben, auf die sich die Container-Rollen beziehen können, um Tabellenköpfe und die Einträge in die Ta-

[6] Dies steht für "Container with Subobject at Same Symbol".

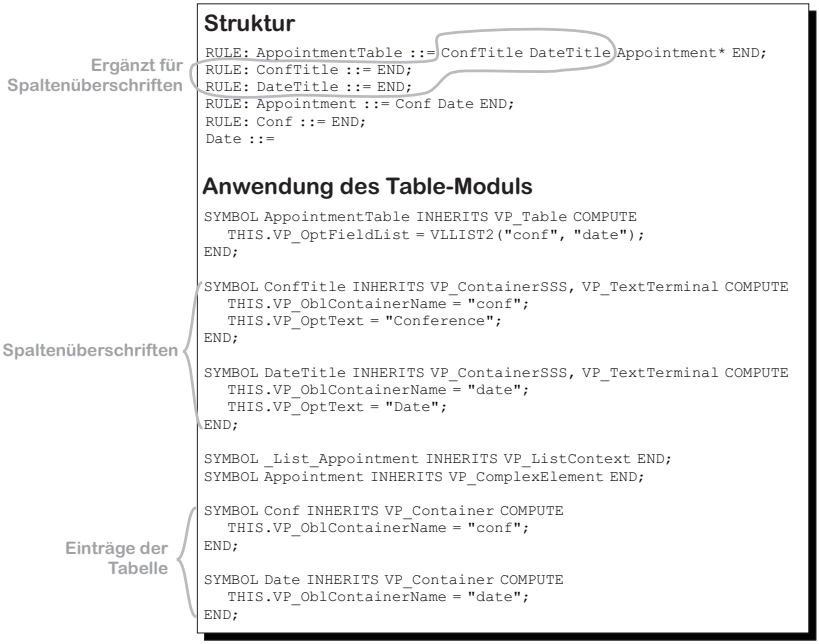


Abbildung 5.15.: Spezifikation für Tabellenbeispiel aus Abb. 5.13

belle festzulegen. Durch diese Vorgehensweise sind für die Spaltenüberschriften die anderen Spezifikationsmodule anwendbar, wodurch das Table-Modul sehr flexibel einsetzbar ist. Allerdings bedeutet dies auch, daß für manche Anwendungen des Table-Moduls die zugrundeliegende Programmrepräsentation geringfügig erweitert werden muß. Das kann insbesondere dann unpraktisch sein, wenn mehrere Sichten verwendet werden. Das zeigt auch die Spezifikation für das Tabellenbeispiel aus Abbildung 5.13, die in Abbildung 5.15 aufgeführt ist.

5.2.1.5. Die Linien-Module

Die drei Linien-Module implementieren das Linien-Muster. Sie stellen zweistellige Beziehungen zwischen den Sprachkonstrukten eines visuellen Programms dar, indem sie Verbindungslinien zwischen den graphischen Darstellungen der in Beziehung stehenden Sprachkonstrukte erzeugen. Sie setzen dazu auf einer Programmrepräsentation auf, die bereits in Abschnitt 4.2 anhand der Sprache der

Petri-Netze und mit unterschiedlichen Schwerpunkten in weiteren Abschnitten diskutiert wurde. Die Linien liegen demnach als Programmkonstrukte vor, die zur Kennzeichnung der Programmkonstrukte, die sie verbinden, deren identifizierende Definitionstabelleneinträge speichern.

Der Vielfalt, mit der Linien in visuellen Sprachen angewendet werden, wird hier durch drei Linien-Module Rechnung getragen, die jeweils unterschiedliche Strategien der Linienführung unterstützen. Die Module haben aber auch weitgehende Gemeinsamkeiten, denen durch gemeinsam genutzte Symbolrollen auch in ihrer Anwendung Rechnung getragen wird. Diese gemeinsam genutzten Symbolrollen liegen in zwei Varianten vor, die unterschiedliche Interaktionsstile umsetzen. Die in Abbildung 5.16(a) oben dargestellten Rollen unterstützen die Erzeugung von Linien in drei Schritten (siehe Abb. 4.20), die unteren, mit dem Suffix "DD" [7] versehenen, verwenden die effizientere, aber weniger flexible Methode der Linienerzeugung mit nur einer Interaktion.

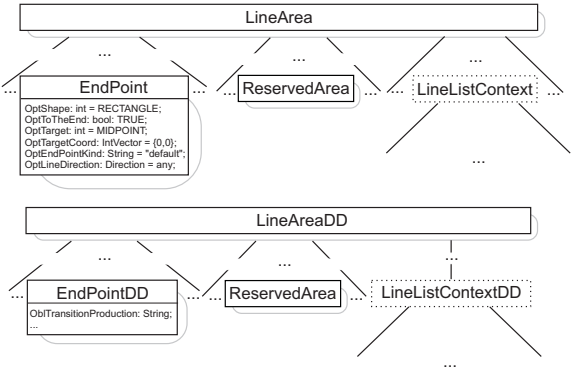
Bei allen drei Modulen werden Informationen über die verbindbaren Programmkonstrukte durch die Symbolrolle Endpoint (EndpointDD) gesammelt. An dieser Stelle können bei der Benutzung der Rolle weitere Angaben gemacht werden, die bestimmen, auf welche Weise Linien an das Programmkonstrukt angeschlossen werden. Die Symbolrolle LineArea (LineAreaDD) kennzeichnet den Teil eines visuellen Programms, in dem die Linien gezeichnet werden können – dies ist in kleineren Sprachen oft das gesamte visuelle Programm. Die Symbolrolle LineListContext (LineListContextDD) muß an den oder die Kontexte im Strukturbaum vererbt werden, in den oder in die neue Linien eingefügt werden können.

Mit Hilfe der Rolle ReservedArea können schließlich bestimmte Teile eines visuellen Programms für die Linienführung reserviert werden. Es besteht die Hoffnung, daß dadurch ein visuelles Programm lesbarer wird; dies muß sich in der Praxis jedoch erst noch erweisen. Reservierte Bereiche werden derzeit aber lediglich beim OrthogonalLine-Modul verwendet, so daß diese Symbolrolle bei den anderen Linienmodulen ignoriert werden kann.

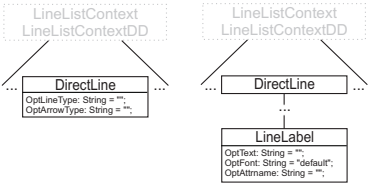
Unterhalb der Symbolrolle LineListContext werden die Symbolrollen angeschlossen, die die Linien-Module implementieren. Diese Linienmodule kapseln jeweils unterschiedliche Techniken der Linienführung. Diese Techniken sind auch gleichzeitig für verschiedene Linientypen verwendbar.

Die Anwendung der Linien-Module wird durch die Symbolrollendiagramme in Abbildung 5.16(b) bis 5.16(d) beschrieben. Die Module unterstützen dabei jeweils zwei Anwendungssituationen, einmal ohne, einmal mit einer textuellen Beschriftung. Die Benutzung des Moduls PolyLine gestaltet sich dabei ein wenig komplexer, da für die manuell verschiebbaren Stützpunkte der Linie derzeit Kontexte im Strukturbaum benötigt werden.

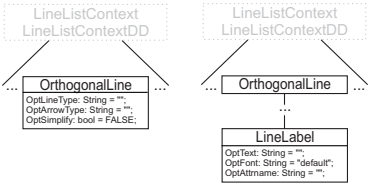
[7] für *direct draw*



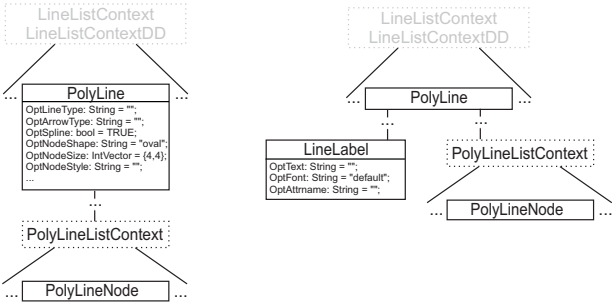
(a) Gemeinsame Symbolrollen der Linien-module



(b) DirectLine-Modul



(c) OrthogonalLine-Modul



(d) PolyLine-Modul

Abbildung 5.16.: Symbolrollendiagramme für die Spezifikationsmodule (Teil 2)

5.2.2. Implementierung visueller Sprachen mit Modulen

Um zu zeigen, welchen Nutzen die Spezifikationsmodule für die Implementierung visueller Sprachen bieten, wurden einige visuelle Sprachen mit Hilfe der Module implementiert. In diesem Abschnitt werden zunächst die implementierten Beispielsprachen vorgestellt und anschließend das Ergebnis einiger Messungen diskutiert. Diese Messungen sollen helfen, den Spezifikationsaufwand und die Qualität der Sprachimplementierung objektiv zu beurteilen.

5.2.2.1. Anwendungsbeispiele

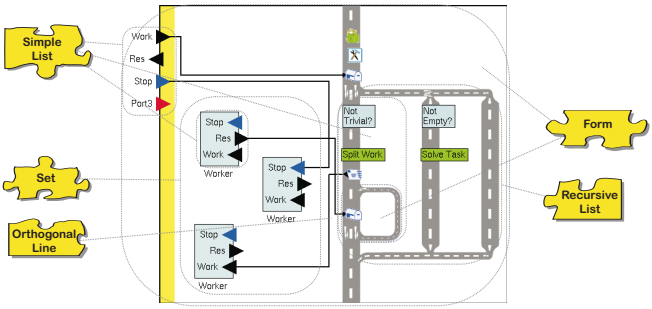
Die Spezifikationsmodule implementieren die wichtigsten visuellen Muster und bilden einen initialen Satz von Beschreibungsmitteln für visuelle Sprachen. Um den Nutzen und die die Praxisrelevanz der Spezifikationsmodule zu zeigen, wurden einige visuelle Sprachen mit Hilfe der Module implementiert. Abbildung 5.17 zeigt ein Bild mit Beispielprogrammen, die mit den generierten Editoren erzeugt wurden. Die gestrichelten Linien und die Puzzleteile gehören nicht zu den Sprachen sondern veranschaulichen jeweils die Anwendung der Spezifikationsmodule.

Die Implementierung der Sprache Streets ist die umfangreichste. Der Editor enthält außer der hier dargestellten Sicht für Prozeduren, einige weitere Sichten, z.B. einen Editor für Typen. Diese Sprachimplementierung ist bemerkenswert, weil hier auf natürliche Weise die Layoutberechnung des Attributauswerters mit der des Constraint-Solvers kombiniert worden ist. Bei der Sprachimplementierung von Hand werden in der Regel die bereits früher erfolgreich angewendeten Methoden benutzt und so das Layout einheitlich durch eine Methode berechnet, das zeigt u.a. auch die manuelle Streets-Implementierung [Kastens und Jung 1998].

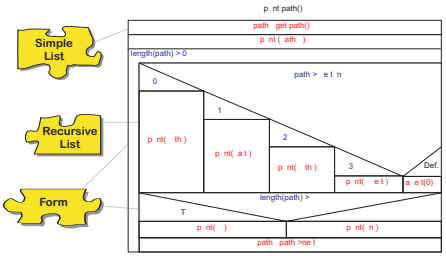
Im Beispielprogramm für Streets sind drei Worker-Prozesse und der Kontrollfluß eines Master-Prozesses abgebildet. Der Master-Prozeß teilt ein Problem auf und übergibt es den Workern zur Lösung. Der Strukturbaum für das Beispielprogramm hat 143 Knoten.

Die Implementierung der Nassi-Shneiderman-Diagramme ist schon an verschiedenen Stellen in dieser Arbeit diskutiert worden. Interessant ist die Anwendung des RecursiveList-Moduls für das Case-Konstrukt wegen der schräg schräg verlaufenden Trennlinie. Sie läßt sich erzeugen, indem bei den dynamischen Zeichnungen die Isometrie der Darstellung aufrechterhalten wird: die Trennlinie besteht aus mehreren Liniensegmenten, deren Steigung gleich ist. Das Nassi-Shneiderman-Beispiel zeigt eine Prozedur, die über eine Datenstruktur iteriert und Informationen ausgibt. Der Strukturbaum besteht aus lediglich 74 Knoten, die aber schon eine beachtliche Menge an Informationen enthalten.

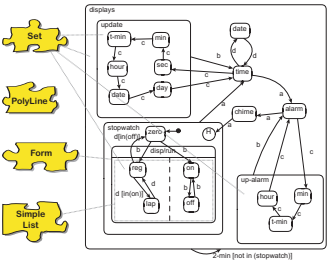
Das Zustandsdiagramm-Beispiel ist ein Teil der Armbanduhr-Spezifikation, die Harel [1988] bei seiner Sprachbeschreibung benutzt hat. Es ist mit 155 Kno-



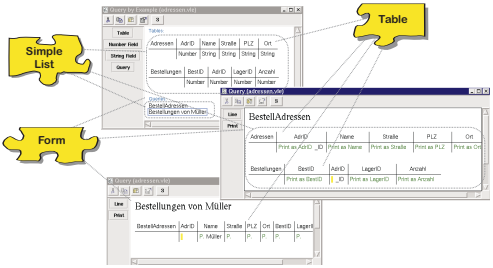
(a) Streets



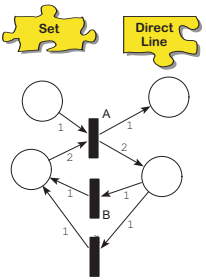
(b) Nassi-Shneiderman-Diagramme



(c) Zustandsdiagramme



(d) Query by Example



(e) Petri-Netze

Abbildung 5.17.: Anwendung der Spezifikationsmodule in einigen visuellen Sprachen

ten, die zur Darstellung eines einzigen Fensters beitragen, das umfangreichste hier behandelte Beispiel. Die Spezifikation der Zustandsdiagramme verwendet das Set-Modul für die hierarchischen Mengen und das SimpleList-Modul für die AND-Superstates. Die Linien wurden mit dem Polyline-Modul erzeugt. Beim Set-Modul wurde die Prüfung auf Überlappungsfreiheit durch den Constraint-Solver aktiviert; für das Beispielprogramm ergibt sich dabei ein Constraint-Netzwerk, das aus 288 Formeln besteht.

Das Beispielprogramm für Query-By-Example wurde bereits ausführlich in Abschnitt 4.6.1.2 (Seite 154) behandelt und ist hier nochmals aufgeführt, um auch ein Beispiel für die Benutzung des Table-Moduls zu präsentieren. Im Gegensatz zu den anderen besteht dieses Beispiel aus mehreren Fenstern. Das Petri-Netz-Beispiel spezifiziert schließlich eine bestimmte Reihenfolge, in der die Transitionen A, B und C feuern dürfen. Die Spezifikation benutzt das DirectLine-Modul für die Verbindungen und das Set-Modul für die Knoten. Beim Set-Modul wurde die Prüfung auf Überlappungsfreiheit ausgeschaltet; das sich sonst ergebende Constraint-Netzwerk ist für eine realistische Menge an Places und Transitions nicht mehr effizient lösbar. Der Strukturbaum besteht hier aus 41 Knoten.

Einige weitere implementierte Beispielsprachen sind nicht in Abbildung 5.17 aufgeführt. Das sind z.B. strukturierte Flußdiagramme, deren Spezifikation sich nur geringfügig von der der Nassi-Shneiderman-Diagramme unterscheidet, sowie einige kleine Beispielsprachen, die zu Demonstrations- und Testzwecken erstellt wurden.

Die Implementierung dieser Beispielsprachen zeigt, daß sich die Spezifikationsmodule gut kombinieren lassen und daß sich eine große Vielfalt an graphischen Darstellungen mit ihnen realisieren läßt. Sämtliche Beispielsprachen wurden fast ausschließlich unter Benutzung der in Abschnitt 5.2.1 beschriebenen Spezifikationsmodule implementiert. Hinzu kommt bei Streets und QBE ein weiteres Spezifikationsmodul, das Konsistenzprüfungen realisiert, siehe Abschnitt 4.6.1.2 (Seite 154). Beim Petri-Netz-Beispiel ist die Prüfung der Korrektheit der Verbindungen dagegen ohne die Benutzung von Modulen wie in Abschnitt 4.6.1.1 beschrieben implementiert.

5.2.2.2. Untersuchung der Spezifikation

Um den Aufwand für die Sprachimplementierung zu messen, ist es üblich, die Länge der Spezifikation zu bestimmen. Bevor ich auf Messungen für die Beispielsprachen eingehe, soll anhand eines Beispiels auf die Aussagekraft der Metrik für diese Spezifikationstechnik eingegangen werden.

Wichtig für den Erstellungsaufwand einer visuellen Sprache ist nicht nur die Länge der Spezifikation, sondern auch die Komplexität der Spezifikation verglichen mit anderen Spezifikationstechniken. Das zeigt gut ein Beispiel für eine visuelle Sprache, deren Identität hier keine Rolle spielen soll. Für die betrachtete

```
SYMBOL _List_Schrank
COMPUTE
  THIS.x = 0;
  THIS.y = 0;

  CHAINSTART HEAD.xpos = THIS.x;
  THIS.width = MAX(SUB(TAIL.xpos, THIS.x), 10);
  THIS.height = MAX(CONSTITUENTS Schrank.height
                WITH (int, MAX, IDENTICAL, ZERO), 10);

  VLFixInspointCreate(_currn, "redline",
    VLREGION(THIS.x, THIS.y, THIS.x, ADD(THIS.y, THIS.height)),
    INSPOINT_SON);
END;
```

(a) manuelle Attributberechnung

```
SYMBOL Schrankwand INHERITS VP_SimpleList
COMPUTE
  THIS.VP_OptDirection=visEast;
  THIS.VP_OptInsertionPointType="redline";
  THIS.VP_OptMinWidth2=222;
  THIS.VP_OptElementDistance=30;
  THIS.VP_OptAlignInWidth=VisAlignBottom;
END;
```

(b) Anwendung eines Spezifikationsmoduls

Abbildung 5.18.: Vergleich der Komplexität zweier Spezifikationstechniken

Sprache wurden zwei Spezifikationen mit unterschiedlichen Spezifikationstechniken erstellt. Beide Implementierungsalternativen verwenden das VLEli-System. Einmal wurden dabei die Spezifikationsmodule verwendet, ein anderes Mal direkt die Spezifikationstechniken des zugrundeliegenden VLEli-Systems benutzt. Die Längen der Spezifikationen liegen in etwa im Verhältnis 1:2.

Ist die Benutzung der Spezifikationsmodule lediglich halb so aufwendig wie die direkte Benutzung von VLEli? Ein Auszug aus den Spezifikationen offenbart anderes, siehe Abbildung 5.18 (zitiert aus [Schmidt und Schindler 2000]). Beide Ausschnitte leisten in etwa dieselbe Aufgabe. Während in Abb. 5.18(a) die Layoutberechnungen für eine einfache Liste direkt durch Attributberechnungen spezifiziert werden, sind sie in Abb. 5.18(b) in ein Spezifikationsmodul gekapselt, das hier lediglich benutzt wird.

In der oberen Spezifikation müssen zahlreiche Abhängigkeiten zwischen den einzelnen Angaben berücksichtigt werden. Dies erhöht den zur Erstellung der Spezifikation benötigten Aufwand. Die Spezifikation wird insgesamt gesehen je-

doch relativ kurz, weil Konstrukte wie CONSTITUENTS sehr viele Attributberechnungen in vielen Kontexten spezifizieren.

In der unteren Spezifikation sind die einzelnen Angaben dagegen weitgehend orthogonal zueinander und können sogar weggelassen werden. Dadurch wird die Spezifikation leichter verständlich und einfacher wartbar. Es ist sogar möglich, die Spezifikation schrittweise zu erstellen und mit der visuellen Darstellung zu experimentieren, bis sich das gewünschte Aussehen und Verhalten einstellt. Die Länge der Spezifikation ist dabei relativ hoch, weil für jedes überschriebene Layoutattribut eine Zeile benötigt wird.

In Ermangelung besserer Alternativen soll im folgenden trotzdem die Länge als Maß für den Spezifikationsaufwand dienen. Sie wird hier durch die Anzahl der Zeilen gemessen, die nicht leer sind oder ausschließlich Kommentare enthalten. In Tabelle 5.1 ist die Gesamtlänge der Spezifikationen der implementierten visuellen Sprachen aufgeschlüsselt nach Spezifikationsteilen angegeben. Die erste Spalte (Repr) mißt die Länge der Spezifikation der Programmrepräsentation. Sie besteht aus der kontextfreien Grammatik für die Sprache und Definitionen der persistenten Attribute.

Die zweite Spalte zählt die Anzahl der Zeilen, die für die Anwendung der Spezifikationsmodule benötigt werden. Sie ist mit “Graphische Darstellung” überschrieben, weil die Spezifikationsmodule i.W. die graphische Darstellung und Informationen zur Durchführung der Editieroperationen berechnen. In der dritten Spalte ist die Länge der textuellen Spezifikation der dynamischen Zeichnungen angegeben. Die vierte Spalte enthält schließlich die Summe der sonst noch erforderlichen Spezifikationen, die unter anderem auch die Definition der Fenster des generierten Editors enthält. Hier wurden allerdings keine Berechnungen erfaßt, die für Analyse- und Codegenerierung benötigt werden.

In der letzten Spalte ist schließlich die Länge der aus diesen Spezifikationen generierten Quelltextdateien in C angegeben. Die Zahlen messen ausschließlich die Länge der generierten Module. Die vollständige Implementierung besteht zusätzlich aus ca. 18 000 Zeilen festem C-Code und aus ca. 8 800 Zeilen Tcl-Code.

Aus der Tabelle können einige wichtige Informationen über die Spezifikationtechnik abgeleitet werden. Zunächst läßt sich ein Indikator für den zur Implementierung einer Sprache benötigten Aufwand bestimmen, indem die Gesamtlänge der Spezifikationen in Relation zu Spalte 2 (Repr) gesetzt wird. Das Verhältnis ist über die Spezifikationen relativ konstant und liegt zwischen 4,6 und 7,1[8] Für einen Sprachentwerfer gibt dies den erwarteten Aufwand für die Implementierung einer visuellen Sprache an, weil die Repräsentation typischerweise als erste fertiggestellt wird.

[8] Das Beispiel VCore wurde hier nicht berücksichtigt, da dessen Spezifikation überall die Vorgaben für die Layoutparameter der Spezifikationsmodule benutzt und deshalb untypisch kurz ausfällt.

Sprache	Spezifikation					Gener. Code
	Repr.	Graph. Darst.	Dyn. Zeichn.	Weitere Spez.	Gesamt	
Streets	252	529	340	371[a,b]	1492	67 539
NSD.	29	67	100	10	206	11 372
strukt. Flußdiag.	22	49	65	8	144	8 878
Zustandsdiagramme	45	152	-	46	243	14 411
Query-By-Example	55	133	34	117[b]	339	18 758
Petri-Netze	25	48	-	43[c]	116	8 703
VCore	6	7	-	5	18	5 553

[a] Hinzu kommen einige Pixelgraphik-Bilder
[b] Konsistenzerhaltung zwischen Definition und Anwendung (Spezifikationsmodul)
[c] Prüfung der Linien (von Hand geschriebene Attributberechnungen)

Tabelle 5.1.: Länge der Spezifikationen der Beispielsprachen

Darüberhinaus ist die Spezifikation der Repräsentation vom Aufwand her vergleichbar mit dem Aufwand für die Erstellung der kontextfreien Grammatik einer entsprechenden textuellen Sprache. In diesem Licht besehen, mißt das Verhältnis 4,6 – 7,1 den Aufwand, der für die Implementierung eines Editors für eine visuelle Sprache mehr investiert werden muß, als für eine entsprechende Sprache mit textueller Notation. Dieses Verhältnis gilt auch für umfangreichere Sprachen, wie die Implementierung der Sprache Streets signalisiert. Das ist ein gutes Ergebnis dieser Arbeit, weil die Spezifikation der kontextfreien Grammatik einer textuellen Sprache verglichen mit dem Spezifikationsaufwand für Analyse und Weiterverarbeitung eher gering ist.

Ein weiteres wichtiges Verhältnis kennzeichnet die hohe Ebene, die für die Spezifikation visueller Sprachen erreicht wird. Vergleicht man die letzten beiden Spalten miteinander, so stellt man fest, daß der generierte Code zwischen 45 (für umfangreiche Spezifikationen) bis zu über 300 Mal so lang ist, wie die Summe der Spezifikationen. Dieses Indiz für die Höhe der Spezifikationsebene kann durch Vergleich mit der handgeschriebenen Implementierung von Streets erhärtet werden, für die ca. 12 000 Zeilen Tcl-Code erstellt wurden. Beide Sprachvarianten decken zwar nicht denselben Sprachumfang ab, sind jedoch miteinander vergleichbar.

Im weiteren kann der Spezifikationsaufwand anhand der Sprache der Nassi-Shneiderman-Diagramme mit dem VPE-System [Grant 1998] verglichen werden. Für den implementierten Teil der Sprache wird beim VPE-System mit 154 Zeilen nur knapp 3/4 der hier benötigten Spezifikation gebraucht. Der Größenunterschied erklärt sich dadurch, daß die Spezifikationsmodule auf einem universellen Werkzeugsystem aufbauen, wodurch eine größere Flexibilität erreicht wird. Wei-

terhin ist, wie oben angegeben, die Spezifikationsnotation für die Anwendung der Spezifikationsmodule zwar "wortreich", aber vergleichsweise einfach. Schließlich ist beim VLEli-System ein gewisser Mehraufwand erforderlich, weil die abstrakte Struktur separat von der graphischen Darstellung spezifiziert wird. Dadurch werden aber auch wichtige Vorteile erzielt, beispielsweise wenn mehrere Sichten implementiert werden müssen.

5.2.2.3. Effizienz der generierten Sprachimplementierung

Ein wesentlicher Faktor für die Benutzbarkeit der generierten Editoren ist die Zeit, die vergeht, bis nach einer Benutzerinteraktion die graphischen Darstellungen der offenen Fenster zu aktualisiert sind [Burnett u.a. 1995]. Verzögerungen von einer Sekunde kann der Benutzer dabei bereits als störend empfinden. Sind diese Zeiten größer, so wird der Gedankenfluß eines Benutzers in hohem Maße behindert, die Benutzbarkeit eines solchen Systems nimmt ab.

In Abhängigkeit von der Art der Interaktion kommt es bei den hier generierten Editoren zu unterschiedlichen Verzögerungen. Führt ein Benutzer Editieroperationen durch, die den sichtbaren Ausschnitt des Programms verändern, z.B. indem ein verdecktes Fenster sichtbar gemacht oder ein Rollbalken benutzt wird, so wird Darstellung vom Canvas-Widget des Tcl/Tk-Systems aktualisiert. Das Widget speichert dazu in einer internen Repräsentation die graphische Darstellung und kann daraus die Fenster aktualisieren. Diese Operation ist recht effizient und soll hier nicht weiter berücksichtigt werden.

Eine andere Art der Benutzerinteraktion sind Änderungen des visuellen Programms. Um die graphische Darstellung zu aktualisieren, wird zunächst ein Attributauswerter (im folgenden Check-Auswerter genannt) für das visuelle Programm ausgeführt. Seine Aufgabe ist der Transport von Informationen zwischen verschiedenen Fenstern und die Durchführung von Strukturprüfungen, siehe Abschnitt 4.3.2.1 und 4.6.1 (Seite 117 bzw. 151). Im Anschluß daran wird für jedes offene Fenster ein weiterer Attributauswerter ausgeführt, der die graphische Darstellung für jeweils ein Fenster neu erzeugt.

Um die Effizienz der generierten Editoren zu messen, wurden beide Verarbeitungsschritte separat berücksichtigt. Für die Messungen der Verarbeitungszeiten wurde ein mit 450 MHz getaktetes Intel Pentium III-System eingesetzt. Das benutzte Betriebssystem ist RedHat-Linux Version 6.1. Die Zeiten wurden mit der Tcl-Funktion `time` gemessen, die die Betriebssystemfunktion `timeofday` benutzt. Es wurde jeweils der kleinste Wert bei 5-maliger Wiederholung der Messungen verwendet, um Einflüsse anderer Prozesse möglichst auszuschließen.

Um die Verarbeitungszeit des Check-Auswerters zu bestimmen, wurde der generierte Streets-Editor benutzt. Von den implementierten visuellen Sprachen wird beim Streets-Editor der (in Anzahl Zeilen Spezifikation gemessen) umfangreichste Gebrauch von diesem Verarbeitungsschritt gemacht. Die dabei u.a. zu lösende

Aufgabe verdeutlicht Abbildung 5.19. Fügt ein Benutzer einen neuen Port in die im linken Fenster dargestellte Prozedur ein, so fügt der Check-Auswerter automatisch entsprechende Ports an den Anwendungsstellen ein und hält so die Anwendungsstellen mit der Definition konsistent.

Als Testprogramm wurde ein Streets-Programm mit 10 Definitionen und 50 Anwendungen dieser Definitionen erstellt. Der Strukturbaum für das Testprogramm besteht aus 1051 Knoten. Es wurden zwei Messungen durchgeführt. Bei der ersten Messung muß der Check-Auswerter Korrekturen an 10 Programmstellen vornehmen und benötigt dafür insgesamt 0,053 Sekunden. Bei der zweiten Messung führt der Check-Auswerter lediglich Prüfungen durch und muß keine Korrekturen ausführen. Hierbei werden lediglich 0,005 Sekunden benötigt.

Dieses Ergebnis zeigt, daß der Durchlauf durch den Strukturbaum sehr schnell ist und die Verarbeitungsdauer für einen Attributauswerter primär von den Operationen abhängt, die vom Attributauswerter angestoßen werden.

Weitere Messungen wurden durchgeführt, um den Zeitbedarf für die Aktualisierung eines Fensters zu bestimmen. Eine objektive Betrachtung wird hier dadurch erschwert, daß für die Berechnung der graphischen Darstellung bei den verschiedenen Sprachen unterschiedliche Techniken benutzt werden. Weiterhin ist der Zeitbedarf für die Aktualisierung abhängig von der Größe des verarbeiteten visuellen Programms und der Art der Programmänderung.

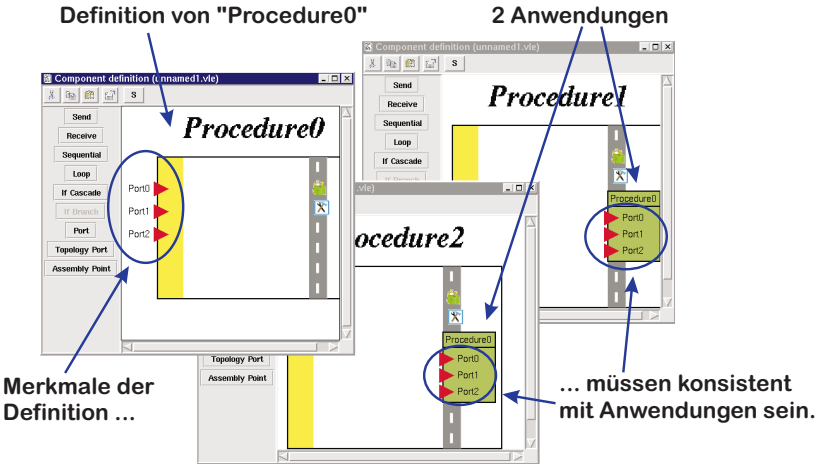


Abbildung 5.19.: Konsistenzerhaltung zwischen Definitionen und Anwendungen in Streets

Sprache	Beispielprg.	Attr.	Solver	Constr.	Canvas	Summe
Streets	Abb. 5.17(a)[a]	0,144s	0,009s	0,011s	0,219s	0,384s
	Abb. 5.17(a)[b]	0,157s	0,013s	0,011s	0,227s	0,409s
NSD	Abb. 5.17(b)	0,039s	-	-	0,082s	0,121s
	50 Schleifen[c]	0,136s	-	-	0,331s	0,466s
Zustandsdiagr.	Abb. 5.17(c)[a]	0,230s	0,058s	0,106s	0,260s	0,654s
	Abb. 5.17(c)[d]	0,139s	2,625s[e]	0,054s	0,145s	2,963s
QBE	Abb. 5.17(d)[f]	0,033s	-	-	0,075s	0,108s
Petri-Netze[g]	Abb. 5.17(e)	0,044s	0,023s	0,026s	0,051s	0,144s

[a] Die Programmänderungen haben keine Constraints verletzt
[b] Die Programmänderungen haben Constraints verletzt
[c] 50-fach geschachtelte Schleife. Der Strukturbaum für dieses Beispiel hat 355 Knoten.
[d] Das Beispiel wurde verkleinert, damit das Constraint-Netzwerk effizient lösbar wurde. Das Netzwerk für dieses Beispiel hatte 148 Formeln.
[e] Dieser Wert schwankte in hohem Maße, da das Constraint-Netzwerk nicht immer in derselben Art und Weise verändert werden konnte. Es wurde ein repräsentativer Wert gewählt.
[f] Mittleres Fenster
[g] Hier war die Prüfung auf Überlappungsfreiheit deaktiviert

Tabelle 5.2.: Dauer der Fensteraktualisierung für Programmänderungen in verschiedenen Sprachen

Um Zeiten für möglichst viele Situationen zu erhalten, wurden für die Messung verschiedene visuelle Sprachen und verschieden große Beispielprogramme zugrundegelegt. Bei Sprachimplementierungen, die den Constraint-Solver verwenden, wurden ferner Zeiten für verschiedenartige Änderungen gemessen, einmal solche, die das Constraint-Netzwerk nicht verletzen und einmal solche, bei denen der Solver neue Werte für Variable des Constraint-Netzwerks berechnen muß. Weitere Details zu Sprachen und Beispielprogrammen sind in den ersten beiden Spalten und den Fußnoten in Tabelle 5.2 zu entnehmen. Die referenzierte Abbildung befindet sich auf Seite 196.

In der dritten bis sechten Spalte wurde die Dauer einzelner Verarbeitungsschritte der Aktualisierung gemessen. Die Spalte “Attr” gibt die Zeit an, die der generierte Attributauswerter jeweils benötigt hat. Da der eigentliche Baumdurchlauf dabei relativ billig ist (s.o.), wird die hier angegebene Zeit i.W. benötigt, um das Layout der Darstellung zu berechnen und ggf. das Constraint-Netzwerk zu erzeugen. Bei den Zustandsdiagrammen wird hier etwas mehr Zeit benötigt, was der Erzeugung des Constraint-Netzwerks zugerechnet werden kann: Die Anzahl der benötigten Formeln ist quadratisch in der Anzahl der Zustände.

Die Spalte “Solver” mißt die Laufzeit des Constraint-Solvers, wobei hier auch die Zeiten zählen, während denen die Werte der Constraintvariable und die geän-

derten Formeln an den Solver übermittelt werden. Bei den Zustandsdiagrammen ergeben sich stark schwankende Zeiten, wenn eine Programmänderung zu einem ungültigen Constraint-Netzwerk führt. Das reflektiert die Tatsache, daß Disjunktionen, wenn sie in größerer Zahl benötigt werden, zu einer Explosion des Lösungsraums führen, den offenbar auch Parcon durch "spezielle Optimierungen" nicht immer effizient durchsuchen kann.[9]

Die Spalte "Constraints" gibt die Zeit an, die benötigt wird, um die Änderung eines Constraint-Netzwerks zu ermitteln, siehe dazu Abschnitt 4.4.2.4 (Seite 137). Diese Zahlen sind relativ klein, vor allem wenn man bedenkt, daß die entsprechenden Programmteile als Tcl-Scripte implementiert sind und lediglich interpretiert werden. Die Spalte "Canvas" gibt schließlich die Zeit an, die für die Aktualisierung der internen Repräsentation des Canvas-Widget benötigt wird. Auch diese Programmteile werden derzeit interpretiert. Sie liefern zwar den größten Anteil am Gesamt-Zeitbedarf für die Aktualisierung der Darstellung, sind aber immer noch erfreulich klein.

Die letzte Spalte der Tabelle gibt die Zeit an, die bei den Beispielsprachen und -programmen vergeht, bis die graphische Darstellung eines Fensters aktualisiert ist und der Benutzer weiterarbeiten kann. Bis auf die Ausnahme, die durch die Explosion des Lösungsraums des Constraint-Solvers begründet ist, sind die Zeiten klein genug, um effizientes Editieren eines visuellen Programms zu ermöglichen. Dies gilt insbesondere auch für größere visuelle Programme, wenn die Informationen wie bei den Streets- und QBE-Programmen in vielen, von der Größe her überschaubaren Fenstern enthalten sind.

5.2.2.4. Benutzbarkeit der generierten Sprachimplementierung

Es ist bekannt, daß die Akzeptanz für eine (neue) visuelle Sprache in hohem Maße von der Qualität und den Eigenschaften der sie implementierenden Entwicklungsumgebung abhängen [Citrin 1993]. Trotzdem sind bislang nur wenige Versuche dokumentiert, um die Benutzbarkeit der generierten Sprachimplementierung bei Generatoreinsatz zu untersuchen. Eine Ursache dafür ist, daß die Benutzbarkeit einer Entwicklungsumgebung in hohem Maße von Eigenschaften der implementierten visuellen Sprache abhängt und deshalb nur schwer beurteilt werden kann.

Ein Beispiel dafür liefern Green und Petre [1996]: von den dort eingeführten Kriterien lassen sich höchstens die "Viskosität", die "Sichtbarkeit" und gewisse Aspekte der "Sekundären Notation" auf Eigenschaften der Sprachimplementierung übertragen. Die Viskosität mißt den Aufwand, der für die Durchführung einer einzelnen Programmänderung aufgewendet werden muß. Ist sie niedrig, so ist die Arbeit mit der Sprachimplementierung flüssig und es sind nur wenige

[9] Damit die hier benötigten Zeiten nicht unerträglich lang werden, kann die Verarbeitung des Constraint-Solvers nach einer einstellbaren Zeit abgebrochen werden.

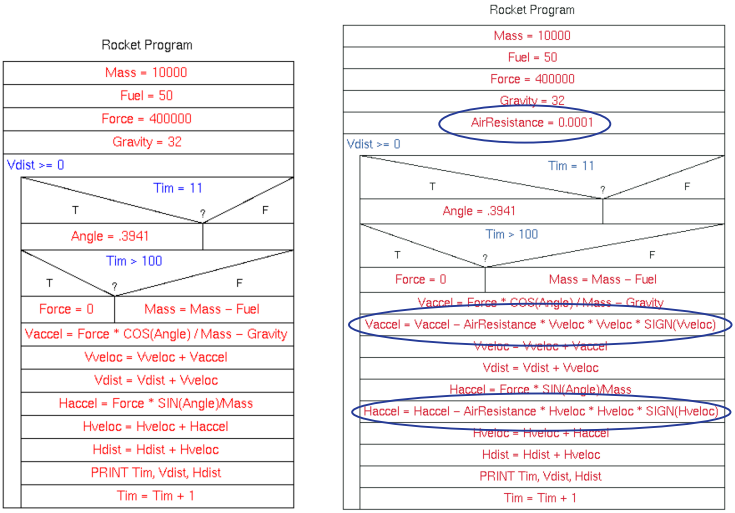
Nachkorrekturen notwendig, beispielsweise um das Layout zu verbessern. Die Frage nach der Sichtbarkeit ist die Frage, ob sich zwei beliebige Teile eines visuellen Programms auch gleichzeitig nebeneinander darstellen lassen. Das ist für das Verständnis eines Programms von Bedeutung. Möglichkeiten zur Sekundären Notation geben schließlich einem Betrachter eines visuellen Programms eine über die "eigentliche" Semantik eines visuellen Programms hinausgehende Lesehilfe. Das ist insofern eine Eigenschaft der Entwicklungsumgebung, als die dafür notwendigen Ausdrucksmittel bereitgestellt werden müssen, z.B. Farbgebung, Kommentare oder Freiheiten beim Layout.

Green und Petre geben lediglich für die Viskosität ein Meßverfahren an. Die anderen Eigenschaften werden lediglich diskutiert. Das Kriterium der "Sichtbarkeit" ist bei den hier erzeugten Entwicklungsumgebungen durch den Einsatz der Fenster erfüllt. Weiterhin gibt es in VLEli zahlreiche Möglichkeiten für sekundäre Notationen – fast alle Einstellungsmöglichkeiten der Spezifikationsmodule lassen sich auch auf diese Weise nutzen. Ein besondere Eigenschaft ist der Einsatz des Constraint-Solvers, der Freiheiten beim Layout gestattet und trotzdem die Viskosität niedrig hält (zumindest solange er noch effizient einsetzbar ist).

Um die relative Viskosität bei drei Entwicklungsumgebungen für unterschiedliche Sprachen zu bestimmen, wurden jeweils Programme implementiert, die dieselbe Aufgabe lösen – es handelte sich um die Berechnung der Flugbahn einer Rakete. Erfahrene Testpersonen sollten eine Programmänderung realisieren und erhielten dazu als Vorlage einen Ausdruck der angestrebten Änderungen. Der dazu erforderliche Zeitbedarf wurde als Maß für die Viskosität verwendet.

Da Green und Petre sowohl das Testprogramm, als auch die gemessenen Zeiten aufgeführt haben, kann der Test hier für die Sprache der Nassi-Shneiderman-Diagramme nachgestellt werden. Das Testprogramm zeigt Abbildung 5.20. Zur Programmänderung mußten die rechts mit Kreisen markierten Anweisungen hinzugefügt werden. Die für den generierten Nassi-Shneiderman-Editor gemessene Zeit ist zusammen mit den von Green und Petre gemessenen Zeiten für Basic, LabView und Prograph in Tabelle 5.3 aufgeführt. Sie liegt für unser System zwischen den für Basic und für Prograph gemessenen Zeiten, was für eine generierte Sprachimplementierungen sehr gut ist. Dafür sind ursächlich zwei Faktoren von Bedeutung: die automatisch durchgeführten Layout-Änderungen bewirken, daß für die erforderliche Größenanpassung der Darstellung keinerlei Interaktionen benötigt werden. Die Möglichkeit, Texte durch Anklicken direkt zu editieren führt im weiteren gegenüber dem Öffnen von Dialogfenstern zu einer Beschleunigung.

Eine etwas andersartige Evaluation der Benutzbarkeit wird von Grant [1998] für das VPE-System durchgeführt. Grant betrachtet eine Sprache, die in einer visuellen und einer textuellen Notation existiert, hier ML bzw. VML [Cardelli 1983]. Um die Benutzbarkeit des VPE-Systems zu bestimmen, läßt er kleine Programmfragmente in beiden Sprachen konstruieren und mißt jeweils die im Durchschnitt



(a) Vor der Änderung

(b) Nach der Änderung

Abbildung 5.20.: Testprogramm zur Messung der Viskosität (nach [Green und Petre 1996])

benötigte Zeit. Beim VPE-System benötigten die Testpersonen im Durchschnitt die 1,4 bis 2,7-fache Zeit für die Konstruktion der visuellen, verglichen mit der Dauer zur Konstruktion der textuellen Programme.

Auch dieser Test wurde für das VLEli-System nachgestellt. Hierbei wurde wiederum die Sprache der Nassi-Shneiderman-Diagramme verwendet. Als textuelle Sprache wurde C++ benutzt, wobei die Sprachkonstrukte jedoch auf diejenigen beschränkt wurden, die auch im NSD-Editor zur Verfügung standen. Für den Test wurden zwei etwas umfangreichere Programme verwendet, nämlich eine Quick-Sort- und eine Selection-Sort-Implementierung, die [Sedgewick 1992] entnommen wurden. Diese Testprogramme enthalten verglichen mit denen von Grant einige Folgen und zahlreiche textuelle Terminale und sind somit als realistischere Tests einzustufen.

Zur Durchführung der Messungen erhielt die Testperson jeweils einen Ausdruck des zu implementierenden Programms in seiner fertigen Form. Der Zeitbedarf zur Konstruktion des textuellen bzw. des visuellen Programms wurde gemessen. Die Ergebnisse sind in Tabelle 5.4 aufgeführt. Die Zeiten zur Erstellung

des visuellen Programms sind naturgemäß etwas länger, weil zum Einfügen der Programmkonstrukte zwischen Tastatur und Maus gewechselt werden muß. Daß sie hier so niedrig liegen, belegt wiederum die Vorteile des automatischen Layouts und die des direkten Editierens von Texten.

Leider ist keine der beiden Techniken anwendbar, um die anderen Layouttechniken zu evaluieren. Zum einen ist die Raketensimulation nicht in Sprachen wie den Zustandsdiagrammen oder Petrinetzen umzusetzen, desweiteren gibt es hier auch keine eingeführte textuelle Notation, mit der verglichen werden könnte.

5.3. Literaturbezug

In diesem Kapitel habe ich die Spezifikationsmodule beschrieben und ihre Anwendung für einige visuelle Beispielsprachen diskutiert. Ein wesentlicher Aspekt der Module ist, daß durch sie ein Großteil der Angaben gekapselt wird, die für die Implementierung eines Sprachkonstrukts einer visuellen Sprache erforderlich sind. Außer Bezügen auf Spezifikationsmodule werden so nur wenige weitere Angaben für die Implementierung einer visuellen Sprache benötigt. Ein Modul überdeckt fast alle Spezifikationen für ein Sprachkonstrukt und implementiert ein visuelles Muster, so daß es vom Sprachentwerfer leicht identifiziert werden kann. Wie ich im vorigen Abschnitt gezeigt habe, lassen sich visuelle Sprachen mit den Modulen besonders kurz und einfach spezifizieren. Weiterhin können die Module flexibel miteinander kombiniert und in vielfältiger Weise angepaßt werden, so daß sich zahlreiche Sprachen mit ihnen realisieren lassen.

Es gibt eine Reihe von Ansätzen, bei denen Module angewendet werden können, um Teile einer visuellen Sprache zu implementieren. Zunächst können im LOGGIE-System [Backlund u.a. 1990] einzelne Attributberechnungen aus einer Bibliothek wiederverwendet werden. Wie bei den Spezifikationsmodulen sind dadurch Teile von Spezifikationen wiederverwendbar. Die Module des LOGGIE-

Sprache	Zeit
VLEli NSD-Editor	108s
Basic	63s
Prograph	194s
LabView	508s

Die Zeiten für Basic, Prograph und LabView sind [Green und Petre 1996] entnommen und gerundet.

Tabelle 5.3.: Zeitbedarf zur Änderung des Testprogramms aus Abb. 5.20

Testprogramm	t_{NSD}	t_{C++}	Faktor t_{NSD}/t_{C++}
Quicksort	165s	138s	1,2
Selection-Sort	125s	86s	1,45

Tabelle 5.4.: Konstruktionsaufwand für einfache Programme

Systems kapseln jedoch sehr feingranular lediglich einzelne Attributberechnungen, so daß sich keine hohe Abstraktionsebene erzielen läßt.

Im weiteren wird in einer Reihe von Ansätzen die Layoutberechnung durch Module gekapselt. Bei GenGEd [Bardohl 1998] sind dies etwa die "*high level constraints*", bei VIVID [Dangberg und Müller 1999] die Layoutmanager und bei LOGGIE die "*edit-semantic-attributes*", die u.a. Constraints der Darstellung beschreiben. Bei diesen Systemen sind für die Sprachimplementierung im Unterschied zu VLEli aber viele weitere, zum Teil komplexe Angaben erforderlich, die nicht in Module gekapselt werden können.

Schließlich basieren Ansätze wie Escalante [McWhirter und Nutt 1994], DV-Centro [Bronwn 1997] oder Unidraw [Vlissides und Linton 1990] auf den Generalisierungsmechanismen objektorientierter Sprachen. Bei diesen Ansätzen wird ein Großteil der zur Sprachimplementierung notwendigen Angaben durch Module abgedeckt, die als Klassen realisiert sind. Jedes einzelne Modul kapselt jedoch nur einzelne Aspekte der Sprachimplementierung – eine bestimmte Layouttechnik oder eine bestimmte Editieroperation. Zur korrekten Anwendung der Module sind umfangreiche Kenntnisse der jeweiligen Bibliothek erforderlich, so daß diese Systeme schwierig zu benutzen sind.

6. Zusammenfassung und Ausblick

In dieser Arbeit habe ich eine Methode vorgestellt, mit der Entwicklungsumgebungen für visuelle Sprachen generiert werden können. Die erzeugten Entwicklungsumgebungen umfassen einen spezialisierten Strukturedi tor für eine visuelle Sprache, sowie Werkzeuge, mit denen Programme der Sprache analysiert und weiterverarbeitet werden können. Die Methode wurde im Rahmen dieser Arbeit praktisch umgesetzt und prototypisch untersucht. Im folgenden fasse ich wesentliche Aspekte meiner Arbeit zusammen und gehe auf Ideen für deren Weiterführung ein.

Programmrepräsentation

Meine Methode unterscheidet sich von anderen Ansätzen durch die Repräsentation, die zur Speicherung der visuellen Programme verwendet wird. Ich benutze dazu Strukturbäume und Referenzen auf Einträge in eine Definitionstabelle. So lassen sich zum einen auch diejenigen visuellen Programme repräsentieren, die nichthierarchische Beziehungen enthalten. Zum anderen können Werkzeuge und Generatoren für die Verarbeitung textueller Sprachen eingesetzt werden, um auch visuelle Sprachen zu implementieren. Die für Analyse und Weiterverarbeitung zuständigen Teile der Entwicklungsumgebungen können so wesentlich einfacher als bei anderen Ansätzen implementiert werden. Dies wirkt sich auch auf die Benutzbarkeit der generierten Strukturedi toren aus, da die Analyseergebnisse auch für die Implementierung “intelligenter” Editieroperationen verwendet werden können.

Durch die Programmrepräsentation können einige einfache Strukturen, das sind Tupel und Alternativen direkt dargestellt werden. Die komplexeren Strukturen wie Sequenzen, Mengen, Tabellen und Matrizen werden aus den einfachen aufgebaut. Im VLEli-System werden dabei Sequenzen transparent für den Sprachentwickler unterstützt. Indem Konstrukte zum Spezifizieren und Operationen zum Editieren von Sequenzen implementiert wurden, werden Sequenzen auf Tupel und Alternativen zurückgeführt.

In zukünftigen Arbeiten könnte untersucht werden, inwieweit es lohnend ist, auch andere Strukturen wie z.B. Matrizen in ähnlicher Weise zu unterstützen. Die Repräsentation der Matrizen ist auch deswegen interessant, weil nicht alle Beziehungen durch die Baumstruktur repräsentiert werden können und durch Einträge

in die Definitionstabelle dargestellt werden müssen. Indem Matrizen transparent unterstützt werden, kann also nicht nur das Editieren von Matrizen verbessert, sondern auch die Spezifikation visueller Sprachen mit Matrizen weiter vereinfacht werden. Allerdings kommen Matrizen in visuellen Sprachen weitaus seltener vor als Sequenzen, so daß untersucht werden muß, ob sich der Implementierungsaufwand lohnt.

Spezifikationsmodule

Eine Konsequenz aus der Methode der Programmrepräsentation ist, daß insbesondere der Generator Liga zur Sprachimplementierung benutzt werden kann. Liga generiert Attributauswerter aus attributierten Grammatiken und integriert effektive Techniken, mit denen diese Grammatiken modularisiert werden können. Da in meinem Ansatz umfangreiche Teile der Strukturediteren durch attributierte Grammatiken spezifiziert werden, kann ein Großteil des Know-Hows der Sprachimplementierung in Spezifikationsmodule eingebettet werden. Mit den Modulen können so visuelle Sprachen in einfacher Weise und auf hohem Niveau implementiert werden.

Die Spezifikationsmodule kapseln Berechnungen, die in verschiedenen Kontexten des Strukturbaums ausgeführt werden sollen. Ein Spezifikationsmodul besteht deshalb aus mehreren Rollen, die angewendet werden, indem sie an Symbole der attributierten Grammatik vererbt werden. Um die Spezifikationsmodule in der richtigen Weise zu benutzen, müssen einige Bedingungen eingehalten werden. Ausdrucksmittel, um die konsistente Anwendung von Rollen zu gewährleisten und die bei Fehlern zu sinnvollen Fehlermeldungen führen, sind im Liga-System nur unzureichend vorhanden. Dieser Mangel wird auch außerhalb des Bereichs der Implementierung visueller Sprachen, etwa bei der Benutzung der Typ-Module des Eli-Systems deutlich.

In zukünftigen Arbeiten könnte das Liga-System erweitert oder eine Spezialsprache zur Anwendung der Spezifikationsmodule eingeführt werden. Damit könnte die Qualität der Fehlermeldungen deutlich verbessert werden, was die Benutzung des VLEli-Systems für unerfahrere Sprachentwerfer weiter vereinfachen würde. In einer dazu entworfenen Spezialsprache könnte ferner eine prägnante visuelle Notation verwendet werden. Ein Sprachentwerfer könnte dadurch die Beziehungen zwischen den Spezifikationsmodulen besser erfassen und diese leichter richtig anwenden. Eine zu diesem Zeitpunkt entwickelte, hierfür geeignete Sprache könnte auf den Notationen aufbauen, die ich für die Beschreibung der Spezifikationsmodule angegeben habe.

Um eine prägnante Notation für eine solche Spezialsprache zu entwickeln, sollten jedoch zunächst tiefergehende Kenntnisse über die Bedingungen erarbeitet werden, die für die korrekte Anwendung der zukünftig entwickelten Spezifikationsmodule eingehalten werden müssen. Im Rahmen dieser Arbeit sind lediglich einige wenige Spezifikationsmodule entstanden. Diese waren bereits ausreichend,

um eine Reihe visueller Sprachen mit stark unterschiedlicher graphischer Darstellung zu implementieren und die Methode der Sprachimplementierung so zu untersuchen. Bei diesem ersten Prototyp einer Spezifikationsmodulbibliothek treten aller Wahrscheinlichkeit nach aber noch nicht alle Arten von Abhängigkeiten und Konsistenzbedingungen auf. Im Rahmen der Weiterentwicklung könnten etwa gemeinsame Eigenschaften der Spezifikationsmodule durch neue Module gekapselt werden. Derartige Erweiterungen führen u.U. zu andersartigen Bedingungen für die korrekte Benutzung der Module, die sich in einer vorzeitig entwickelten Notation nicht konsistent integrieren ließen.

Layoutberechnung

Ein weiterer Aspekt dieser Arbeit ist, daß die mit den Spezifikationsmodulen erzeugten Strukturediteren in hohem Maße benutzerfreundlich sind. In jedem Spezifikationsmodul können Layout und Editieroperationen so fein aufeinander abgestimmt werden, daß sich das typische, vom Benutzer erwartete Verhalten der Sprachkonstrukte einstellt. Um dies zu erreichen, können unterschiedliche Techniken der Layouterzeugung benutzt und auch miteinander kombiniert werden. Hierzu wurden die Berechnungen eines Constraint-Solvers und eines Attributauswerters zu *einer* Methode der Layoutberechnung integriert, die von den Spezifikationsmodulen in der jeweils gewünschten Weise spezialisiert werden kann.

Ein Problem dieser Kombination betrifft ungünstige Schachtelungen von Sprachkonstrukten mit unterschiedlicher Technik der Layouterzeugung. Hierbei bewirken die Abhängigkeiten bei der Größenberechnung, daß der Constraint-Solver u.U. mehrfach gestartet werden muß, um das Layout einer graphischen Darstellung zu berechnen. Dies reduziert zum einen die Effizienz der generierten Strukturediteren. Zum anderen können die in einem Constraint-Solver-Lauf getroffenen Entscheidungen später nicht mehr revidiert werden. Das führt dazu, daß nicht alle idealerweise möglichen Layoutkorrekturen automatisch durchgeführt werden können.

In dieser Arbeit wurde in erster Linie das Effizienzproblem gelöst, das der mehrfache Aufruf des Constraint-Solvers darstellt. Die umgesetzte Lösung für dieses Problem bewirkt aber auch, daß in Einzelfällen die Benutzerfreundlichkeit der generierten Editoren weiter eingeschränkt wird. In zukünftigen Arbeiten könnte eine Lösung erarbeitet werden, die den Aspekt der Benutzerfreundlichkeit stärker berücksichtigt.

Visuelle Muster

Die konzeptionelle Grundlage für die Spezifikationsmodule bilden die visuellen Muster, die ich in dieser Arbeit eingeführt habe. Die Muster beschreiben die graphische Darstellung von Sprachkonstrukten visueller Sprachen und die Editieroperationen, die benutzt werden sollten, um mit diesen Sprachkonstrukten zu interagieren.

Die visuellen Muster sind auch für den *Entwurf* visueller Sprachen geeignet. Zum ersten lassen sie sich leicht anhand der abstrakten Struktur einer Sprache identifizieren. Ein Sprachentwickler erhält dadurch eine Übersicht über die Techniken, mit denen ein Sprachkonstrukt visualisiert werden kann. Zum zweiten beschreiben die Muster bewährte, vielfach eingesetzte Darstellungstechniken und kapseln somit das Entwurfs-Know-How visueller Sprachen. Dies wurde durch die Auswahl der zur Erarbeitung der Muster untersuchten Sprachen und durch die Vorgehensweise bei der Identifikation der Muster erreicht. Zum dritten sind den Mustern Konsequenzen zugeordnet, die die Anwendung eines Musters für eine visuelle Sprache hat. Diese Konsequenzen drücken sich etwa in Form von Ausprägungen bestimmter Eigenschaften der entworfenen Sprache aus. Sie können die Grundlage für die Auswahl eines aus einer Reihe alternativer visueller Muster für ein Sprachkonstrukt bilden.

Die Spezifikationsmodule, die im Rahmen dieser Arbeit entwickelt wurden, setzen jeweils die Visualisierungs- und Interaktionstechniken eines Musters in wiederverwendbare Spezifikationen um. Sprachen, die mit visuellen Mustern entworfen wurden, können somit leicht auch implementiert werden, weil die zu einem visuellen Muster gehörenden Spezifikationsmodule leicht identifizierbar sind. Bei der Entwicklung der Spezifikationsmodule wurde im weiteren durch die visuellen Muster erreicht, daß die Module sich zur Implementierung einer großen Menge visueller Sprachen eignen und daß Module gut gegeneinander ausgetauscht werden können, wenn sie ähnliche abstrakte Strukturen visualisieren.

Die Auswirkungen der visuellen Muster auf den Entwurf visueller Sprachen wurden in dieser Arbeit lediglich durch die Entwicklung einiger Beispielsprachen untersucht. In diesem Bereich könnten durch zukünftige Arbeiten weitere Erkenntnisse darüber gewonnen werden, inwieweit der Sprachentwurf durch visuelle Muster vereinfacht wird. Eine solche Untersuchung kann auch zur Identifizierung neuer visueller Muster führen.

Die in dieser Arbeit vorgestellte Methode kapselt das Expertenwissen für den Entwurf und die Implementierung visueller Sprachen. Die Methode eignet sich für eine große Klasse visueller Sprachen. Mit ihr können auch Werkzeuge erzeugt werden, die die von Benutzern eingegebenen visuellen Programme weiterverarbeiten. Ich habe mich in dieser Arbeit auf die Entwicklung dieser Methode und ihre Integration in das Eli-System konzentriert. Die prototypische Untersuchung der Methode hat ergeben, daß wesentliche Entwicklungsziele erreicht wurden. Visuelle Sprachen können in einfacher Weise und auf einer hohen Abstraktionsebene spezifiziert werden. Die erzeugten Entwicklungsumgebungen enthalten Editoren, die benutzerfreundlich und effektiv einsetzbar sind. Die vorgestellte Methode wird deshalb ihren Anteil an der weiteren Forschung im Bereich der visuellen Sprachen haben.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Albizuri-Romero 1984 Miren Begona Albizuri-Romero. GRASE - a graphicsl syntax-directed editor for structured programming. *SIGPLAN Notices*, 19 (2):28–37, Februar 1984.

Alblas und Melichar 1991 Henk Alblas und Bořivoj Melichar (Hrsg). *Attribute Grammars, Applications and Systems*, Band 545 d. Reihe *Lecture Notes in Computer Science*. Springer Verlag, New York–Heidelberg–Berlin, Juni 1991. Prague.

Alexander u.a. 1977 Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King und Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

Andries u.a. 1998 M. Andries, G. Engels und J. Rekers. How to represent a visual specification. In Kim Marriott und Bernd Meyer (Hrsg), *Visual Language Theory*, Kapitel 8. Springer, 1998.

Arefi u.a. 1990 Farahangiz Arefi, Charles E. Hughes und David A. Workman. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3):349–360, März 1990.

Augusteijn 1990 Lex Augusteijn. The elegant compiler generation system. In Pierre Deransart und Martin Jourdan (Hrsg), *Attribute Grammars and their Applications (WAGA)*, Band 461 d. Reihe *Lecture Notes in Computer Science*, S. 238–254. Springer Verlag, New York–Heidelberg–Berlin, September 1990. Paris.

Backlund u.a. 1990 B. Backlund, O. Hagsand und B. Pherson. Generation of visual language-oriented design environments. *Journal of Visual Language and Computing*, 1(4):333–354, 1990.

Backlund u.a. 1989 Björn Backlund, Olof Hagsand und Björn Pehrson. Generation of graphic language-oriented design environments. SICS research report R-89/8911, Swedish Institute of Computer Science, 1989.

Bahlke und Snelting 1992 R. Bahlke und G. Snelting. Design and structure of a

semantics-based programming environment. *International Journal of Man-Machine Studies*, 37:469–479, 1992.

Bardohl 1998 Roswitha Bardohl. GenGed: A generic graphical editor for visual languages based on algebraic graph grammars. In *Proceedings 1998 IEEE Symposium on Visual Languages*, S. 48–55, Halifax, Nova Scotia, September 1998. IEEE Computer Society Press.

Bardohl 2000 Roswitha Bardohl. *GenGED - Visual Definition of Visual Languages based on Algebraic Graph Transformation*. Doktorarbeit, Technische Hochschule Berlin, Deutschland, 2000.

Bardohl und Taentzer 1997 Roswitha Bardohl und Gabriele Taentzer. Defining visual languages by algebraic specification techniques and graph grammars. In *Proceedings Workshop on Theory of Visual Languages*, S. 27–42, Capri, Italy, September 1997.

Barford und Vander Zanden 1989 Lee A. Barford und Bradley T. Vander Zanden. Attribute grammars in constraint-based graphics systems. *Software—Practice and Experience*, 19(4):309–328, April 1989.

Bartoli u.a. 1995 Alberto Bartoli, Paolo Corsini, Gianuca Dini und Antonio Prete. Graphical design of distributed applications trough reusable components. In *IEEE Parallel & Distributed Technology*, S. 37–50, Spring 1995.

Beguelin und Dongarra 1991 Adam Beguelin und Jack Dongarra. Graphical development tools for network-based concurrent supercomputing. In *Proc. Supercomputing'91*, S. 435–444, Los Alamitos, Calif., 1991. IEEE Computer Society Press.

Belina u.a. 1991 Ferenc Belina, Dieter Hogrefe und Amardeo Sarma. *SDL with Applications from Protocol Specification*. BCS Practitioner Series. Carl Hanser Verlag, 1991.

Bohm und Jacopini 1966 C. Bohm und G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5), 1966.

Booch 1991 Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin / Cummings, 1991.

Borning 1981 A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, Oktober 1981. Auch in [Glinert 1990b, S. 416–449].

Borning 1979 Alan Hamilton Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory*. Doktorarbeit, Stanford University, Juli 1979.

Bottoni u.a. 1999 P. Bottoni, S. K. Chang, M. F. Costabile, S. Levialdi und P. Mussio. Dimensions of visual interaction design. In *Proceedings 1999 IEEE Symposium on Visual Languages*, S. 48–55, Tokyo, Japan, September, 13–16 1999. IEEE Computer Society Press.

Bronwn 1997 P. C. Bronwn. Satisfying the graphical requirements of visual languages in the dv-centro framework. In *Proceedings 1997 IEEE Symposium on Visual Languages*, S. 84–91. IEEE Computer Society Press, 1997.

Burnett u.a. 1995 M. M. Burnett, A. Goldberg und T. G. Lewis (Hrsg). *Visual Object-Oriented Programming: Concepts and Environments*. Manning, 1995.

Burnett u.a. 1995 M. M. Burnett u.a.. Scaling up visual programming languages. *IEEE Computer*, 28(3):45–54, März 1995.

Burnett 2000 Margaret M. Burnett. Visual language research bibliography. http://www.cs.orst.edu/burnett/vpl.html, August 2000.

Burnett und Baker 1994 Margaret M. Burnett und Marla J. Baker. A classification system for visual programming languages. *Journal of Visual Language and Computing*, 5(3):284–300, September 1994.

Card u.a. 1983 S. K. Card, T. P. Moran und A. Newell. *The Psychology of Human-Computer Interaction*. Erlbaum, Hillsdale, NJ, 1983.

Cardelli 1983 L. Cardelli. Two-dimensional syntax for functional languages. In P. Degano und E. Sandwall (Hrsg), *Integrated Interactive Computing Systems - Proceedings of the European ECICS 82 Conference, Stresa*. Elsevier, 1983. Auch in [Glinert 1990b, S. 188–196].

Chabrier u.a. 1988 Bruno Chabrier, Vincent Lextrait und Paul Franchi-Zannettacci. GIGAS: a graphical interface generator from attribute specifications. In *Actes des Journées Int. "Le Génie Logiciel et ses Applications"*, S. 1265–1283. EC2, Paris, Dezember 1988. Toulouse.

Chan und Lee 1998 Patrick Chan und Rosanna Lee. *The Java Class Libraries*, Band 2 d. Reihe *The Java Series from the Source*. Addison Wesley Longman, Reading, MA, zweite Auflage, 1998.

Chang 1987 Shi-Kuo Chang. Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39, Januar 1987. Auch in [Glinert 1990b, S. 7–17].

Chang u.a. 1989 Shi-Kuo Chang, Michael J. Tauber, Bing Yu und Jing-Sheng Yu. A visual language compiler. *IEEE Transactions on Software Engineering*, SE-15 (5):506–525, 1989.

Chen und Bovik 1990 Dapang Chen und Alan C. Bovik. Visual pattern image coding. *IEEE Transaction on Communications*, 38(12):2137–2146, Dezember 1990.

Chok und Marriott 1998 Sit Sen Chok und Kim Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of the 11th Annual Symposium on User Interface Software and Technology*, S. 185–194, San Francisco, California, November 1998. IEEE Computer Society Press.

Chok u.a. 1999 Sit Sen Chok, Kim Marriott und Tom Paton. Constraint-based diagram beautification. In *Proceedings 1999 IEEE Symposium on Visual Languages*, S. 12–19, Tokyo, Japan, September, 13–16 1999. IEEE Computer Society Press.

Citrin 1993 Wayne Citrin. Requirements for graphical front ends for visual languages. In Ephraim P. Glinert und Kai A. Olsen (Hrsg), *Proceedings of the 1993 IEEE Symposium on Visual Languages*, S. 142–150. IEEE Press, 24–27 August 1993.

Coplien 1992 James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, USA, 1992. An advanced book for any C++ expert-wanna-be.

Coplien 1995 James Coplien. A generative development process pattern language. In James O. Coplien und Douglas C. Schmidt (Hrsg), *Pattern Languages of Program Design*, S. 183–237. Addison-Wesley Publishing Company, 1995.

Costagliola u.a. 1997a G. Costagliola, A. de Lucia, S. Orefice und G. Tortora. A framework of syntactic models for the implementation of visual languages. In *Proceedings 1997 IEEE Symposium on Visual Languages*, Isle of Capri, Italy, September, 23–26 1997. IEEE Computer Society Press.

Costagliola u.a. 1999 G. Costagliola, F. Ferrucci, G. Polese und G. Vitiello. Supporting hybrid and hierarchical visual language definition. In *Proceedings 1999 IEEE Symposium on Visual Languages*, S. 236–243, Tokyo, Japan, September, 13–16 1999. IEEE Computer Society Press.

Costagliola u.a. 1997b Gennaro Costagliola, Andrea De Lucia, Sergio Orefice und Genoveffa Tortora. A parsing methodology for the implementation of visual systems. *IEEE Transactions on Software Engineering*, 23(12):777–799, Dezember 1997.

Cournarie und Beaudouin-Lafon 1995 Eric Cournarie und Michel Beaudouin-Lafon. Alien: A prototype-based constraint system. In *Object-Oriented Programming for Graphics*, 1995. Überarbeitete Version eines Papiers aus EUROGRAPHICS Workshop Object-Oriented Graphics, Texel, Niederlande, 4.-7. Juni 1991.

Cox u.a. 1989 P. T. Cox, F. R. Giles und T. Pietrzykowsky. Prograph: A step towards libarating programming from textual conditioning. In *IEEE Proc.*

Workshop on Visual Languages, S. 150–156, 1989. Auch in [Burnett u.a. 1995, Kapitel 3].

Cox und Pietrzykowsky 1988 P. T. Cox und T. Pietrzykowsky. Using a pictorial representation to combine dateflow and object-orientation in a language independant programming mechanism. In *Proceedings International Computer Science Conference*, S. 695–704, 1988. Auch in [Glinert 1990b, S. 313–322].

Cox u.a. 1998 Philip T. Cox, Hugh Glaser und Stuart Maclean. A visual development environment for parallel applications. In *Proceedings 1998 IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, September 1998. IEEE Computer Society Press.

Crimi u.a. 1990 Claudia Crimi u.a.. Automating visual language generation. *IEEE Transactions on Software Engineering*, SE-16(10):1122–1135, 1990.

Cypher 1994 Allen Cypher (Hrsg). *Watch what I do: Programming by Demonstration*. MIT Press, 1994.

Dangberg und Müller 1999 Andreas Dangberg und Wolfgang Müller. Generation of interactive visual environments for direct manipulation of database content. In *Proceedings 1999 IEEE Symposium on Visual Languages*, S. 178–179, Tokyo, Japan, September, 13–16 1999. IEEE Computer Society Press.

Day 1988 R. S. Day. Alternative representations. *The Psychology of Learning and Motivation*, 22:261–305, 1988.

Deransart und Jourdan 1990 Pierre Deransart und Martin Jourdan (Hrsg). *Attribute Grammars and their Applications (WAGA)*, Band 461 d. Reihe *Lecture Notes in Computer Science*. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.

Deransart u.a. 1988 Pierre Deransart, Martin Jourdan und Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, Band 323 d. Reihe *Lecture Notes in Computer Science*. Springer-Verlag, New York–Heidelberg–Berlin, August 1988.

Dueck und Cormack 1990 Gerald D. P. Dueck und Gordon V. Cormack. Modular attribute grammars. *The Computer Journal*, 33:164–172, 1990. See also: research report CS-88-19, University of Waterloo (May 1988).

Eli 2000 Eli: Compiler construction made easy. http://www.upb.de/cs/agkastens/eli_homeE.html, 2000.

Engels u.a. 1986 G. Engels, M. Nagl und W. Schaefer. On the structure of structure-oriented editors for different applications. In Peter Henderson (Hrsg), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, S. 190–198, 1986.

Erwig und Meyer 1995 M. Erwig und B. Meyer. Heterogeneous visual languages - integrating textual and visual programming. In *Proceedings 1995 IEEE Symposium on Visual Languages*, S. 318–325, Darmstadt, Germany, September 1995. IEEE Computer Society Press.

Erwig 1997 Martin Erwig. Abstract visual syntax. In *Proceedings 2nd IEEE International Workshop on Theory of Visual Languages*, S. 15–25, 1997.

Erwig 1998 Martin Erwig. Abstract syntax and semantics of visual languages. *Journal of Visual Language and Computing*, 9:461–483, Oktober 1998.

Farrow u.a. 1992 Rodney Farrow, Thomas J. Marlowe und Daniel M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Programming Languages*, S. 223–234, Albuquerque, NM, Januar 1992. ACM press.

Franchi-Zannettacci 1989 Paul Franchi-Zannettacci. Attribute specifications for graphical interface generation. In G. X. Ritter (Hrsg), *Information Processing '89*, S. 149–155. North-Holland, Amsterdam, August 1989.

Freeman-Benson u.a. 1990 Bjorn N. Freeman-Benson, John Maloney und Alan Borning. An incremental constraint-solver. *Communications of the ACM*, 33 (1):54–63, Januar 1990.

Fukushima 1995 K. Fukushima. Neocognitron: A model for visual pattern recognition. In Michael A. Arbib (Hrsg), *The Handbook of Brain Theory and Neural Networks*, S. 613–617. MIT Press, Cambridge, Massachusetts, 1995.

Gamma u.a. 1996 Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison Wesley, Reading, MA, 1996.

Ganzinger und Giegerich 1984 Harald Ganzinger und Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, S. 157–170, Montréal, Juni 1984. ACM press. Published as *ACM SIGPLAN Notices*, 19(6).

Geisser 1998 Oliver Geisser. Generierung von Strukturediteren aus Übersetzerspezifikationen. Diplomarbeit, Universität Paderborn, Deutschland, Juli 1998.

Geist und Sunderam 1992 G. Geist und V S. Sunderam. Network-based concurrent computing on the pvm system. *Concurrency: Practice and Experiences*, 4 (4):293–311, 1992.

Giacalone und Smolka 1988 A. Giacalone und S. A. Smolka. Integrated environments for formally well-founded design and simulation of concurrent systems. *IEEE Transactions on Software Engineering*, 14:787–802, 1988.

Giese u.a. 1999 Holger Giese, Jörg Graf und Guido Wirtz. Seamless visual object-oriented behaviour modeling for distributed software systems. In *Proceedings of the 1999 IEEE Symposium on Visual Languages*, Tokyo, Japan, September 1999. IEEE Computer Society Press.

Gilmore und Green 1984 D. J. Gilmore und T. R. G. Green. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, 21(1):31–48, 1984.

Glinert 1987 Ephraim P. Glinert. Out of flatland: Towards 3-D visual programming. In *IEEE Proceedings 2nd Joint Computer Conference*, S. 292–299, 1987.

Glinert 1990a Ephraim P. Glinert. Nontextual programming environments. In Shi-Kuo Chang (Hrsg), *Principles of Visual Programming Systems*. Plenum Press, 1990.

Glinert 1990b Ephraim P. Glinert (Hrsg). *Visual Programming Environments: Paradigms and Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

Glinert und Gonczarowski 1987 Ephraim P. Glinert und Jakob Gonczarowski. A (formal) model for (iconic) programming environments. In *Proceedings of IFIP INTERACT'87: Human-Computer Interaction*, 1. Human Factors in System Development: 1.12 Programming Tools and Environments I, S. 283–290, 1987.

Golin und Magliery 1993 E. J. Golin und T. Magliery. A compiler generator for visual languages. In *Proceedings 1993 IEEE Symposium on Visual Languages*, S. 314–321, Bergen, Norway, August 1993. IEEE Computer Society Press.

Golin u.a. 1989 Eric Golin, Robert V. Rubin und James Walker II. The visual programmers workbench (extended abstract). In G. X. Ritter (Hrsg), *Information Procedding '89*, S. 143–148. North-Holland, Amsterdam, August 1989.

Good 1999 Judith Good. Vpls and novice program comprehension: How do different languages compare. In *Proceedings 1999 IEEE Symposium on Visual Languages*, S. 262–269, Tokyo, Japan, September 1999. IEEE Computer Society Press.

Graf 1987 Michael L. Graf. A visual environment for the design of distributed systems. In *Proceedings of the 1987 International Workshop on Visual Languages*, S. 330–343, August 1987.

Grant 1998 Calum A. M. Grant. Visual language editing using a grammar-based visual structure editor. *Journal of Visual Languages and Computing*, 9:351–374, 1998.

Gray u.a. 1992 R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane und W. M. Waite.

Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM*, 35(2):121–131, Februar 1992.

Green 1989 T. R. G. Green. Cognitive dimensions of notations. In A. Sutcliffe und L. Macaulay (Hrsg), *People and Computers V – Proceedings of the Fifth Conference of the British Computer Society*, S. 443–460. Cambridge University Press, 1989.

Green und Petre 1992 T. R. G. Green und M. Petre. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings 6th European Conference on Cognitive Ergonomics*, 1992.

Green und Petre 1993 T. R. G. Green und M. Petre. Learning to read graphics: Some evidence that 'seeing' an information display is an aquired skill. *Journal of Visual Language and Computing*, 4:55–70, 1993.

Green und Petre 1996 T. R. G. Green und M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Language and Computing*, 7(2):131–174, 1996.

Griebel 1996 Peer Griebel. *Parcon – Paralleles Lösen von Graphischen Constraints*. Dissertation, Universität Paderborn, Februar 1996.

Griebel u.a. 1996 Peer Griebel, Wolfgang Müller, Christoph Tahedel und Holger Uhr. Integrating a constraint solver into a real-time animation environment. In *Proceedings 1996 IEEE Symposium on Visual Languages*, S. 12–19. IEEE Computer Society Press, 1996.

Griebel u.a. 1997 Peer Griebel, Manfred Pöpping, Gerd Szwillus und Holger Uhr. Parcon: Ein schneller Solver für graphische Constraints. *KI - Künstliche Intelligenz: Themenheft Constraints*, 1997.

Grosch 1989 Josef Grosch. AG - an attribute evaluator generator. Report 16, GMD Forschungsstelle Karlsruhe, Germany, 1989.

Grosch 1999 Josef Grosch. Are attribute grammars used in industry? In D. Parigot und M. Mernik (Hrsg), *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, S. 1–15, Amsterdam, The Netherlands, März 1999. INRIA rocquencourt.

Grundy u.a. 1998 J. C. Grundy, W. B. Mugridge und J. G. Hosking. Visual specification of multi-view visual environments. In *Proceedings 1998 IEEE Symposium on Visual Languages*, S. 236–243, Halifax, Nova Scotia, September 1998. IEEE Computer Society Press.

Göttler 1986 Herbert Göttler. Graph grammars and diagram editing. In *Graph Grammars and Their Applications to Computer Science (Third International*

Workshop), Band 291 d. Reihe *LNCS*, S. 211–231. Springer Verlag, New York, 1986.

Göttler 1989 Herbert Göttler. Graph grammars, A new paradigm for implementing visual languages. In Nachum Dershowitz (Hrsg), *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Band 355 d. Reihe *Lecture Notes in Computer Science*, S. 152–166, Chapel Hill, NC, April 1989. Springer-Verlag, Berlin.

Göttler 1992 Herbert Göttler. Diagram editors = graphs + attributes + graph grammars. *International Journal of Man-Machine Studies*, 37:481–502, 1992.

Harel 1988 David Harel. On visual formalisms. *Communications of the ACM*, 31 (5):514–530, Mai 1988.

Hedin 1989 G. Hedin. An object-oriented notation for attribute grammars. In *3rd European Conference on Object-Oriented Programming (ECOOP'89)*, BCS Workshop Series, S. 329–345, Nottingham, U.K., Juli 1989. Cambridge University Press.

Hedin 1992 G. Hedin. *Incremental Semantic Analysis*. Ph.D. thesis, Lund University, Lund, Sweden, 1992.

Hedin 1991 Görel Hedin. Incremental static-semantics analysis for object-oriented languages using door attribute grammars. In Henk Alblas und Bořivoj Melichar (Hrsg), *Attribute Grammars, Applications and Systems*, Band 545 d. Reihe *Lecture Notes in Computer Science*, S. 374–379. Springer-Verlag, New York–Heidelberg–Berlin, Juni 1991. Prague.

Hedin 1999 Gorel Hedin. Reference Attributed Grammars. In D. Parigot und M. Mernik (Hrsg), *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, S. 153–172, Amsterdam, The Netherlands, März 1999. INRIA rocquencourt.

Hedin 1994 Görel Hedin. An overview of door attribute grammars. In Peter A. Fritzson (Hrsg), *5th Int. Conf. on Compiler Construction (CC' 94)*, Band 786 d. Reihe *Lecture Notes in Computer Science*, S. 31–51, Edinburgh, April 1994.

Hendry und Green 1994 D. G. Hendry und T. R. G. Green. Creating, comprehending and explaining spreadsheets: A cognitive interpretation of what discretionary users think of the spreadsheet model. *International Journal of Human-Computer Studies*, 40:1033–1065, 1994.

Hirakawa u.a. 1990 Masahito Hirakawa, Minoru Tanaka und Tadao Ichikawa. An iconic programming system, hi-visual. *IEEE Transactions on Software Engineering*, SE-16(10):1178–1184, October 1990.

Hood 1985 Robert Hood. Efficient abstractions for the implementation of struc-

tured editors. In *Proceedings of the ACM SIGPlan '85 Symposium on Language Issues in Programming Environments*, S. 171–178, Seattle, WA, Juli 1985.

Jacobson u.a. 1992 Ivar Jacobson, Magnus Christerson, Patrik Jonsson und Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.

Jamal und Wenzel 1995 Rahman Jamal und Lothar Wenzel. The applicability of the visual programming language LABView to large real-world applications. *IEEE Proc. Symp. on Visual Languages*, S. 99–105, 1995.

Jamal und Wenzel 1997 Rahman Jamal und Lothar Wenzel. *LabVIEW – Programmiersprache der vierten Generation*. Prentice Hall, 1997.

Johnson u.a. 1993 Jeff Johnson, Bonnie Nardi, Craig Zarmer und James Miller. ACE: Building interactive graphical applications. *Communications of the ACM*, 36(4):41–55, April 1993.

Jourdan u.a. 1990 Martin Jourdan, Carole Le Bellec und Didier Parigot. The OLGA attribute grammar description language: Design, implementation and evaluation. In *Attribute Grammars and their Applications (WAGA)*, Band 461 d. Reihe *Lecture Notes in Computer Science*, S. 222–237, New York–Heidelberg–Berlin, September 1990. Springer-Verlag.

Jourdan und Parigot 1991 Martin Jourdan und Didier Parigot. Internals and externals of the FNC-2 attribute grammar system. In Henk Alblas und Bořivoj Melichar (Hrsg), *Attribute Grammars, Applications and Systems*, Band 545 d. Reihe *Lecture Notes in Computer Science*, S. 485–504, New York–Heidelberg–Berlin, Juni 1991. Springer-Verlag.

Jung u.a. 2000 Matthias T. Jung, Uwe Kastens, Christian Schindler und Carsten Schmidt. A pattern-based generator for implementation of visual languages. In *Proceedings 2000 IEEE International Symposium on Visual Languages*, S. 71–72, Seattle, Washington, September 2000. IEEE Computer Society Press.

Kahn und Sraswat 1990 K. M. Kahn und V. A. Sraswat. Complete visualizations of concurrent programs and their execution. In *IEEE Proc. Workshop on Visual Languages*, S. 7–15. IEEE Computer Society Press, 1990.

Kamada und Kawai 1991 Tomihisa Kamada und Satoru Kawai. A general framework for visualizing abstract objects and relations. *ACM Transactions on Graphics*, 10(1):1–39, Januar 1991.

Kastens 1976 Uwe Kastens. Ein Übersetzer-erzeugendes System auf der Basis Attributierter Grammatiken. Dissertation, Interner Bericht 10, Fakultät für Informatik, University Karlsruhe, September 1976.

Kastens 1980 Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3): 229–256, 1980.

Kastens 1987 Uwe Kastens. Lifetime analysis for attributes. *Acta Informatica*, 24 (6):633–652, November 1987.

Kastens 1990 Uwe Kastens. *Übersetzerbau*. Handbuch der Informatik. Oldenbourg Verlag, München, 1990.

Kastens 1991a Uwe Kastens. An attribute grammar system in a compiler construction environment. In H. Alblas und B. Melichar (Hrsg), *Proceedings of the International Summer School on Attribute Grammars, Application and Systems*, Band 545 d. Reihe *Lecture Notes on Computer Science*, S. 380–400. Springer Verlag, 1991.

Kastens 1991b Uwe Kastens. Attribute grammars as a specification method. In H. Alblas und B. Melichar (Hrsg), *Proceedings of the International Summer School on Attribute Grammars, Application and Systems*, Band 545 d. Reihe *Lecture Notes on Computer Science*, S. 16–47. Springer Verlag, 1991.

Kastens 1999 Uwe Kastens. Lido - reference manual. http://www.upb.de/cs/ag-kastens/elionline4.3/lidoref_toc.html, August 1999. Revision 4.25.

Kastens u.a. 1982 Uwe Kastens, Brigitte Hutt und Erich Zimmermann. *GAG: A Practical Compiler Generator*, Band 141 d. Reihe *Lecture Notes in Computer Science*. Springer-Verlag, New York–Heidelberg–Berlin, 1982.

Kastens und Jung 1998 Uwe Kastens und Matthias Jung (Hrsg). *Streets Abschlußbericht*. University of Paderborn, 1998. Auch unter http://www.upb.de/cs/ag-kastens/paper/streets.ps.gz.

Kastens u.a. 1998 Uwe Kastens, Peter Pfahler und Matthias Jung. The Eli system. In Kai Koskimies (Hrsg), *Proceedings 7th International Conference on Compiler Construction CC'98*, Band 1383 d. Reihe *Lecture Notes in Computer Science*, S. 294–297. Springer Verlag, März 1998.

Kastens und Waite 1991 Uwe Kastens und William M. Waite. An abstract data type for name analysis. *Acta Informatica*, 28(6):539–558, Juli 1991.

Kastens und Waite 1994 Uwe Kastens und William M. Waite. Modularity and Reusabiltiy in Attribute Grammars. *Acta Informatica*, 31:601–627, 1994.

Knuth 1979 D. E. Knuth. *TEX and METAFONT, New Directions in Typesetting*. Digital Press, Billerica, MA, 1979.

Knuth 1968 Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, Juni 1968. Korrektur: *Mathematical Systems Theory*, 5(1): 95–96, März 1971.

Koskimies 1991 Kai Koskimies. Object-orientation in attribute grammars. In Henk Alblas und Bořivoj Melichar (Hrsg), *Attribute Grammars, Applications and Systems*, Band 545 d. Reihe *Lecture Notes in Computer Science*, S. 297–329. Springer-Verlag, New York–Heidelberg–Berlin, Juni 1991. Prague.

Kraemer-Fuhrmann u.a. 1993 O. Kraemer-Fuhrmann, L. Schaefers und C. Scheidler. TRAPPER - A graphical programming environment for parallel systems. In *International Workshop on Software Engineering for High Energy Physics*, Oberammergau, Oktober 1993.

Kranzlmüller u.a. 1999 Dieter A. Kranzlmüller, Nenad Stankovic und Jens Volkert. Debugging parallel programs with visual patterns. In *Proceedings 1999 IEEE Symposium on Visual Languages*, S. 180–181, Tokyo, Japan, September 1999. IEEE Computer Society Press.

Krasner und Pope 1988 Glenn E. Krasner und Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August / September 1988.

Kruglinski 1994 David J. Kruglinski. *Inside Visual C++: Version 1.5*. Microsoft Press, Bellevue, WA, USA, zweite Auflage, 1994.

Kuiper und Saraiva 1998 Matthijs Kuiper und João Saraiva. Lrc: A generator for incremental language-oriented tools. In Kai Koskimies (Hrsg), *Compiler Construction CC'98*, Band 1383 d. Reihe *Lecture Notes in Computer Science*, S. 298–301, Portugal, April 1998. Springer Verlag.

Lextrait und Zarli 1990 Vincent Lextrait und Alain Zarli. Meta-generation of incremental and graphical structure-oriented editors. *BIGRE*, 70, September 1990.

Lieberman 2000 Henry Lieberman. Programming by example (introduction). *Communications of the ACM*, 43(3):72–74, März 2000.

Lin u.a. 1997 Han X. Lin, Yee-Yin Choong und Gavriel Salvendy. A proposed index of usability: A method for comparing the relative usability of different software systems. *Behaviour and Information Technology*, 16(4 / 5):267–278, 1997.

Lohse u.a. 1994 Gerald L. Lohse, Kevin Biolsi, Neff Walker und Henry H. Rueter. A classification of visual representations. *Communications of the ACM*, 37 (12):36–49, Dezember 1994.

Lorho 1977 Bernard Lorho. Semantic attributes processing in the system DELTA. In A. Ershov und Cornelius H. A. Koster. (Hrsg), *Methods of Algorithmic Language Implementation*, Band 47 d. Reihe *Lecture Notes in Computer Science*, S. 21–40. Springer-Verlag, New York–Heidelberg–Berlin, 1977.

Marriott und Meyer 1998 Kim Marriott und Bernd Meyer. The ccmg visual language hierarchy. In Kim Marriott und Bernd Meyer (Hrsg), *Visual Language Theory*, Kapitel 4. Springer, 1998.

Marriott u.a. 1998 Kim Marriott, Bernd Meyer und Kent Wittenbourg. A survey of visual language specification and recognition. In Kim Marriott und Bernd Meyer (Hrsg), *Visual Language Theory*, Kapitel 2. Springer, 1998.

Marriott und Stuckey 1998 Kim Marriott und Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

Matwin und Pietrzykowski 1985 Stanislaw Matwin und Tomasz Pietrzykowski. Prograph: a preliminary report. *Computer Languages*, 10(2):91–126, 1985.

Mauw 1996 S. Mauw. The formalization of Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1643–1657, Juni 1996.

Mayhew 1992 Deborah J. Mayhew. *Principles and Guidelines in Software User Interface Design*. Prentice Hall, Englewood Cliffs, NJ, 1992.

McGuinness 1986 C. McGuinness. Problem representation: The effects of spatial arrays. *Memory & Cognition*, 14:270–280, 1986.

McIntyre und Glinert 1992 D. W. McIntyre und E. P. Glinert. Visual tools for generating iconic programming environments. *IEEE Proc. Workshop on Visual Languages*, S. 162–168, 1992.

McIntyre 1998 David W. McIntyre. Comp.lang.visual - frequently-asked questions list. Newsgroup comp.lang.visual, März 1998.

McMinds 1993 Donald L. McMinds. *Mastering OSF/Motif Widgets*. Addison-Wesley, Reading, MA, USA, zweite Auflage, 1993.

McWhirter und Nutt 1992 Jeffrey D. McWhirter und Gary J. Nutt. A characterization framework for visual languages. In *Proceedings of the IEEE 1992 Workshop on Visual Languages, VL'92*, S. 246–248. IEEE Computer Society Press, 1992.

McWhirter und Nutt 1994 Jeffrey D. McWhirter und Gary J. Nutt. Escalante: An environment for the rapid construction of visual language applications. In Allen L. Ambler und Takayuki Dan Kimura (Hrsg), *Proceedings of the 10th IEEE Symposium on Visual Languages*, S. 15–22, Los Alamitos, CA, USA, October, 4–7 1994. IEEE Computer Society Press.

Mernik u.a. 1999 Marjan Mernik, Mitja Lenič, Enis Avdičauˇsević und Viljem ˇZumer. Multiple attribute grammar inheritance. In D. Parigot und M. Mernik (Hrsg), *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, S. 57–76, Amsterdam, The Netherlands, März 1999. INRIA rocquencourt.

Meyers 1991 Scott Meyers. Difficulties in integrating multiview development systems. *IEEE Software*, 8(1):49–57, 1991.

Minas 1997 M. Minas. Diagram editing with hypergraph parser support. In *Proceedings 1997 IEEE Symposium on Visual Languages*, S. 226–233, Isle of Capri, Italy, September, 23–26 1997. IEEE Computer Society Press.

Minas 1998 M. Minas. Automatically generating environments for dynamic diagram languages. In *Proceedings 1998 IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, September 1998.

Minas und Viehstaedt 1995 M. Minas und G. Viehstaedt. Diagen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *Proceedings 1995 IEEE Symposium on Visual Languages*, S. 203–210, Darmstadt, Germany, September 1995.

Minör 1992 Sten Minör. Interacting with structure-oriented editors. *International Journal of Man-Machine Studies*, 37:399–418, 1992. auch als Technischer Bericht LU-CS-TR91-74 von der Lund University, Schweden erschienen.

Minör und Magnusson 1996 Sten Minör und Boris Magnusson. Using mjølner orm as a structure-based meta environment. In Gerd Szwillus und Lisa Neal (Hrsg), *Structure-based Editors and Environments*, Kapitel 2. Academic Press, 1996.

Müller u.a. 1998 Wolfgang Müller, Christian Geiger und Waldemar Rosenbach. SAM - An animated 3D programming language. In *Proceedings 1998 IEEE Symposium on Visual Languages*, Halifax, Canada, September 1998. IEEE Computer Society Press.

Modugno u.a. 1994 F. Modugno, T. Green und B. Byers. *Visual Programming in a Visual Domain: A Case Study of Cognitive Dimensions*, Kapitel IX. Cambridge University Press, Cambridge, UK., 1994.

Myers 1990 Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Language and Computing*, 1(1):97–123, 1990.

Myers 1994 Brad A. Myers. Program visualization. In John J. Marciniak (Hrsg), *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994. Auch in [Myers 1990].

Myers u.a. 1990 Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish und Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, November 1990.

Nædele und Janneck 1998 Martin Nædele und Jörn W. Janneck. Design patterns in petri net system modeling. In *Proceedings on the 4th IEEE Interna-*

tional Conference on Engineering of Complex Computer Systems, Monterey, CA, USA, August 1998.

Nardi 1993 Bonnie A. Nardi. *A Small Matter of Programming: Perspectives of End-User Computing*. MIT Press, Cambridge, MA, 1993.

Nardi und Zarmer 1993 Bonnie A. Nardi und Craig L. Zarmer. Beyond models and metaphors: Visual formalisms in user interface design. *Journal of Visual Language and Computing*, 4:5–33, 1993.

Nassi und Shneiderman 1973 I. Nassi und B. Shneiderman. Flowchart techniques for structured programming. In *Proc. SIGPLAN'73*, S. 12–26, 1973. Auch in [Glinert 1990b, S. 72-78].

National Instruments 2000 National Instruments. Labview 6i from national instruments. http://www.natinst.com/labview, 2000.

Newton und Browne 1992 P. Newton und J. C. Browne. The CODE 2.0 graphical parallel programming language. In ACM (Hrsg), *Conference proceedings / 1992 International Conference on Supercomputing, July 19–23, 1992, Washington, DC*, S. 167–177, New York, NY 10036, USA, 1992. ACM Press.

Niemann und Bardohl 2000 M. Niemann und R. Bardohl. Tool-based specification of visual languages and graphic editors. In *Tools and Algorithms for the Construction and Analysis of Systems*, Band 1785 d. Reihe *Lecture Notes in Computer Science*, S. 456–470, Berlin, Deutschland, März/April 2000. Springer-Verlag.

Pandey und Burnett 1993 Rajeev K. Pandey und Margaret M. Burnett. Is it easier to write matrix manipulation programs visually or textually? An empirical study. In *Proceedings 1993 IEEE Symposium on Visual Languages*, S. 344–351, Bergen, Norway, August 1993. IEEE Computer Society Press.

Parigot 2000 Didier Parigot. Bibliography on attribute grammars. http://www-rocq.inria.fr/oscar/www/fnc2/AGabstract.html, 2000.

Parigot und Mernik 1999 Didier Parigot und Marjan Mernik (Hrsg). *Second Workshop on Attribute Grammars and their Applications WAGA'99*, Amsterdam, The Netherlands, März 1999. ETAPS'99, INRIA rocquencourt. Satellite event of ETAPS'99.

Parr 1997 Terence John Parr. *Language translation using PCCTS and C++: a reference guide*. Automata Publishing Company, San Jose, CA, USA, Januar 1997.

Petre und Green 1990 M. Petre und T. R. Green. Where to draw the line with text: Some claims by logic designers about graphics in notation. In D. Diaper, D. Gilmore, G. Cockton und B. Shackel (Hrsg), *Human-Computer Interaction – INTERACT'90*, S. 463–468. Elsevier, Amsterdam, 1990.

Petri 1962 C. A. Petri. *Kommunikation mit Automaten*. Dissertation, Universität Bonn, 1962.

Pietrzykowski u.a. 1983 Tomasz Pietrzykowski, Stanislaw Matwin und Tomasz Muldner. The programming language PROGRAPH: Yet another application of graphics. In W. A. Davis (Hrsg), *Proceedings of Graphics Interface '83*, S. 143–145, Edmonton, Alberta, Mai 1983.

Pong und Ng 1983 M. C. Pong und N. Ng. Pigs: A system for programming with interactive graphical support. *Software – Practice and Experience*, 13(9):847–855, 1983.

Poswig 1994 Jörg Poswig. *Visualle Programmiersprachen - Die Realisierung und konzeptionelle Weiterentwicklung eines Prototypen*. Dissertation, Universität Dortmund, 1994.

Poswig 1996 Jörg Poswig. *Visuelle Programmierung: Computerprogramme auf graphischem Wege erstellen*. Hanser Verlag, München, Wien, 1996. ISBN 3-446-17990-9.

Pree 1995 W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, Wokingham, 1995.

Price u.a. 1993 Blaine A. Price, Ronald M. Baecker und Ian S. Small. A principled taxonimy of software visualization. *Journal of Visual Language and Computing*, 4:211–266, 1993.

Prott 1996 Karl-Josef Prott. *Verfahren zur speichereffizienten Attributauswertung in Strukturbäumen*. Dissertation, Universität Paderborn, März 1996.

Ptg 2000 Pattern-based text generator. http://www.upb.de/cs/agkastens/elionline/ptg_toc.html, 2000.

Pugh 1988 William W. Pugh. Incremental computation and the incremental evaluation of functional programs. Technical Report CORNELLCS//TR88-936, Cornell University, Computer Science Department, August 1988.

Quatrani 1998 Terry Quatrani. *Visual Modeling with Rational Rose and UML*. Addison-Wesley, 1998.

Rational Software Corporation u.a. 1997 Rational Software Corporation u.a.. *Unified Modeling Language (version 1.1)*. Rational Software Corporation, September 1997.

Rational Software Corporation u.a. 1999 Rational Software Corporation u.a.. *OMG Unified Modelling Language Specification*, Version 1.3 Auflage, Juni 1999. http://www.rational.com/uml/resources/documentation/.

Rekers und Schürr 1996 J. Rekers und A. Schürr. A graph based framework

for the implementation of visual environments. In *Proceedings 1996 IEEE Symposium on Visual Languages*, S. 148–155, Boulder, Colorado, September 1996.

Repenning und Fahlen 1993 Alex Repenning und Lennart E. Fahlen. Agentsheets: A tool for building domain-oriented visual programming environments. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, Demonstrations, S. 142–143, 1993.

Repenning und Sumner 1995 Alexander Repenning und Tamara Sumner. Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer*, 28(3):17–25, 1995.

Reps 1982 Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *9th ACM Symp. on Principles of Progr. Languages*, S. 169–176. ACM press, Albuquerque, NM, Januar 1982.

Reps u.a. 1983 Thomas Reps, Tim Teitelbaum und Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Progr. Languages and Systems*, 5(3):449–477, Juli 1983.

Reps und Teitelbaum 1989a Thomas W. Reps und Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-based Editors*. Springer Verlag, 1989.

Reps und Teitelbaum 1989b Thomas W. Reps und Tim Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer Verlag, zritte Auflage, 1989.

Robbins u.a. 1998 J. Robbins, D. Hilbert und D. Redmiles. Extending design environments to software architecture design. *Automated Software Engineering*, 5(3):261–290, Juli 1998.

Roussel u.a. 1994 Gilles Roussel, Didier Parigot und Martin Jourdan. Coupling Evaluators for Attribute Coupled Grammars. In Peter A. Fritzson (Hrsg), *5th Int. Conf. on Compiler Construction (CC' 94)*, Band 786 d. Reihe *Lecture Notes in Computer Science*, S. 52–67, Edinburgh, April 1994. Springer-Verlag.

Rumbaugh u.a. 1991 James Rumbaugh, Michael Blaha, William Premerlani, Fredrick Eddy und William Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, 1991.

Rumbaugh u.a. 1999 James Rumbaugh, Ivar Jacobson und Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, 1999.

Sajeev 1998 A. S. M. Sajeev. Parallel programming using visual patterns. Technischer Bericht 98/01, Monash University, Clayton, Melbourne, Victoria 3168, Australia, März 1998.

Sannella und Borning 1992 Michael Sannella und Alan Borning. Multi-garnet: Integrating multi-way constraints with garnet. Technical Report TR-92-07-01, University of Washington, Department of Computer Science and Engineering, Juli 1992.

Saraiva u.a. 1996 J. Saraiva, M. F. Kuiper und S. D. Swierstra. Effective function cache management for incremental attribute evaluation. Technischer Bericht UU-CS-1996-50, Utrecht University, NL, 1996.

Saraiva und Swierstra 1999 Joao Saraiva und Doaitse Swierstra. Generic attribute grammars. In D. Parigot und M. Mernik (Hrsg), *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, S. 185–204, Amsterdam, The Netherlands, März 1999. INRIA rocquencourt.

Sayles u.a. 1994 Jonathan S. Sayles, Steve Karlen, Peter Molchan und Gary Bilodeau. *GUI-based Design and Development for Client/Server Applications, Using PowerBuilder, SQLWindows, Visual Basic, PARTS-Workbench*. John Wiley & Sons, 1994.

Schiffer 1998 Stefan Schiffer. *Visuelle Programmierung: Grundlagen und Einsatzmöglichkeiten*. Addison Wesley, Bonn [u.a.], 1998.

Schmidt und Schindler 2000 Carsten Schmidt und Christian Schindler. Musterbasierte Generierung von Struktur-Editoren für visuelle Sprachen. Diplomarbeit, Universität Paderborn, Deutschland, Januar 2000.

Sedgewick 1992 Robert Sedgewick. *Algorithms in C++ <dt>*. Addison Wesley, 1992. deutsche Übersetzung durch Heribert Bieling.

Selker und Koved 1988 Ted Selker und Larry Koved. Elements of visual language. In *1988 IEEE Proceedings of the Workshop on Visual Languages*, S. 38–44. IEEE Computer Society Press, 1988.

Shizuki u.a. 1999 Buntarou Shizuki, Masahi Toyoda, Etsuya Shibayama und Shin Takahashi. Visual patterns + multi-focus fisheye view: An automatic scalable visualization technique of data-flow visual program execution. In *Proceedings 1999 IEEE Symposium on Visual Languages*, S. 270–277, Tokyo, Japan, September, 13–16 1999. IEEE Computer Society Press.

Shneiderman 1983 B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, S. 57–68, August 1983.

Shu 1988 Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold, 1988.

Singh und Chignell 1992 G. Singh und M. H. Chignell. Components of the visual computer. *The Visual Computer*, 9:115–142, 1992.

Sloane 1995 Anthony M. Sloane. An Evaluation of an Automatically Genera-

ted Compiler. In *ACM Transactions on Programming Languages and Systems*, Band 17, S. 691–703, September 1995.

Stankovic und Zhang 1997 Nenad Stankovic und Kang Zhang. Towards visual development of message-passing programs. In *Proceedings 1997 IEEE Symposium on Visual Languages*, S. 144–151, Isle of Capri, Italy, September, 23–26 1997. IEEE Computer Society Press.

Staufer 1987 Michael Staufer. *Piktogramme für Computer*. de Gruyter, 1987.

Steinman und Carver 1996 Scott B. Steinman und Kevin G. Carver. *Visual Programming with Prograph CPX*. Manning Publications Co., 1996.

Sutherland 1963 Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings AFIPS Spring Joint Computer Conference*, Band 23, S. 329–346, Detroit, Michigan, Mai 1963. Auch in [Glinert 1990b, S.198–215].

Szwillus 1996 Gerd Szwillus. User interface definition based on graphical structure editors. In Gerd Szwillus und Lisa Neal (Hrsg), *Structure-Based Editors and Environments*, Kapitel 8, S. 253–275. Academic Press, 1996.

Tan und Cai 1995 Hung-Knoon Tan und Wentong Cai. VPEcons: A visual constructor for parallel programming. In *Proceedings of the 1st IEEE Int. Conf. on Algorithms and Architectures for Parallel Computing*, Australia, April 1995. IEEE Computer Society Press.

Toyoda u.a. 1997 Masahi Toyoda, Buntarou Shizuki, Shin Takahashi, Satoshi Matsuoka und Etsuya Shibayama. Supporting design patterns in a visual parallel data-flow programming environment. In *Proceedings 1997 IEEE Symposium on Visual Languages*, S. 76–83, Isle of Capri, Italy, September, 23–26 1997. IEEE Computer Society Press.

Tripp 1988 Leonard L. Tripp. A survey of graphical notations for program design - an update. *ACM SIGSOFT Software Engineering Notes*, 13(4):39–44, 1988. Auch in [Glinert 1990b, S. 60–71].

Usher und Jackson 1998 Michelle Usher und David Jackson. A concurrent visual language based on petri nets. In *Proceedings of the 1998 IEEE Symposium on Visual Languages*, Nova Scotia, Canada, September 1998. IEEE Computer Society Press.

Vlissides und Linton 1990 John M. Vlissides und Mark. A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, Juli 1990.

Vose und Williams 1986 G. M. Vose und G. Williams. LabVIEW: Laboratory virtual instrument engineering workbench. *BYTE*, S. 84–92, September 1986.

Waite 1986 W. M. Waite. Generator for attribute grammars – abstract data type. Technischer Bericht 219, GMD Karlsruhe, 1986.

Waite und Goos 1984 W. M. Waite und G. Goos. *Compiler Construction*. Springer Verlag, New York, NY, 1984.

Waite und Kadhim 1995 William M. Waite und Basim M. Kadhim. A general property storage module. Technischer Bericht CU-CS-786-95, University of Colorado at Boulder, September 1995.

Welch 1997 Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, Zweite Auflage, 1997. ISBN 0-13-616830-2.

Whitley 1997 K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Language and Computing*, 8(1): 109–142, Februar 1997.

Wieringa 1998 Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4): 459–527, Dezember 1998.

Wilhelm 1974 Reinhard Wilhelm. Code-optimierung mittels attributierter transformationsgrammatiken. In D. Siefkes (Hrsg), *GI 4. Jahrestagung*, Band 26 d. Reihe *Lecture Notes in Computer Science*, S. 257–266, New York–Heidelberg–Berlin, Oktober 1974. Springer-Verlag. See also: Bericht 7408, Abteilung Mathematik, Tech. University München (1974).

Wilhelm 1991 Reinhard Wilhelm. Attribute reevaluation in OPTRAN. In Henk Alblas und Bořivoj Melichar (Hrsg), *Attribute Grammars, Applications and Systems*, Band 545 d. Reihe *Lecture Notes in Computer Science*. Springer-Verlag, New York–Heidelberg–Berlin, Juni 1991.

Wirtz 1994 G. Wirtz. Modularization and process replication in a visual parallel programming language. In Allen L. Ambler und Takayuki Dan Kimura (Hrsg), *Proceedings of the Symposium on Visual Languages*, S. 72–79, Los Alamitos, CA, USA, Oktober 1994. IEEE Computer Society Press.

Wirtz 1995 G. Wirtz. Modularization, re-use and testing for parallel message-passing programs. In Hesham El-Rewini und Bruce D. Shriver (Hrsg), *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 2: Software Technology*, S. 299–308, Los Alamitos, CA, USA, Januar 1995. IEEE Computer Society Press.

Yang u.a. 1997 Sherry Yang, Margaret M. Burnett, Elyon DeKoven und Moshé Zloof. Representation design benchmarks: A design-time aid for vpl navigable static representations. *Journal of Visual Language and Computing*, 8: 563–599, 1997.

Yeh 1983 Dashing Yeh. On incremental evaluation of ordered attributed grammars. *BIT*, 23:308–320, 1983.

Yeh und Kastens 1988 Dashing Yeh und Uwe Kastens. Improvements of an incremental evaluation algorithm for ordered attributed grammars. *ACM SIGPLAN Notices*, 23(12):45–50, Dezember 1988.

Zhang und Zhang 1998 Da-Qian Zhang und Kang Zhang. VisPro: A visual language generation toolset. In *Proceedings 1998 IEEE Symposium on Visual Languages*, S. 195–202, Halifax, Nova Scotia, September 1998.

Zloof 1975 M. M. Zloof. Query-by-example. In *Proceedings of the AFIPS National Computer Conference*, S. 431–437, Mai 1975.