Generierung von Strukturediteren für anspruchsvolle visuelle Sprachen

Dissertation

Schriftliche Arbeit zur Erlangung des akademischen Grades
„Doktor der Naturwissenschaften“
an der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von

Carsten Schmidt

Paderborn, Januar 2006

Datum der mündlichen Prüfung:
24. Februar 2006

Gutachter:
Prof. Dr. Uwe Kastens, Universität Paderborn
Prof. Dr.-Ing. Mark Minas, Universität der Bundeswehr München
Prof. Dr. Gerd Szwillus, Universität Paderborn

Inhaltsverzeichnis

1 Einleitung

In der Geschichte der Programmiermethodik gab es mindestens zwei „Quantensprünge“. Einer war der Übergang vom Maschinencode zu höheren Programmiersprachen. Ein anderer war der Übergang zur objektorientierten Programmierung, wodurch die Kapselung und Wiederverwendbarkeit von Programmteilen signifikant verbessert wurde. Aber wie sieht der nächste Quantensprung aus?

In diesem Zusammenhang wird derzeit oft eine Vision diskutiert, die manchmal *Language Oriented Programming* (LOP) genannt wird [12, 14]. *Language Oriented Programming* versteht sich als Überbegriff für verschiedene Strömungen wie *Intentional Programming*, MDA, *Generative Programming* etc. Die Grundidee ist, dass künftig spezialisierte, ineinandergreifende Spezialsprachen verwendet werden, um eine bestimmte Aufgabe umzusetzen. Welche Spezialsprachen verwendet und kombiniert werden, hängt von der Aufgabe ab. Dieser Ansatz könnte die Verwendung homogener Universal-Programmiersprachen in weiten Bereichen ablösen. Durch Benutzung anwendungsorientierter Spezialsprachen lassen sich Lösungen auf viel höherem Niveau mit den Begriffen des Problembereichs beschreiben, anstatt sie in die Begriffswelt allgemeiner Programmiersprachen übertragen zu müssen. Auf diese Weise wird die Programmierung einfacher und intuitiver.

Die Ideen hinter LOP gab es zwar schon lange, fristeten aber aufgrund mangelnder Nachfrage und fehlender methodischer Grundlagen ein Schattendasein. Die heutigen Anforderungen an die Softwaretechnik könnten dies ändern.

- Softwaresysteme werden immer komplizierter und die Entwicklungszyklen immer kürzer, so dass herkömmliche Methoden wie objektorientierte Programmierung evtl. bald an ihre Grenzen stoßen.
- Auch Endanwender wollen immer komplexere Probleme lösen und benötigen dazu einfach handhabbare, prägnante Ausdrucksmittel.

Die Lösung beider Probleme könnten anwendungsspezifische, visuelle Sprachen sein. Dieser Trend ist sowohl in der Softwaretechnik als auch in der Endanwenderprogrammierung klar zu erkennen. In der Softwaretechnik werden bereits jetzt visuelle Modellierungs-, Spezifikations- und Programmiersprachen wie UML, SDL, EN-3, LabVIEW, Matlab/Simulink usw. verwendet. Im Forschungsgebiet der Endanwenderprogrammierung hat sich bereits herausgestellt, dass durch intuitive Benutzungsschnittstellen, ausdrucksstarke Spezialsprachen und grafische Repräsentationen auch Nicht-Experten komplexe Aufgaben bewältigen können.

Um diese Ideen weiter voranzutreiben ist es entscheidend, dass sich visuelle Umgebungen möglichst einfach realisieren lassen, denn nur dann amortisiert sich das Kosten/Nutzen-Verhältnis einer visuellen Spezialsprache. Hierbei sind folgende Punkte besonders zu beachten.

- Im Allgemeinen wird für eine neue Spezialsprache nicht nur ein Übersetzer benötigt, sondern auch eine integrierte Entwicklungsumgebung, die den Anwender bei der Benutzung der Sprache unterstützt. Java-Programmierer nutzen beispielsweise Systeme wie IntelliJ IDEA, das die Sprache „kennt“ und so z.B. Refactoring oder Navigation im Programmcode wirkungsvoll unterstützen kann.
- Besonders bei visuellen Sprachen ist es wichtig, maßgeschneiderte Editoren bereitzustellen, die den Anwender bei der Programmerstellung leiten und den Aufwand für das Layout reduzieren.

Ziele Diese Arbeit soll einen Beitrag leisten, die Realisierung von Entwicklungsumgebungen für visuelle Sprachen zu vereinfachen, so dass mit vertretbarem Aufwand neue visuelle Sprachen entwickelt werden können. Dazu wurde das Werkzeugsystem DEViL (*Development Environment for Visual Languages*) entwickelt, das selbst auf dem LOP-Ansatz basiert. Mit Hilfe von anwendungsspezifischen Spezialsprachen und Spezifikationsmodulen können visuelle Strukturediteren auf hohem Niveau spezifiziert werden. Hieraus generiert DEViL automatisch eine vollständige Entwicklungsumgebung, die Editoren, Analysatoren und Codegeneratoren enthält.

DEViL basiert auf Vorarbeiten, die in Zusammenarbeit mit Matthias Jung und Christian Schindler geleistet wurden [29]. Zusammen haben wir das VL-Eli System entwickelt, das ebenfalls Implementierungen visueller Sprachen generieren kann. Im Vergleich zu verwandten Systemen hat unser Ansatz folgende besondere Merkmale.

- Unser Ansatz basiert stark auf der Baumstruktur visueller Sprachen, wohingegen viele andere Ansätze Modellierungen nutzen, die der „part-of“ Beziehung wenig Bedeutung beimessen.
- Die Spezifikation visueller Darstellungen basiert auf so genannten „visuellen Mustern“. Visuelle Muster sind Abstraktionen elementarer Darstellungskonzepte in visuellen Sprachen. Durch die Kombination visueller Muster lassen sich visuelle Sprachen einfach, intuitiv und auf hohem Niveau spezifizieren. Basierend auf dem in Muster-Implementierungen gekapselten Expertenwissen lassen sich aus den Spezifikationen benutzerfreundliche, einfach bedienbare Sprachimplementierungen generieren.

Das Ziel dieser Arbeit ist es, unter Beibehaltung der erfolgversprechenden Konzepte des VL-Eli Systems ein Werkzeugsystem zu entwickeln, das mächtig und flexibel genug ist, um auch große, anspruchsvolle visuelle Sprachen umsetzen zu können. Hierzu habe ich neue Konzepte entwickelt oder sie aus anderen Ansätzen übernommen und in ein Gesamtkonzept integriert. Dabei habe ich darauf geachtet, dass trotz des Zuwachses an Flexibilität kleine, einfache Sprachen weiterhin einfach spezifizierbar bleiben.

Methodische Beiträge Besonders zur Unterstützung unterschiedlich strukturierter Sichten ist es sinnvoll, verschiedene Rollen der abstrakten Programmrepräsentation zu differenzieren. Die abstrakte Programmrepräsentation dient (1) zur Verkörperung der semantisch relevanten Programminformation, (2) zur Speicherung des Informationsgehalts visueller Darstellungen und (3) als Grundlage zur Spezifikation visueller Darstellungen. In dieser Arbeit wird gezeigt, dass in anspruchsvollen Strukturediteren im Allgemeinen mehrere leicht unterschiedliche strukturelle Abstraktionen für diese drei Rollen benötigt werden. Das sich aus dieser Differenzierung ergebende Konzept ermöglicht nicht nur eine konsistente Implementierung, sondern erleichtert es auch, andere Ansätze in den Gesamtzusammenhang einzuordnen und so besser in Beziehung setzen zu können. Obwohl es in der Informatik in vielen Fällen wichtig ist Gemeinsamkeiten zu finden, ist es in diesem Fall wichtig zu differenzieren.

Eine wesentliche Eigenschaft von DEViL und dessen Vorgänger VL-Eli ist die Nutzung der Baumstruktur als wichtiges Spezifikationsmittel. Problematisch ist allerdings, dass die initial verwendeten traditionellen kontextfreien Grammatiken bestimmten Anforderungen visueller Strukturediteren nicht gerecht

werden. In dieser Arbeit habe ich daher eine spezielle Spezifikationssprache für Strukturen entwickelt, die die Vorteile von kontextfreien Grammatiken und objektorientierten Modellierungsmethoden vereint. Die entworfene Spezifikationssprache eignet sich sowohl für visuelle als auch für textuelle Repräsentationen.

Im Bereich der visuellen Muster leistet diese Arbeit drei Beiträge: Erstens verbessert die Differenzierung der abstrakten Strukturen die Anwendbarkeit visueller Muster, so dass die Vorteile des Ansatzes besonders deutlich hervortreten. Zweitens wird das formale Modell weiterentwickelt, das die Kombinierbarkeit visueller Muster beschreibt. Drittens werden neue Muster-Varianten vorgestellt, die die Bibliothek der Muster-Implementierungen ergänzen und zeigen, dass auch komplexe visuelle Muster durch den hier vorgestellten Ansatz wiederverwendbar gekapselt werden können. Beispielsweise die durch das Matrix-Muster realisierte Benutzerfreundlichkeit beim Editieren von Matrizen (siehe Abschnitt 4.2.6) ist mit keinem mir bekannten System vergleichbar.

Ein wichtiger Beitrag der Arbeit sind auch die teils grafischen Spezialsprachen, mit denen große Teile der visuellen Repräsentation einfach und intuitiv spezifiziert werden können. Vor allem die so genannten Generischen Zeichnungen (siehe Abschnitt 4.4) halte ich für einen wichtigen Schritt hin zu benutzerfreundlichen Werkzeugsystemen zur Sprachimplementierung. Hier zeigen sich die Stärken des LOP-Ansatzes besonders deutlich.

Neben diesen methodischen Beiträgen soll diese Arbeit auch zeigen, dass der Ansatz, Implementierungen für visuelle Sprachen aus Spezifikationen zu generieren, auch für größere, anspruchsvolle Sprachen praktisch umsetzbar ist. Dazu habe ich den Generator relativ ausführlich und methodisch fundiert evaluiert. Besonders hervorzuheben ist hierbei ein Feldversuch, mit dem im Rahmen der studentischen Projektgruppe „Generierung von Web-Anwendungen aus visuellen Spezifikationen“ die Praxistauglichkeit des Ansatzes evaluiert wurde (siehe Abschnitt 5.2.3 und 5.2.9).

Struktur der Arbeit Im zweiten Kapitel werden die Grundlagen dieser Arbeit dargestellt. Neben den Eigenschaften visueller Sprachen und Strukturediteren im Allgemeinen werden auch verwandte Ansätze vorgestellt, die für diese Arbeit eine besondere Bedeutung haben. Insbesondere wird genauer auf das VL-Eli System eingegangen, das sozusagen der Vorgänger von DEViL ist.

Im dritten Kapitel wird auf die Unterscheidung zwischen semantischer und

editierbarer Struktur eingegangen. Die semantische Struktur beschreibt den semantisch relevanten Informationsgehalt eines visuellen Programmes, die editierbare Struktur beschreibt den Informationsgehalt einer visuellen Sicht und bildet die Grundlage für deren Implementierung. Der Kern dieses Kapitels beschreibt die Methode, um die Strukturen zu definieren und den Zusammenhang beider Strukturen herzustellen.

Im vierten Kapitel wird beschrieben, wie sich visuelle Sichten basierend auf visuellen Mustern spezifizieren lassen. Hierzu wird eine dritte wichtige strukturelle Abstraktion eingeführt, die so genannte Repräsentationsstruktur. Basierend auf dieser Struktur wird in DEViL die grafische Darstellung durch attributierte Grammatiken spezifiziert. Hierauf baut die Methode der visuellen Muster auf. Neu entworfene Spezialsprachen zur Vereinfachung der Spezifikation runden das Kapitel ab.

Im fünften Kapitel werden die Konzepte von DEViL evaluiert. Es wird gezeigt, dass die neu entwickelten Konzepte keinen Zusatzaufwand bei der Spezifikation einfacher Sprachen verursachen. Die Ergebnisse der Projektgruppe „Generierung von Web-Anwendungen aus visuellen Spezifikationen“ und die Umsetzung von Teilen der *Unified Modeling Language* zeigen, dass der Ansatz auch für große, anspruchsvolle Sprachen geeignet ist. Neben der Evaluation aus Sprachentwickler-Sicht wird DEViL auch aus Sprachanwender-Sicht evaluiert. Hier wird gezeigt, dass die generierten Editoren einfach und intuitiv zu bedienen sind und den Sprachbenutzer gut beim Layout unterstützen.

Das sechste Kapitel fasst schließlich das Wesentliche der Arbeit zusammen. Außerdem enthält das Kapitel einen Ausblick auf zukünftige Erweiterungen und lohnende ergänzende Forschungsarbeiten. Hier wird insbesondere ein visuelles Frontend für DEViL vorgestellt, das derzeit entwickelt wird und den Lernaufwand zur Anwendung von DEViL nochmals drastisch reduzieren kann.

Jedes Kapitel außer Einleitung und Schlussbemerkungen enthält am Ende einen Abschnitt zu verwandten Arbeiten. Hier werden Querbeziehungen zu den Grundlagen und anderen Ansätzen herausgestellt und die Beiträge dieser Arbeit von anderen abgegrenzt.

Hinweise für den Leser Bezeichner aus Programmen und Spezifikationen sind in `Schreibmaschinenschrift` gesetzt. Begriffsdefinitionen, englische Bezeichnungen und Hervorhebungen sind *kursiv* dargestellt.

Die maskuline Form wird in dieser Arbeit auch dann verwendet, wenn sowohl männliche als auch weibliche Personen gemeint sind. In diesem Sinne sind „der Anwender“ sowie „der Sprachentwickler“ ausdrücklich geschlechtsneutral gemeint.

2 Grundlagen

Inhalt

In diesem Kapitel stelle ich die Grundlagen meiner Arbeit dar. Dazu gebe ich in den ersten beiden Abschnitten einen Überblick über das Gebiet der visuellen Sprachen und stelle einige Beispiele vor. Hierdurch möchte ich dem Leser visuelle Sprachen näher bringen und deren Praxisrelevanz unterstreichen.

In den verbleibenden Abschnitten geht es um die Implementierung visueller Sprachen, genauer um die Implementierung visueller Strukturediteren. Dazu gehe in Abschnitt 2.3 auf allgemeine Eigenschaften von Strukturediteren ein und beschreibe wichtige orthogonale Klassifikationsdimensionen. Ferner stelle ich ein funktionales Modell zur Implementierung von Strukturediteren vor. Dieses Modell wird sowohl im weiteren Verlauf des Kapitels als auch in den folgenden Teilen der Arbeit als Erklärungsgrundlage dienen.

In den Abschnitten 2.4 und 2.5 werden andere Systeme zur Sprachimplementierung vorgestellt, bei denen zumindest Teilkonzepte mit DEViL vergleichbar sind. Das VL-Eli System – sozusagen der Vorgänger von DEViL – wird hierbei besonders ausführlich behandelt.

2.1 Visuelle Sprachen

2.1.1 Definition visueller Sprachen

Es ist schwer, den Begriff „visuelle Sprache“ angemessen zu definieren. Im Laufe der Zeit haben etliche Autoren Definitionen beigetragen, die allerdings oft sehr diffus sind oder Interpretationen zulassen, die der Intention widersprechen. Schiffer [50] gibt einen guten Überblick über die Problematik.

Schiffer selbst trägt die folgende, relativ prägnante Definition des Begriffes „visuelle Sprache“ bei:

> *Visuell* ist die Bezeichnung für jene Eigenschaft eines Objekts, durch die 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 wesentliche Rolle spielt hier der Begriff „visuell“. Schiffer hebt hervor, dass textueller Quelltext primär durch das verbale Wahrnehmungssystem interpretiert wird. Visuelle Sprachen hingegen umfassen Elemente wie Farben, Formen, Verbindungen, Überlagerungen, Berührungen usw., deren Interpretation über das visuelle Wahrnehmungssystem erfolgt. Zwar könnte man auch eine visuelle Repräsentation verbal durch eine Bildbeschreibung vermitteln, aber zur Interpretation dieser Bildbeschreibung muss der Zuhörer das beschriebene Bild im Geiste aufbauen.

Das Wesentliche einer visuellen Sprache ist nach Schiffer die visuelle Syntax, also das Regelwerk zur Bildung von Ausdrücken aus Grundsymbolen. Der etwas unglücklich gewählte Begriff „visueller Semantik“ bezeichnet in Schiffers Terminologie visuelle Notationen, die Laufzeitzustände der Objekte repräsentieren.

Wichtig ist Schiffer auch, dynamische Zeichengebung in die Definition einzuschließen. Bei dynamischer Zeichengebung werden Informationen nicht statisch, sondern durch einen zeitlichen Ablauf repräsentiert. Häufig wird diese Darstellungsform im Zusammenhang mit dem Programmierparadigma „Programming by Example“ verwendet. In solchen Systemen führt der Programmierer einen Algorithmus vor, der dann durch das System generalisiert wird.

Eine der vielen alternativen Definitionen stammt von Myers [38]:

> Visuelle Programmierung (VP) bezieht sich auf jedes System, das dem Anwender erlaubt, ein Programm auf zwei- (oder mehr-) dimensionale Weise zu spezifizieren. Obwohl das eine sehr breite Definition ist, werden konventionelle textuelle Sprachen nicht als zweidimensional betrachtet, weil Compiler oder Interpretierer sie als lange, eindimensionale Ströme verarbeiten.

Myers Definition hebt die räumlichen Dimensionen in visuellen Sprachen hervor, die tatsächlich eine bedeutende Rolle spielen. Andere visuelle Aspekte wie Sinnbilder oder Farben werden in dieser Definition jedoch nicht angemessen berücksichtigt.

Eine interessante Frage ist, ob textuelle Sprachen wie Python [22], bei denen die Einrückung der Zeilen Semantik trägt, bereits visuell sind. Die Definitionen lassen diese Interpretation zu und meiner Ansicht nach ist dies tatsächlich ein visuelle Eigenschaft der Sprache. Allerdings wird man Python kaum als visuelle Sprache bezeichnen wollen. Unter dem Begriff „visuelle Sprache“

ist also im Allgemeinen eine Sprache zu verstehen, in der *wesentliche* Teile der Repräsentation auf visuellen Konzepten basieren. Nachfolgend nenne ich solche Sprachen *echt visuell*. Im Gegensatz dazu verstehe ich unter der Menge der visuellen Sprachen im Allgemeinen eine *Obermenge* der textuellen Sprachen. Wenn eine konkrete Sprache als visuell bezeichnet wird, ist normalerweise „echt visuell“ gemeint. Das Verhältnis zwischen visuellen und textuellen Sprachen entspricht dem Verhältnis zwischen reellen und natürlichen Zahlen. Es ist zwar nicht falsch aber irreführend zu sagen, dass 2 eine reelle Zahl ist.

2.1.2 Implementierung visueller Sprachen

Prinzipiell muss zwischen einer visuellen Sprache und einer Implementierung dieser Sprache unterschieden werden. Während dies bei textuellen Sprachen im Allgemeinen nicht schwer fällt, wird diese Unterscheidung bei visuellen Sprachen leicht verwischt. Das liegt daran, dass visuelle Sprachen häufig sehr eng mit ihrer Implementierung verwoben sind.

Es gibt zwei verschiedene Ansätze zur Sprachimplementierung, die sich darin unterscheiden, wie Programme erstellt werden. Beim parsing-basierten Ansatz werden Programme mit einem Universaleditor (entweder Text- oder Grafikeditor) erstellt. Zur Verarbeitung der Programme werden sie parsiert, wobei aus der konkreten Repräsentation die abstrakte Struktur des Programms abgeleitet wird. Demgegenüber beinhaltet die Implementierung beim zweiten Ansatz einen maßgeschneiderten Strukturedítor zur Programmkonstruktion. Hier wendet der Benutzer strukturierte Editieroperationen an, mit denen direkt die abstrakte Struktur des Programms manipuliert wird.

Beide Ansätze wurden sowohl im Kontext textueller als auch visueller Sprachen vielfach umgesetzt. Bei textuellen Sprachen konnten sich Strukturedítoren nicht durchsetzen. Das liegt vermutlich daran, dass Nutzer beim parsing-basierten Ansatz ihren gewohnten Texteditor benutzen können und dass sie nicht von Anfang an zu syntaktischer Korrektheit gezwungen werden, also freier arbeiten können. Des Weiteren können versierte Computerbenutzer Programme vermutlich schneller tippen, als sie strukturiert einzugeben.

Bei visuellen Sprachen sind zumindest im kommerziellen Bereich derzeit Strukturedítoren vorherrschend. Das liegt vermutlich daran, dass einerseits das Parsieren visueller Ausdrücke schwieriger ist und es andererseits sehr mühsam sein kann, visuelle Programme mit einem Universaleditor zu erstel-

len, der keine adäquaten sprachspezifischen Hilfsmittel anbietet. Im visuellen Bereich ist allerdings noch offen, welcher Ansatz sich endgültig durchsetzen wird.

Ein Mittelweg zwischen beiden Extremen sind so genannte „intelligente“ Editoren (z.B. [5, S.477]). Intelligente Editoren sind sprachspezifische Umgebungen, die im Kern auf einem Universaleditor basieren. Die Programmfragmente werden bereits während der Eingabe soweit wie möglich analysiert, so dass der Editor den Benutzer bereits während der Programmentwicklung sowohl in syntaktischen als auch in semantischen Fragen wirkungsvoll unterstützen kann. Ein kommerziell sehr erfolgreiches System dieser Art für eine textuelle Sprache ist IntelliJ IDEA der Firma *JetBrains*. IntelliJ IDEA ist eine Entwicklungsumgebung für Java, die einen freien Texteditor und einen inkrementellen Parser beinhaltet, so dass syntaktische und semantische Fehler bereits während der Eingabe angezeigt werden können. Das Programm kann auch Programmteile automatisch formatieren, neue Klassen strukturiert erstellen und bietet Hilfsmittel zur Refaktorisierung. Intelligente Editoren für visuelle Sprachen werden z.B. von DiaGen II generiert (siehe Abschnitt 2.5.6). Mit solchen Editoren können visuelle Programme frei editiert werden, wobei aber zumindest sprachspezifische Grundelemente zur Verfügung gestellt werden. Da die Teilrepräsentationen bereits während der Konstruktion analysiert werden, können schon früh syntaktische Fehler aufgezeigt und Operationen zur automatischen Formatierung angeboten werden. Die Editoren lassen sich auch um strukturierte Editieroperationen erweitern.

2.1.3 Vor- und Nachteile visueller Sprachen und Strukturediteren

Derzeit werden sowohl visuelle als auch textuelle Sprachen eingesetzt und es ist nicht absehbar, dass die (echt) visuellen Sprachen die textuellen verdrängen. Es stellt sich also die Frage nach den Vor- und Nachteilen visueller Sprachen. Separat hiervon stellt sich ebenfalls die Frage nach den Vor- und Nachteilen von Strukturediteren, denn wie oben ausgeführt wird auch dieses Thema kontrovers diskutiert.

Natürlich kann ich im Rahmen dieser Arbeit nicht auf alle Aspekte dieser Fragestellung eingehen, aber es sind zumindest die wichtigsten Argumente zu nennen, denn nur so können die entwickelten Werkzeuge gezielt auf die Stärken des gewählten Ansatzes abgestimmt werden. Viele der nachfolgend genannten Argumente werden von Schiffer [50] ausführlich diskutiert.

Vor- und Nachteile visueller Repräsentationen Sind visuelle Sprachen besser als textuelle? Klar ist, dass unspezifische Behauptungen wie „ein Bild sagt mehr als tausend Worte“ hier nicht weiterhelfen. Meine Antwort lautet trotzdem: Ja! Zur Begründung sei an meine Interpretation des Begriffs „visuelle Sprache“ erinnert. Da visuelle Sprachen eine echte Obermenge von textuellen Sprachen sind, bestehen beim Sprachentwurf mehr Möglichkeiten. Wird für einen bestimmten Anwendungszweck die „perfekte“ Sprache entworfen, ist es sehr unwahrscheinlich, dass sie „zufällig“ in die kleinere Klasse der textuellen Sprachen fällt. Streng genommen müsste ich meine Aussage also so formulieren: „Für jeden Anwendungsfall gibt es eine visuelle Sprache, die mindestens so gut ist wie jede textuelle“.

Ich sehe folgende konkrete Vorteile visueller Repräsentationen gegenüber textuellen:

- Bestimmte Strukturen lassen sich übersichtlicher darstellen.
- Bestimmte Repräsentation erlauben Layoutfreiheiten, durch die menschlichen Betrachtern informelle Zusatzinformationen übermittelt werden können.
- Manchmal lassen sich quantitative Eigenschaften wirkungsvoll visuell darstellen.
- Visuelle Sprachen ermöglichen es, die Symbolik des Anwendungsbereichs zu übernehmen.
- Visuelle Repräsentationen können dem Benutzer besser verdeutlichen, wie er ein Programm editieren kann.
- Die Layoutfreiheiten mancher Repräsentationsformen fördern die inkrementelle Entwicklung visueller Ausdrücke.

Ein Beispiel für Repräsentationsformen, die die Übersichtlichkeit fördern, sind visuelle Graph-Repräsentationen. Aus diesem Grund werden Graphen in der Informatik seit langem visuell dargestellt. Diese Darstellungsform eignet sich besonders dann, wenn ein einzelner Knoten nicht allzu viele Kanten besitzt und die Kanten häufig in beide Richtungen verfolgt werden müssen. Ein weiteres Beispiel für übersichtliche Darstellungen sind visuelle Schachtelungen, die Teil-Ganzes-Beziehungen besser ausdrücken können als Klammern in textuellen Sprachen.

Auch für den zweiten Spiegelpunkt sind Graphen ein gutes Beispiel. Wenn sich die Knoten des Graphen frei platzieren lassen, können dadurch z.B. eng zusammengehörende Teile oder Symmetrien hervorgehoben werden. Von elektronischen Schaltplänen ist bekannt, dass derartige informelle Lesehilfen dort häufig und wirkungsvoll eingesetzt werden.

Quantitative Größen lassen sich manchmal durch Koordinaten, Winkel oder andere grafische Attribute repräsentieren. Dies führt häufig zu sehr prägnanten Repräsentationen mit einem engen Bezug zum Anwendungsbereich. Ein gutes Beispiel hierfür sind Generische Vektorgrafik-Zeichnungen (siehe Abschnitt 4.4.1).

Ein Beispiel für den vierten Spiegelpunkt ist LabVIEW (siehe Abschnitt 2.2.3). Dort wird das Paradigma „Bauelemente und Leitungen“ verwendet, um Anwendern die Programmierung im Bereich der Mess- und Regeltechnik zu erleichtern.

Da in visuellen Darstellungen die Struktur eines Programms besser erkennbar ist als in textuellen, helfen sie dem Benutzer dadurch, mit der Repräsentation zu interagieren. Der Benutzer kann die Grundobjekte der Repräsentation besser unterscheiden. Zusätzlich lassen sich in visuellen Repräsentationen gut Hinweise auf Interaktionsmöglichkeiten integrieren. Beispiele sind Sinnbilder zum Auf- und Zuklappen von Teilrepräsentationen oder „Griffe“ zum Anfassen visueller Objekte.

Der letzte Punkt drückt aus, dass z.B. in UML-Klassendiagrammen Klassen an beliebiger Stelle gezeichnet werden können. Das ist insbesondere beim Entwurf auf Papier wichtig, denn dort können schon gezeichnete Klassen nicht nachträglich verschoben werden. Visuelle Sprachen eignen sich daher für „Brainstorming-Sitzungen“ und inkrementellen Programmentwurf.

Visuelle Konstrukte haben im Vergleich zu textuellen Repräsentationen aber auch Nachteile.

- Visuelle Repräsentationen brauchen im Allgemeinen mehr Platz als textuelle Kodierungen. [1]
- Ist eine visuelle Repräsentation in mehr als einer Dimension größer als der sichtbare Bildschirmbereich, so verliert man darin schnell die Übersicht.

[1] Man kann dieses Faktum auch positiv formulieren: „Visuelle Repräsentationen sind nicht so gedrängt wie ihre textuellen Gegenstücke.“

- Die Konstruktion visueller Ausdrücke mit Universaleditoren kann aufwändig sein.
- Die Wartung eines visuellen Layouts kann selbst bei Verwendung eines Strukturediteors aufwändig sein.
- Da Bezeichner in visuellen Programmrepräsentationen häufig sparsamer eingesetzt werden, verliert der Betrachter wichtige informelle Hinweise zum Programmverständnis.

Dass visuelle Repräsentationen mehr Platz benötigen als textuelle liegt vor allem daran, dass die räumliche Anordnung formelle oder informelle Semantik trägt und daher nicht beliebig gewählt werden kann, so dass Verschnitt entsteht. Der Nachteil besteht in erster Linie darin, dass somit weniger Information in den sichtbaren Bildschirmbereich passt, wodurch die Darstellung schneller unübersichtlich wird. Die Unübersichtlichkeit wird teilweise durch den zweiten Spiegelpunkt verstärkt.

Der zweite Nachteil, die Unübersichtlichkeit mehrdimensionaler Repräsentationen, wird häufig dadurch vermieden, indem man einen größeren visuellen Ausdruck in mehrere kleine unterteilt.

Der dritte Punkt, der Aufwand zur Konstruktion visueller Ausdrücke, lässt sich weitgehend durch den Einsatz von Strukturediteoren vermeiden, wobei man dann allerdings die Nachteile von Strukturediteoren in Kauf nehmen muss.

Der vierte Punkt, der Aufwand zur Wartung des Layouts, ergibt sich daraus, dass in visuellen Darstellungen informelle Zusatzinformationen durch das Layout ausgedrückt werden können. Diese Informationen muss der Sprachbenutzer nach Änderungen ebenfalls aktualisieren, was auch bei Benutzung von Strukturediteoren Zusatzaufwand erfordert.

Der letzte Punkt stellt fest, dass Bezeichner in Programmen sehr wichtige Sekundärinformationen für den Betrachter sind. Man stelle sich nur ein Java-Programm vor, in dem alle Bezeichner durch Zahlen ersetzt wurden. Daher können auch visuelle Sprachen nicht ganz auf textuelle Bestandteile verzichten.

Vor- und Nachteile von Strukturediteoren Sprachspezifische Systeme (d.h. Strukturediteoren und „intelligente“ Editoren) haben im Vergleich zu sprachunspezifischen Systemen folgende Vorteile.

- Sie unterstützen den Benutzer bei der Erstellung syntaktisch und semantisch korrekter Programme und
- sie unterstützen den Benutzer beim Layout.

Prinzipiell haben sie nur den Nachteil, dass der Benutzer nicht mehr seinen Lieblings-Editor benutzen kann, sondern sich auf ein neues System einstellen muss. Je stärker sich die Bedienfunktionen der Editoren unterscheiden, desto stärker kann sich der Wechsel auf die Produktivität des Benutzers auswirken. Das Problem lässt sich aber durch Anpassbarkeit oder Standardisierung von Bedienfunktionen abmildern.

Der Vergleich zwischen Strukturediteren und „intelligenten“ freien Editoren fällt nicht ganz so einseitig aus. Strukturediteren haben hier folgende Vorteile.

- Sie geben Hilfestellung bei der Programmkonstruktion, so dass der Nutzer nicht alle Details der Syntax kennen muss.
- Sie helfen beim Umgang mit komplexen Strukturen.

Besonders der zweite Punkt ist beachtenswert und die wesentliche Motivation dieser Arbeit. Der Umgang mit komplexen Strukturen kann mit verschiedenen Mechanismen erleichtert werden. Es können z.B. Interaktionsmechanismen vorgesehen werden, durch die sich Teilstrukturen je nach Bedarf ein- und ausblenden lassen. Eine weitere häufig angewendete Technik ist die Implementierung mehrerer Sichten auf die Struktur. Durch Übersichts- und Detaildarstellungen lässt sich das Grobverständnis des Programms fördern. Durch parallele Sichten lassen sich bestimmte Spezifikationsaspekte hervorheben und andere ausblenden. Ein sehr gutes Beispiel für diesen Ansatz ist LabVIEW (siehe Abschnitt 2.2.3), wo die Benutzungsschnittstelle und die Funktionalität als zwei verschiedene Sichten auf das Programm betrachtet werden. Des Weiteren können Strukturen manchmal automatisch erzeugt oder angepasst werden. Das automatische Erzeugen von Strukturen ist z.B. beim Aufruf von Funktionen sinnvoll, wo bereits Platzhalter für die aktuellen Parameter vorgegeben werden können, die der Benutzer nur noch „ausfüllen“ muss.

Ein Strukturediteor hat im Vergleich zu „intelligenten“ freien Editoren aber auch Nachteile.

- Da Strukturediteren syntaktisch korrekte Programme erzwingen, können sich Nutzer in ihrem Arbeitsfluss gestört fühlen.

- Vor allem visuelle Strukturediteren garantieren im Allgemeinen nicht, dass die konkreten Darstellungen eindeutig interpretierbar sind.
- Zumindest bei textuellen Sprachen lassen sich Programme mit freien Editoren schneller erstellen.

Der erste Punkt stellt fest, dass der Sprachbenutzer zur Verwirklichung bestimmter Änderungen vorplanen muss, wie diese durch strukturierte Editieroperationen umgesetzt werden können. Dies beansprucht kognitive Kapazitäten, die dem Nutzer nicht mehr für semantische Überlegungen zur Verfügung stehen.

Nach dem zweiten Spiegelpunkt garantieren Strukturediteren im Allgemeinen nicht die eindeutige Interpretierbarkeit konkreter Repräsentationen. Beispielsweise Beschriftungen an Linien lassen sich manchmal nicht eindeutig zuordnen, wenn mehrere Linien in der Nähe sind. Der Grund dafür ist, dass Strukturediteren nur die Abbildung der abstrakten Struktur auf die konkrete Repräsentation berechnen und die verwendeten Layoutkonzepte nur schwer die Bijektivität der Abbildung sicherstellen können.

Der dritte Punkt ist darin begründet, dass zumindest professionelle Computerbenutzer häufig sehr schnell tippen können. Dies ist bei Endanwendern nicht unbedingt der Fall.

Zusammenfassung Zusammenfassend lässt sich sagen, dass visuelle Sprachen für bestimmte Anwendungen durchaus gewinnbringend eingesetzt werden können und dass Strukturediteren besonders für visuelle Sprachen sinnvoll erscheinen. Strukturediteren unterstützen den Sprachentwickler bei der Konstruktion und beim Layout visueller Programme und andererseits verbessern visuelle Repräsentationen die Intuitivität von Strukturediteren.

Beim Entwurf eines Generators für visuelle Strukturediteren muss besonders darauf geachtet werden, dass die oben genannten Vorteile voll ausgeschöpft werden können. Der Spezifikationsmechanismus für die visuelle Darstellung sollte mächtig genug sein, um gute Repräsentationsformen wählen und die Symbolik des Anwendungsbereichs übernehmen zu können. Die Editoren sollten intuitiv bedienbar sein, um den Benutzer optimal bei der Programmkonstruktion und beim Layout zu unterstützen. Schließlich sollte der Generator Spezifikationsmechanismen für strukturelle Kopplungen und parallele Sichten bieten, so dass komplexe Strukturen möglichst einfach erstellt und bearbeitet werden können.

2.2 Beispiele für visuelle Sprachen

Nachfolgend sollen einige Beispiele für visuelle Sprachen vorgestellt werden, um dem Leser einen Eindruck vom Anwendungsgebiet zu geben und die Praxisrelevanz des Themas zu belegen. Anhand dieser Sprachen werden in den folgenden Kapiteln auch die Spezifikationsmechanismen von DEViL erklärt.

2.2.1 Unified Modeling Language

Die *Unified Modeling Language* (UML) [41, 26] hat bereits eine längere Entwicklung hinter sich und liegt derzeit in der Version 2.0 vor. Sie hat sich als Standard zur Modellierung von Anforderungen und Entwürfen in der Softwaretechnik durchgesetzt. Die Weiterentwicklung von UML verfolgt derzeit das Ziel, vollständige Systemmodellierungen zu ermöglichen, so dass sich hieraus direkt Implementierungen generieren lassen.

UML 2 besitzt 13 Diagrammtypen, von denen sechs zur Modellierung der Systemstruktur und sieben zur Modellierung des Systemverhaltens dienen. Für diese Arbeit sind besonders die Klassendiagramme und Zustandsdiagramme relevant.

Klassendiagramme werden zur Modellierung von Strukturen eines zu entwerfenden oder abzubildenden Systems verwendet. Die zentralen Sprachkonstrukte sind Klassen, Attribute, Operationen sowie Vererbungs- und Assoziationsbeziehungen. Ein Beispiel für ein Klassendiagramm ist in Abbildung 2.1 dargestellt.

Klassendiagramme dienen nicht nur als Anwendungsbeispiel, sondern sie sind auch ein wichtiges Kalkül zur Definition abstrakter Strukturen. Beispielsweise ist die abstrakte Syntax von UML selbst durch eine vereinfachte Form von UML-Klassendiagrammen spezifiziert. Man spricht in diesem Zusammenhang auch vom so genannten Metamodell.

Zustandsdiagramme modellieren das Verhalten bestimmter Programmobjekte basierend auf dem Konzept der endlichen Automaten. Die Zustandsautomaten, die in UML eingesetzt werden, gehen auf Harel [21] zurück, der unter anderem hierarchisch strukturierte Zustände eingeführt hat. Die zentralen Sprachkonstrukte sind Zustände, Transitionen, Trigger und Guards. Ein Beispiel ist in Abbildung 2.2 zu sehen.

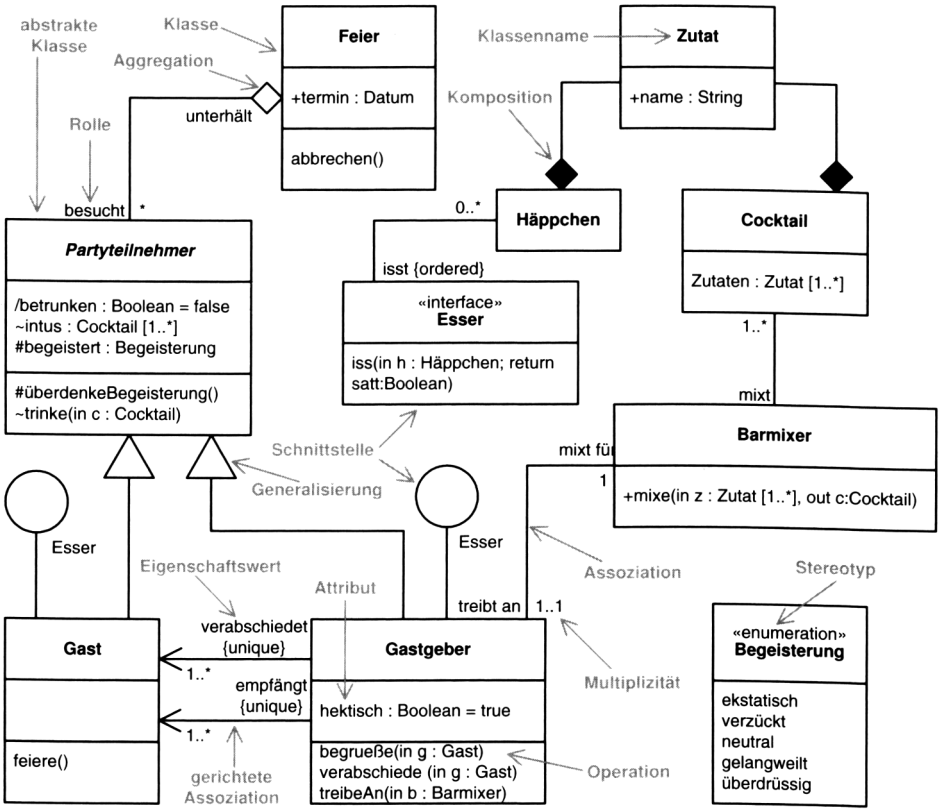


Abbildung 2.1: Sprachelemente in UML-Klassendiagrammen (aus [26])

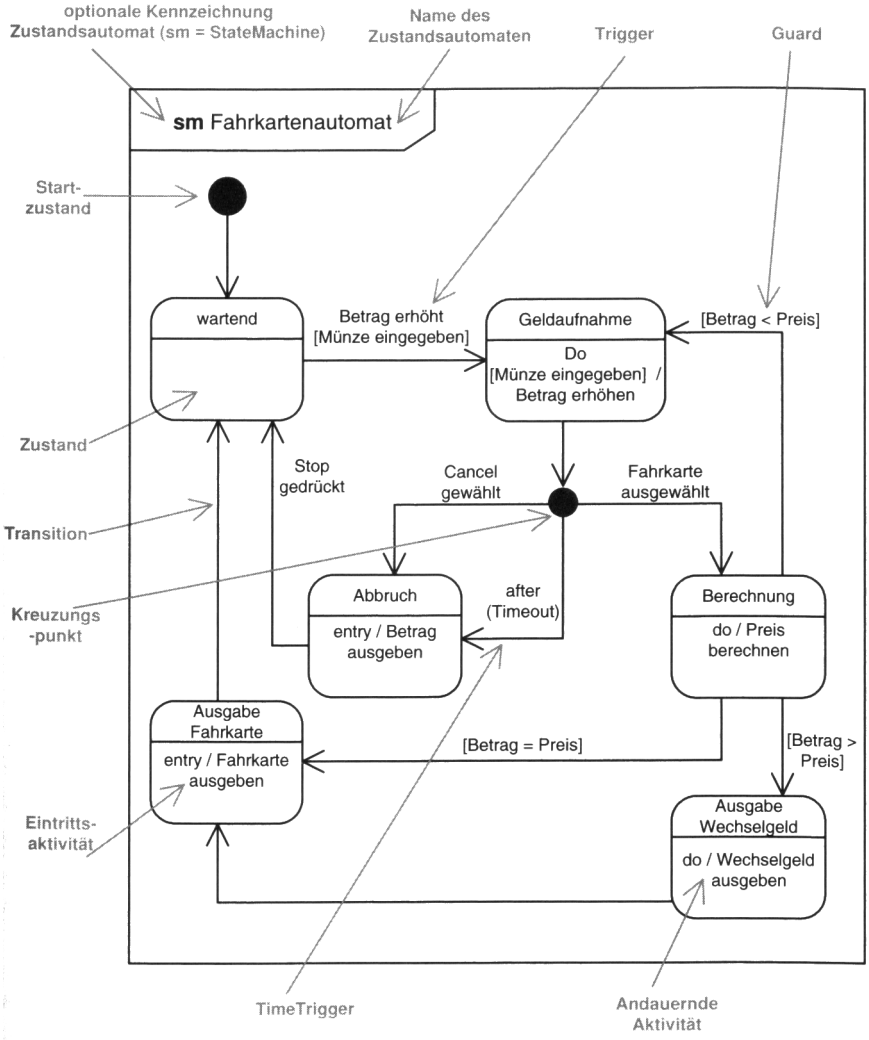


Abbildung 2.2: Sprachelemente in UML-Zustandsdiagrammen (aus [26])

Relevanz im Kontext dieser Arbeit Als Anwendungsbeispiel für meinen Ansatz ist UML vor allem deshalb interessant, weil teilweise eine große Distanz zwischen der abstrakten Struktur eines UML-Modells und dessen konkreter Repräsentation zu überbrücken ist. Die Klassen eines Modells können auf mehrere Klassendiagramme verteilt sein und dieselbe Klasse kann durchaus mehrfach in den Diagrammen auftreten. Teilweise zeigen UML-Diagramme unterschiedliche Sichten auf gleiche oder überlappende Strukturen, wodurch weitere Redundanz entsteht. Ein wichtiges Thema dieser Arbeit ist der Umgang mit dieser Redundanz (siehe Kapitel 3).

Wie bereits erwähnt sind UML-Klassendiagramme auch als Kalkül zur Syntax-Definition, also auf Ebene des Generator-Entwurfs interessant. Das in dieser Arbeit entwickelte Syntax-Kalkül ist eng mit UML vewandt, worauf in Abschnitt 3.1.2 näher eingegangen wird.

2.2.2 Nassi-Shneiderman Diagramme

Nassi-Shneiderman Diagramme, auch Struktogramme genannt, wurden 1973 von Nassi und Shneiderman [40] eingeführt, um imperative Programme strukturiert beschreiben zu können. Nassi-Shneiderman Diagramme sind nach DIN 66261 standardisiert und besitzen einen hohen Bekanntheitsgrad. Für die Softwaretechnik sind sie allerdings kaum von Bedeutung.

Abbildung 2.3 zeigt ein Beispiel für ein Nassi-Shneiderman Diagramm. Programmanweisungen werden als Rechteck und Sequenzen als vertikale Folge dargestellt. Die Details elementarer Anweisungen wie Zuweisungen sind in textueller Form in dem Rechteck enthalten. Bei Fallunterscheidungen werden die alternativen Zweige nebeneinander angeordnet und die Bedingung der Fallunterscheidung wird zwischen schräg verlaufenden Linien oberhalb der Zweige dargestellt. Schleifen sind Rechtecke, die den Schleifenrumpf als eingebettetes Rechteck enthalten.

Relevanz im Kontext dieser Arbeit Nassi-Shneiderman Diagramme repräsentieren eine Klasse visueller Sprachen, die hauptsächlich auf Schachtelung und Aneinanderreihung von Konstrukten basieren. Sie bilden damit ein Gegengewicht zu graphartigen visuellen Sprachen wie UML-Klassendiagrammen oder UML-Zustandsdiagrammen.

Da Nassi-Shneiderman Diagramme relativ einfach sind, dienen sie in dieser Arbeit auch als einführendes Beispiel für die Spezifikation visueller Darstel-

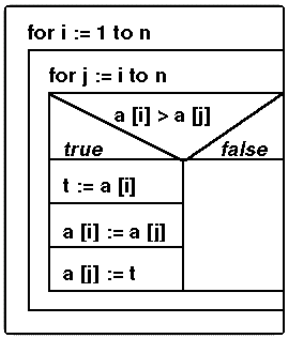


Abbildung 2.3: Ein Nassi-Shneiderman Diagramm (aus [50])

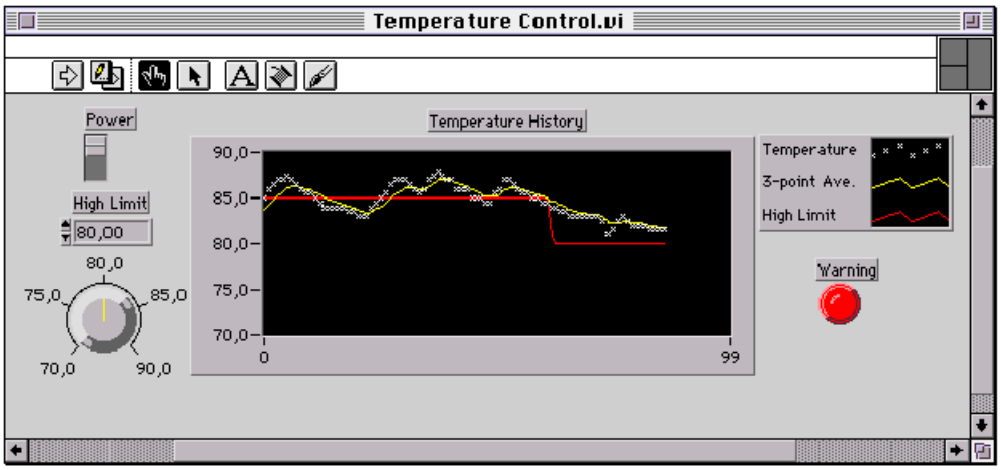
lungen (siehe Abschnitt 4.1.3) und insbesondere für das Konzept der Generischen Zeichnungen (siehe Abschnitt 4.4).

2.2.3 LabVIEW

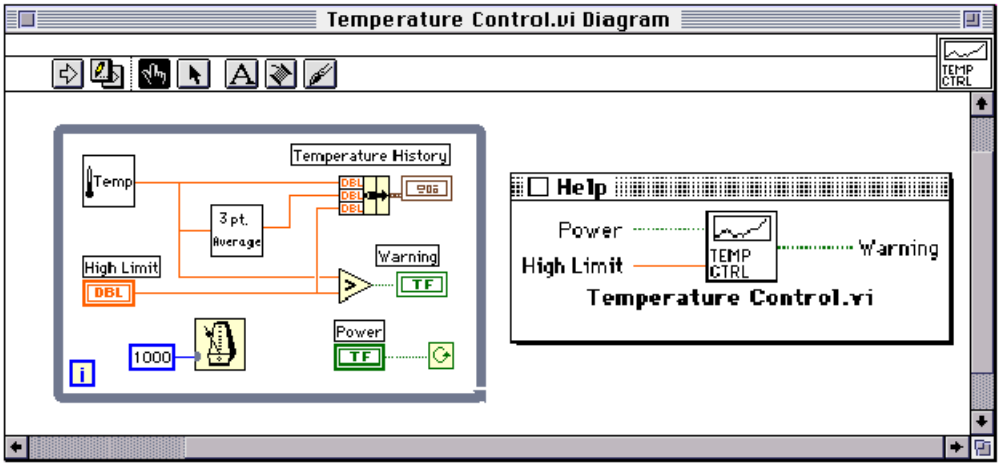
LabVIEW [64] ist eine kommerziell eingesetzte Datenflusssprache, mit der Programme für Mess- und Steuerungssysteme realisiert werden können. Das System eignet sich u.a. für die Überwachung, Auswertung und Visualisierung von Messdaten sowie zur Simulation elektronischer Schaltungen.

Abbildung 2.4 zeigt ein LabVIEW-Programm, das aus zwei Teilspezifikationen besteht. Die Objekte in diesen Spezifikationen werden „virtuelle Instrumente“ genannt. Sie ähneln in Aussehen und Funktionsweise realen elektrischen Instrumenten. Der obere Teil, in dem die Instrumente wie physikalische Schalter oder Anzeigeinstrumente dargestellt sind, spezifiziert die Benutzungsschnittstelle des Programms. Der untere Teil, in dem die Verdrahtung der Instrumente und zusätzliche Verknüpfungs- und Kontrollkonstrukte enthalten sind, spezifiziert dessen Wirkung.

Diese strikte Trennung zwischen Oberflächen-Spezifikation und Spezifikation der Funktionalität ist eine Besonderheit von LabVIEW. Da die LabVIEW-Implementierung auf einem Strukturedítor basiert, kann die Konstruktion und Wartung solcher Programme wirkungsvoll unterstützt werden. Beide Teildiagramme lassen sich als Sichten auf einer gemeinsamen Struktur betrachten. Wenn z.B. ein neues Instrument eingefügt wird, erscheint es in beiden Sichten und muss jeweils in den dort vorhandenen Kontext integriert werden. Dank des Strukturedítors sind die Sichten also automatisch zueinander konsistent.



Frontplatte mit Verbinder



Blockschaltbild mit Hilfefenster für Verbinder

Abbildung 2.4: Bildschirmfoto des LabVIEW Systems (aus [50])

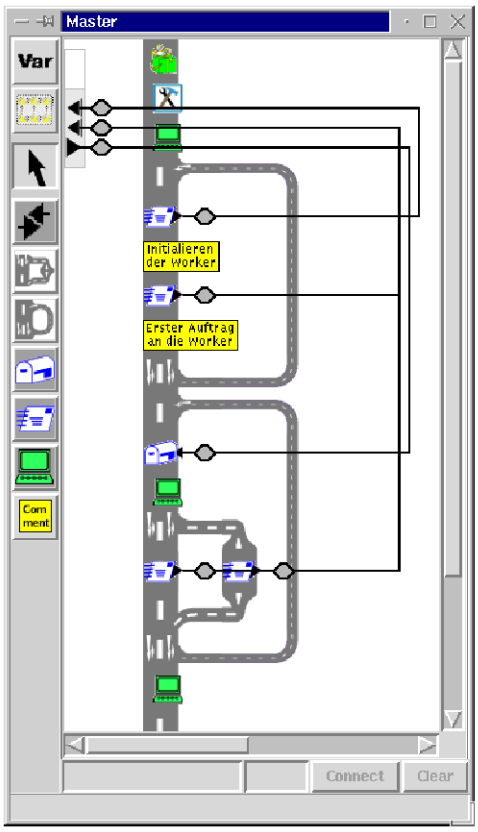
Relevanz im Kontext dieser Arbeit LabVIEW ist ein gutes Beispiel für Sprachen, die die Symbolik eines Anwendungsbereichs übernehmen, um die Benutzung der Sprache einfacher und intuitiver zu machen. Ferner demonstriert LabVIEW gut den Nutzen paralleler Sichten auf eine gemeinsame Struktur, um verschiedene Spezifikationsaspekte zu separieren. Schließlich wird LabVIEW kommerziell eingesetzt und zeigt damit, dass es erfolgreiche visuelle Programmiersprachen gibt.

2.2.4 Streets

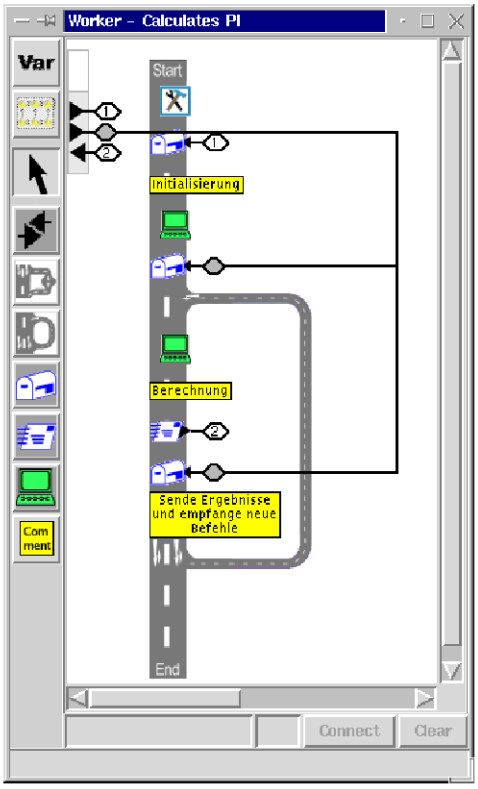
Streets ist eine visuelle Programmiersprache, die Anfängern die Implementierung paralleler Algorithmen erleichtern soll. Sie wurde 1998 im Rahmen einer Projektgruppe an der Universität Paderborn entwickelt [30]. Sie basiert auf imperativer Programmierung und asynchroner Nachrichtenübermittlung zwischen Prozessen.

Abbildung 2.5 zeigt ein Programm zur Berechnung der Zahl π, das auf dem Master-Worker-Prinzip beruht. Der Kontrollfluss wird durch eine Straße visualisiert, die prinzipiell von oben nach unten verläuft. Fallunterscheidungen werden durch Verzweigungen der Straße symbolisiert. Programmschleifen sind an Abbiegungen und zurückführenden Straßenteilen zu erkennen. Sinnbilder wie Briefe, Briefkästen oder Computer visualisieren verschiedene Arten von imperativen Anweisungen. Die beiden erstgenannten Sinnbilder stehen für Anweisungen zum Senden bzw. Empfangen von Nachrichten. Hier werden Linienverbindungen benutzt, um den jeweiligen Kommunikationspartner zu spezifizieren.

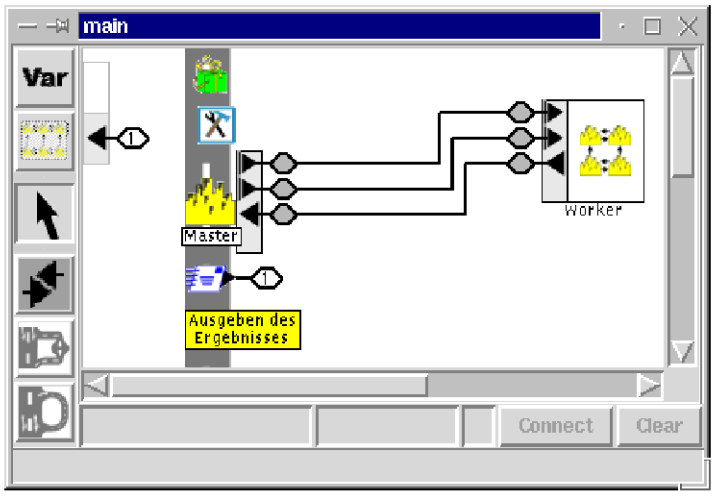
Relevanz im Kontext dieser Arbeit Auch *Streets* basiert ähnlich wie LabVIEW auf Sinnbildern und komplexen visuellen Metaphern. Ein wesentliches Merkmal und die besondere Herausforderung von Streets ist die Metapher der Straße. Am Beispiel von Streets wird in Abschnitt 4.4.2 gezeigt, wie sich auch diese Repräsentation basierend auf Generischen Kachel-Zeichnungen in DEViL spezifizieren lässt.



(a) Programm des Master-Prozesses



(b) Programm der Worker-Prozesse



(c) Hauptprogramm

Abbildung 2.5: Ein Programm der visuellen Programmiersprache *Streets*

2.3 Strukturediteren

Im weitesten Sinne ist jedes Programm, mit dem eine anwendungsspezifische Datenstruktur manipuliert werden kann, ein Strukturediteur. Ein Beispiel für einen Strukturediteur in diesem Sinne ist eine Oberfläche zur Textverarbeitung. Die manipulierte Datenstruktur enthält z.B. Absätze, Überschriften, Zeichen und Formatierungen.

Die anwendungsspezifische Datenstruktur eines Strukturediteurs wird vom Benutzer anhand einer konkreten Repräsentation manipuliert. In einer Oberfläche zur Textverarbeitung kann es z.B. eine Sicht geben, die (fast) so aussieht wie die ausgedruckte Version des Dokuments. Genauso gut könnte eine Textverarbeitung aber auch eine Repräsentation benutzen, die nicht versucht, das Layout des Ergebnisses nachzuahmen, sondern die die Struktur des Textes hervorhebt. Oft können Strukturediteuren die anwendungsspezifische Datenstruktur auf mehrere Arten darstellen. In einigen Textverarbeitungs-Oberflächen kann man z.B. auswählen, ob man mit einer „what you see is what you get“-Sicht oder einer strukturierten Sicht arbeiten möchte.

Im Allgemeinen kann man Sichten zu verschiedenen Zwecken einsetzen. Das oben beschriebene Paar von Sichten zeigt die Struktur auf gleicher Detailebene, hebt aber unterschiedliche Aspekte hervor. Paare solcher Sichten werden nachfolgend *parallele Sichten* genannt. Des Weiteren werden häufig Übersichts- und Detailsichten in Strukturediteuren verwendet. In Übersichtsdarstellungen wird die Grobstruktur gezeigt, in der bestimmte Objekte nur als Sinnbilder repräsentiert sind. Detailsichten stellen ein einzelnes Objekt mit all seinen Details dar.

Die anwendungsspezifische Datenstruktur eines Strukturediteurs kann man als abstrakte Sprache mit bestimmter Syntax und bestimmter Semantik auffassen. Die Semantik der Datenstruktur für eine Textverarbeitung definiert z.B., wie ein in dieser Sprache formuliertes Dokument zu formatieren und auszudrucken ist.

Im Kontext dieser Arbeit sind vor allem Strukturediteuren relevant, die Sprachen im engeren Sinne implementieren. Da entsprechende Systeme häufig zur Softwareentwicklung dienen, nenne ich die in Strukturediteuren konstruierten Ausdrücke nachfolgend *Programme*. Dadurch sollen aber Strukturediteuren für Spezifikations- und Abfragesprachen nicht ausgeschlossen werden.

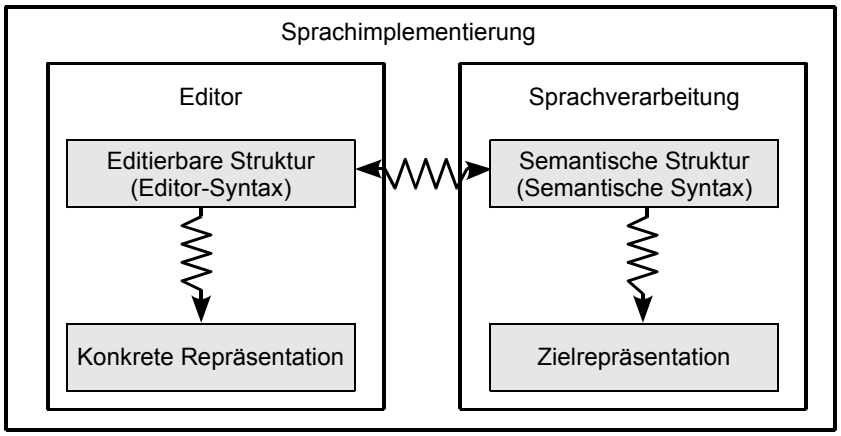


Abbildung 2.6: Grundmodell einer Sprachimplementierung

2.3.1 Ein funktionales Modell zur Sprachimplementierung

Zu einer vollständigen Sprachimplementierung gehört nicht nur ein Strukturediteor, sondern auch ein System zur Weiterverarbeitung der konstruierten Programme. Um auf die Eigenschaften von Strukturediteoren und Sprachimplementierungen genauer eingehen zu können, möchte ich das in Abbildung 2.6 gezeigte funktionale Modell zugrunde legen. Hiernach besteht eine Sprachimplementierung aus einem Editor und einer Sprachverarbeitungskomponente. Der Benutzer des Strukturediteors manipuliert anhand einer konkreten Repräsentation die so genannte *editierbare Struktur*, während als Grundlage der Sprachverarbeitung die so genannte *semantische Struktur* verwendet wird.

Wichtig ist hier, dass es sich bei den Pfeilen nach unten um *Funktionen* handelt, d.h. die konkrete Repräsentation und die Zielrepräsentation lässt sich vollständig und eindeutig aus der editierbaren Struktur bzw. der semantischen Struktur berechnen. Natürlich können tatsächlichen Implementierungen auch andere Modelle zugrunde liegen, aber jeder Strukturediteor kann anhand dieses Modells interpretiert werden. Der Vorteil ist, dass hier klar zwischen Informationsgehalt und Darstellung unterschieden wird.

Die semantische Struktur Die semantische Struktur repräsentiert den semantisch relevanten Informationsgehalt des Programms. Sie abstrahiert von allen Informationen, die lediglich der Programmdarstellung dienen. Die semantische Struktur besitzt eine eigene Syntax und auf Basis dieser Syntax ist die Semantik von Programmen definiert.

Sprachimplementierungen besitzen in der Regel Analysatoren, Übersetzer oder Simulatoren, die auf der semantischen Struktur arbeiten. Aus diesem Grund sollte die semantische Struktur möglichst redundanzfrei sein und nur Informationen enthalten, die für die Bedeutung des Programms relevant sind. Die semantische Struktur lässt sich des Weiteren als Austauschformat zwischen verschiedenen Implementierungen der gleichen Sprache verwenden.

Im Fall von UML wird die Syntax der semantischen Struktur „abstrakte Syntax“ genannt und ist durch das UML-Metamodell definiert. Sowohl die Definition der UML-Semantik als auch das UML-Austauschformat XMI basieren hierauf. In dieser Arbeit nenne ich die Syntax der semantischen Struktur abweichend von der UML-Terminologie *semantische Syntax*.

Die editierbare Struktur Die editierbare Struktur repräsentiert den Informationsgehalt einer visuellen Programmrepräsentation und enthält damit u.a. auch alle Layoutentscheidungen des Benutzers. Dabei sollte die Information trotzdem möglichst abstrakt und redundanzfrei sein. Beispielsweise enthielte die editierbare Struktur eines Klassendiagramms die Information, dass eine Klasse an bestimmten Koordinaten dargestellt werden soll. Die Information, dass die Klasse durch ein Rechteck dargestellt wird, muss in der editierbaren Struktur nicht enthalten sein, da sie redundant ist. Das hat den Vorteil, dass sich dieses Darstellungsdetail allein durch eine Modifikation der Abbildungsvorschrift auf die konkrete Repräsentation ändern lässt und die editierbare Struktur unberührt bleibt. Die editierbare Struktur lässt sich als abstraktes Austauschformat für konkrete Repräsentationen interpretieren.

In dieser Arbeit nenne ich die Syntax der editierbaren Struktur abkürzend *Editor-Syntax*.

Diskussion Dieses einfache Modell erlaubt bereits eine relativ differenzierte Diskussion von Strukturediteor-Eigenschaften. Es basiert auf einer strikten Trennung zwischen „Informationsgehalt“ und daraus ableitbaren Repräsentationen. Ein wichtiger Vorteil dieser Sichtweise ist, dass strukturelle Operationen vollkommen unabhängig von den daraus abgeleiteten Repräsentationen betrachtet werden können. In diesem Modell ist z.B. die Implementierung eines *Undo*-Mechanismus vollkommen unabhängig von der Realisierung der konkreten Repräsentation.

Im Gegensatz zu den Pfeilen nach unten ist der Zusammenhang zwischen editierbarer Struktur und semantischer Struktur weitaus komplizierter. Die

editierbare Struktur kann Layoutinformationen enthalten, die nicht aus der semantischen Struktur ableitbar sind. Anders herum ist es nicht ausgeschlossen, dass die editierbare Struktur nur einem Ausschnitt der semantischen Struktur entspricht. Dies ist der Fall, wenn es mehrere Sichten gibt, die Teile der semantischen Struktur auf unterschiedliche Weise darstellen. In diesem Fall lässt sich also auch nicht aus der editierbaren Struktur die semantische ableiten. Die Strukturen existieren also gewissermaßen „gleichberechtigt" nebeneinander.

Auf den genauen Zusammenhang beider Strukturen wird in Kapitel 3 eingegangen. An dieser Stelle sei aber bemerkt, dass sich semantische Struktur und editierbare Struktur in der Praxis nicht sehr stark voneinander unterscheiden müssen. Bei Editoren mit vollkommen automatischem Layout können beide Strukturen zusammenfallen und bei einfachen Editoren mit Layoutfreiheiten kann die editierbare Struktur häufig als Spiegelbild der semantischen Struktur betrachtet werden, das um zusätzliche Layoutinformationen angereichert ist. In einem UML-Klassendiagramm enthielten die Klassen-Objekte der editierbaren Struktur z.B. benutzerdefinierte Layoutinformationen wie z.B. Positionsangaben, während die Klassen-Objekte in der semantischen Struktur keine Layoutinformationen besitzen. Aus diesem Grund unterscheiden einige Ansätze (z.B. VL-Eli) die beiden Abstraktionen nicht. In einem solchen Fall nenne ich die Vereinigung der beiden Strukturen „abstrakte Struktur" und die Syntax entsprechend „abstrakte Syntax".

Erwähnt werden soll an dieser Stelle auch, dass die editierbare Struktur naturgemäß als Grundlage zur Spezifikation konkreter Repräsentationen dient und daher idealerweise möglichst genau der Struktur der Darstellung entsprechen sollte. Das bedeutet, dass es zu jedem grafischen Objekt der konkreten Repräsentation genau ein entsprechendes strukturelles Objekt in der editierbaren Struktur geben sollte, dass die Schachtelung von Objekten in der konkreten Repräsentation durch eine dafür geeignete Relation in der editierbaren Struktur repräsentiert sein sollte usw. Die Syntax der editierbaren Struktur beschreibt nach dieser Sichtweise die Grundbausteine und Kombinationsmöglichkeiten der grafischen Objekte in konkreten Repräsentationen. Je nachdem, ob z.B. ein Rechteck und eine Zeichenkette in der konkreten Repräsentation untrennbar miteinander verbunden sind oder ob sie erst zusammengefügt werden müssen, enthält die editierbare Struktur ein oder zwei Objekte dafür. Auch aus dieser Forderung können sich Unterschiede zwischen editierbarer und semantischer Struktur ergeben.

2.3.2 Klassifikation von Strukturediteren

Auf Basis des oben beschriebenen funktionalen Modells lassen sich Sprachimplementierungen in den folgenden sechs unabhängigen Dimensionen klassifizieren.

1. Art der konkreten Repräsentation
2. Layoutfreiheiten
3. Interaktionsmechanismen
4. Niveau der editierbaren Struktur
5. Schärfe der Editor-Syntax
6. Anzahl der Repräsentanten von semantischen Objekten

Nachfolgend werden die Varianten in den einzelnen Dimensionen separat diskutiert.

Dimension 1: Art der konkreten Repräsentation

Die konkrete Repräsentation der editierbaren Struktur kann sehr unterschiedlich sein. Häufig unterscheidet man zwischen visuellen und textuellen Repräsentationen. Textuelle Repräsentationen sind dabei nach der in Abschnitt 2.1.1 dargestellten Interpretation visueller Sprachen ein Spezialfall visueller Repräsentationen.

Es gibt noch weitere häufig vorkommende „Spezialfälle“. Solche Spezialfälle zeichnen sich dadurch aus, dass die visuelle Darstellung auf einem sehr spezifischen Konzept basiert und dieses auch in der gesamten Sicht durchgehalten wird. Weitere häufig auftretende Spezialfälle sind

- einfache visuelle Graph-Darstellungen,
- Sichten mit Dialogelementen und
- Sichten mit auf- und zuklappbaren Bäumen, wie sie häufig in Dateimanagern verwendet werden.

All diese „eingeschränkten“ visuellen Sichten werden häufig verwendet und sind sehr weit verbreitet. Das liegt vermutlich daran, dass entsprechende Implementierungskomponenten bzw. fertige Systeme dafür zur Verfügung stehen und einfach verwendet werden können.

Genau wie es bestimmte eingeschränkte Darstellungsarten gibt, lassen sich auch einige „erweiterte“ Darstellungsarten identifizieren.

- dreidimensionale Repräsentationen
- 2,5-dimensionale Repräsentationen
- interaktionsbasierte Repräsentationen
- Repräsentationen mit temporalen Anteilen

Obwohl visuelle Sprachen (zumindest derzeit) überwiegend auf zweidimensionalen Repräsentationen basieren, gibt es natürlich auch Varianten mit dreidimensionalen Darstellungen. Solche Darstellungen können unter gewissen Umständen die Übersichtlichkeit fördern. Das in Abbildung 2.7 gezeigte Hierarchiewerkzeug in VisaVis [46] dient z.B. dazu, Programmteile verschiedener Abstraktionsstufen miteinander in Beziehung zu setzen. Ein Nachteil dreidimensionaler Repräsentationen ist allerdings, dass sie mit den heute gebräuchlichen Peripheriegeräten schwerer handhabbar sind. Da z.B. Ausdrucke nur zweidimensionale Projektionen des Programms sind, ist deren Nutzen beschränkt.

Der Begriff „2,5-dimensional“ stammt ursprünglich von Glinert [17]. Dieser hat die von Programmiersprachen genutzten räumlichen Dimensionen analysiert und erkannt, dass zur Darstellung eines Programms oft mehrere verbundene zweidimensionale Diagramme verwendet werden. Beispielsweise kann oft zur Anzeige von Details eines Sprachobjekts ein eigenes Fenster, d.h. eine Detail-Sicht geöffnet werden. LabVIEW (siehe Abschnitt 2.2.3) enthält eine Variante dieses Mechanismus. Dort sind bestimmte Programmteile innerhalb einer Sicht visuell „übereinandergestapelt“, so dass jeweils nur ein Stapelelement sichtbar ist. Auch die Repräsentation in Abbildung 2.7 ist eigentlich nur 2,5-dimensional, da in der dritten Dimension nur diskrete Ebenen genutzt werden. Zumindest gibt es dort aber die Möglichkeit, beliebig im Raum zu navigieren.

Interaktionsbasierte Repräsentationen zeichnen sich dadurch aus, dass die Interaktion zu einem wichtigen Mittel der Programm-Exploration gemacht

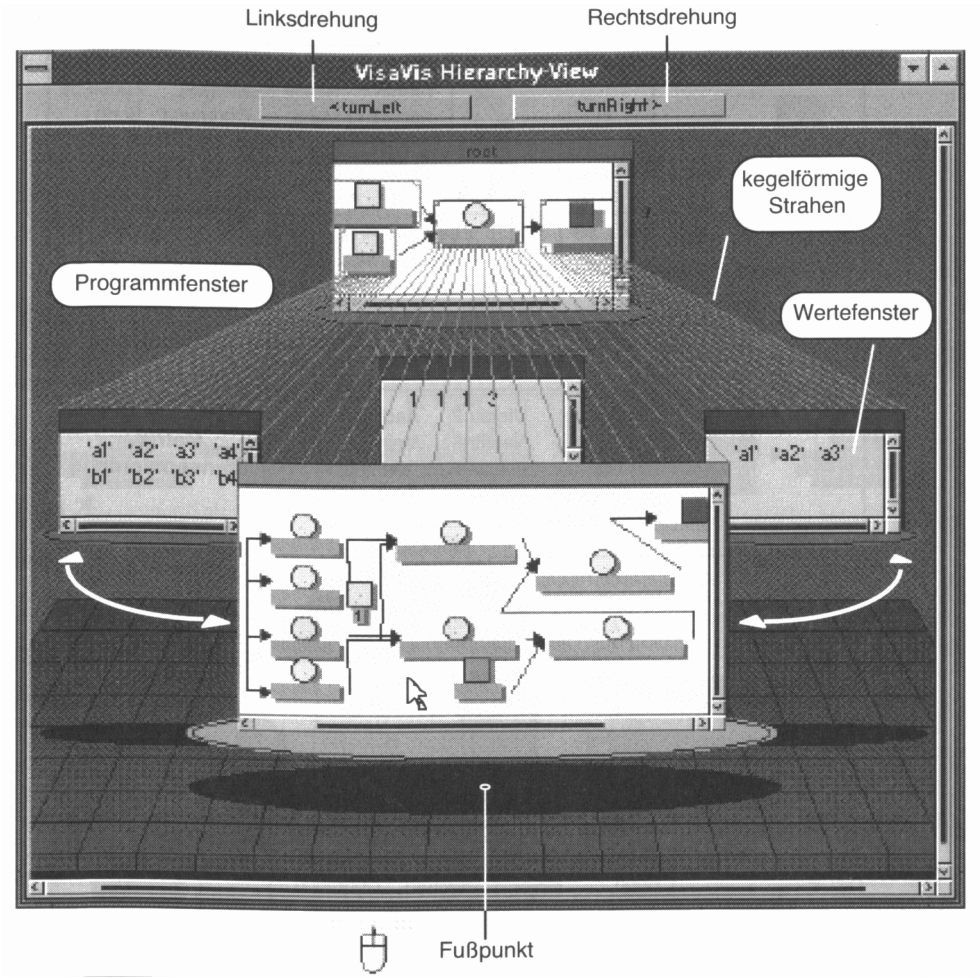


Abbildung 2.7: Hierarchiewerkzeug in VisaVis (aus [46])

wird. Ein Beispiel sind auf- und zuklappbare Details, wie sie z.B. bei der Baumdarstellung in Dateimanagern verwendet werden. Ein weiteres Beispiel sind Sprechblasen mit Zusatzinformationen, die sich öffnen, wenn der Sprachbenutzer den Mauszeiger über ein visuelles Objekt bewegt. Auch die „übereinandergestapelten" Programmteile in LabVIEW können als interaktionsbasiert betrachtet werden, da auch in diesem Fall Interaktionen erforderlich sind, um verdeckte Elemente sichtbar zu machen.

Bei Repräsentationen mit temporalen Anteilen spielt die zeitliche Dimension eine Rolle. In diesem Fall werden bestimmte Informationen nicht statisch, sondern nur durch zeitliche Abläufe repräsentiert. Diese Technik wird häufig in Systemen verwendet, bei denen der Benutzer bestimmte Abläufe „programmiert", indem er sie selbst durchführt (z.B. [61]).

Dimension 2: Layoutfreiheiten

„Layoutfreiheit" bedeutet in diesem Zusammenhang, dass der Sprachbenutzer die Möglichkeit hat, bestimmte, für die Programmsemantik unbedeutende Darstellungsdetails zu beeinflussen, um die Übersichtlichkeit zu verbessern.

Editoren können dem Benutzer sowohl kontinuierliche als auch diskrete Layoutfreiheiten einräumen. Ein Beispiel für eine kontinuierliche Layoutfreiheit findet sich in einem Graph-Editor, bei dem die Knoten des Graphen vom Benutzer frei positioniert werden können. Ein Beispiel für eine diskrete Layoutfreiheit findet sich in einem Strukturediteur für eine textuelle Sprache, bei dem die Zeilenumbrüche durch den Benutzer bestimmt werden können. Auch alternative Darstellungsvarianten eines semantischen Konstrukts kann man als diskrete Layoutfreiheiten auffassen. In textuellen Sprachen gibt es z.B. teilweise semantisch äquivalente Schreibweisen wie „`repeat X until Y`" und „`until Y do X`".

Im Modell in Abbildung 2.6 machen sich Layoutfreiheiten dadurch bemerkbar, dass der editierbaren Struktur in Bezug zur semantischen zusätzliche Informationen hinzugefügt werden müssen. Das bedeutet auch, dass der Import einer semantischen Struktur schwieriger wird. Da z.B. UML sowohl viele diskrete als auch viele kontinuierliche Layoutfreiheiten besitzt ist es dort schwierig, eine semantische Struktur (XMI) zu importieren und angemessen darzustellen.

Im Bereich der visuellen Sprachen haben Layoutfreiheiten Vor- und Nachteile. Ein Vorteil ist, dass der Sprachbenutzer dadurch seine eigenen Layoutvorstellungen umsetzen kann und so auch so genannte *sekundäre Notationen* [19] einbringen kann. Hierdurch kann der Sprachbenutzer dem menschlichen Betrachter des Programms zusätzliche informelle Informationen bereitstellen. Ein Nachteil ist allerdings, dass visuelle Repräsentationen mit vielen kontinuierlichen Layoutfreiheiten häufig eine hohe *Viskosität* [19] besitzen, d.h. dass Änderungen des Programms sehr zeitaufwändig sein können, da in diesem Fall auch das Layout von Hand angepasst werden muss. Aus diesem Grund ist es wichtig, dass eine Sprachimplementierung hier Hilfsmittel zur Verfügung stellt und einen angemessenen Mittelweg zwischen beiden Extremen realisiert.

Dimension 3: Interaktionsmechanismen

Auch die Mechanismen zur Interaktion mit der grafischen Darstellung können sehr unterschiedlich sein. Sie hängen sehr stark von der Art der grafischen Darstellung ab.

In grafischen Oberflächen zur Textverarbeitung, die nach vorheriger Diskussion auch Strukturediteren sind, wird die Eingabeposition durch einen permanent vorhandenen *Cursor* bestimmt.

Bei anderen Strukturediteren (z.B. solche, die DEViL generiert) sind die Einfügestellen abhängig vom Objekt, das eingefügt werden soll. Daher muss zunächst das einzufügende Objekt ausgewählt werden, bevor eine entsprechende Einfügestelle ausgewählt werden kann. Ein ähnlicher Interaktionsmechanismus findet sich auch bei Programmen zur Bearbeitung von Vektorgrafiken.

Eine dritte Klasse von Systemen basiert nicht auf Einfügestellen, sondern auf dem Anwenden von Operationen auf Objekte. In Strukturediteren für textuelle Sprachen werden z.B. Nichtterminale ausgewählt, die dann basierend auf einer Grammatik-Produktion „expandiert“ werden. Auch Generatoren für visuelle Strukturediteren, die auf Graphgrammatiken basieren, generieren Editoren mit entsprechenden Interaktionstechniken. Das führt dazu, dass teilweise sehr viele Objekte selektiert werden müssen, bevor eine entsprechende Graphgrammatik-Produktion angewendet werden kann.

Die Editieroperationen können auf verschiedene Weise ausgelöst werden. Häufig wird in diesem Zusammenhang das Stichwort *direct manipulation* [60] genannt. Bei diesem Ansatz kann der Anwender die grafischen Objekte mit

der Maus direkt „anfassen“ und manipulieren, also z.B. verschieben. Operationen können aber auch durch Betätigen von Knöpfen, durch Menübefehle oder durch Tastaturkürzel ausgelöst werden. Für Anfänger gilt *direct manipulation* als sehr benutzerfreundlich. Für erfahrene Benutzer ist es allerdings wichtig, auch Tastaturkürzel für häufig angewendete Operationen bereitzustellen.

An dieser Stelle konnte natürlich nur ein kurzer Überblick über dieses Thema gegeben werden. Interessant ist aber, dass die eingesetzten Mechanismen sehr anwendungsspezifisch zu sein scheinen und stark von der grafischen Darstellung abhängen.

Dimension 4: Niveau der editierbaren Struktur

Das Niveau der editierbaren Struktur bestimmt, wie eng die Konzepte der editierbaren Struktur an die Konzepte der semantischen Struktur angelehnt sind.

Ein Beispiel dafür, dass es sinnvoll sein kann, ein niedrigeres Niveau der editierbaren Struktur zu wählen, ist ein Petrinetz-Editor. In der semantischen Struktur ist es sinnvoll, Sprachkonstrukte zum Verbinden von Stellen mit Transitionen und Sprachkonstrukte zum Verbinden von Transitionen mit Stellen zu unterscheiden, da beide Konstrukte bei der Übersetzung evtl. unterschiedlich behandelt werden müssen. Für die editierbare Struktur ist es aber sinnvoll, hierfür nur ein Konstrukt bereitzustellen, um die Benutzerfreundlichkeit des Editors zu verbessern. Welches der beiden semantischen Konstrukte gemeint ist, ergibt sich in diesem Fall erst durch Betrachtung der Endpunkte.

Ein weiteres Beispiel findet sich bei Strukturediteren für textuelle Sprachen. Hier ergibt sich die Editor-Syntax häufig aus einer kontextfreien Grammatik. Die sich durch Namensanalyse ergebenden Querrelationen sind in diesem Fall nicht Teil der editierbaren Struktur. Betrachtet man die Querrelationen als Teil der semantischen Struktur, so hat auch hier die editierbare Struktur ein niedrigeres Niveau.

Im Kontext dieser Betrachtung lassen sich auch Sprachimplementierungen einordnen, die auf angepassten Universaleditoren basieren. Solche Systeme besitzen einen relativ sprachunabhängigen, freien Editor und wenden Parsing-Techniken an, um aus der editierbaren Struktur eine sprachspezifi-

sche semantische Struktur abzuleiten. Ein Beispiel für ein System, das solche Editoren generiert, ist DiaGen II (siehe Abschnitt 2.5.6).

Im Sinne dieser Klassifikation lassen sich auch freie Editoren als Strukturediteren interpretieren, wenn man die zugrundeliegende Datenstruktur solcher Editoren als Sprache sehr niedrigen Niveaus betrachtet. Solche Sprachen beschreiben Diagramme durch die Sprachmittel „grafisches Primitiv“, „Position“, „Größe“ und evtl. „Konnektor“ und „Verbindung“. Diese Sprache hat natürlich keinen direkten Bezug zu der Diagrammsprache, für die der Editor eingesetzt wird. Hieran sieht man, dass der Begriff „Strukturediteor“ immer auf eine bestimmte Sprache bezogen werden muss. Für eine Sprache zur Beschreibung von Zeichensequenzen ist Emacs ein Strukturediteor, für Java ein freier Editor.

Unter Verwendung der oben eingeführten Begriffe lässt sich ein Strukturediteor also wie folgt definieren: Ein Editor ist ein Strukturediteor für die Sprache L, wenn die editierbare Struktur die gleichen Sprachkonzepte und ein vergleichbares Niveau wie die semantische Struktur von L besitzt.

Dimension 5: Schärfe der Editor-Syntax

Die Editor-Syntax modelliert „gültige“ Strukturen. In Strukturediteoren ist es normalerweise nicht möglich, eine bzgl. dieser Syntax „ungültige“ Struktur zu konstruieren. Allerdings bedeutet das nicht, dass die in diesem Sinne gültigen Strukturen auch semantisch korrekt sind, denn selbst wenn das Niveau der editierbaren Struktur hoch ist, beschreibt die editierbare Syntax im Allgemeinen nur eine Obermenge aller korrekten Programme. Die Schärfe der editierbaren Syntax bestimmt, wie viel Spielraum zwischen syntaktisch korrekten und semantisch korrekten Programmen existiert.

Die Schärfe der editierbaren Syntax ergibt sich aus dem Kalkül der Syntaxbeschreibung. Im traditionellen Übersetzerbau ist die Schärfe recht gering, denn die Syntax textueller Sprachen wird normalerweise durch eine kontextfreie Grammatik beschrieben. Ergänzende Prüfungen werden hier auf einer übergeordneten Spezifikationsebene durch attributierte Grammatiken realisiert. Demgegenüber lässt sich durch bestimmte graphbasierte Modellierungsansätze eine wesentlich größere Schärfe erreichen.

Es ist allerdings zu beachten, dass eine schärfere Modellierung nicht unbedingt erstrebenswert ist. Da ein Strukturediteor normalerweise keine syntaktisch inkorrekten Strukturen erlaubt, kann eine schärfere Syntax unter Um-

ständen zu komplexeren und damit benutzerunfreundlicheren Editieroperationen führen (siehe Abschnitt 2.5.5). Des Weiteren ist es schwierig, bei Syntaxfehlern sinnvolle Fehlermeldungen zu erzeugen. Werden Konsistenzbedingungen dagegen durch separate Prüfungen modelliert, können Klartext-Fehlermeldungen ausgegeben werden.

Dimension 6: Anzahl der Repräsentanten eines semantischen Objekts

Im einfachsten Fall gehört zu jedem Objekt der semantischen Struktur genau ein grafisches Objekt der konkreten Repräsentation. Eine Steigerung ergibt sich, wenn ein Strukturediteor mehrere parallele Sichten bietet. In diesem Fall kann das gleiche semantische Objekt durchaus mehrere Repräsentationen besitzen. In LabVIEW (siehe Aschnitt 2.2.3) gibt es z.B. separate Sichten für die Oberfläche und die Verschaltung eines virtuellen Instruments. Jedes Bauteil tritt in beiden Sichten auf, die in den Sichten dargestellten Informationen sind aber vollkommen unterschiedlich.

Diese Fälle, in denen die Anzahl der Objekt-Repräsentanten zumindest konstant ist, lassen sich noch relativ einfach realisieren. Es gibt aber auch Fälle, bei denen eine nicht-konstante Anzahl von Objekt-Repräsentanten erforderlich ist. Ein Beispiel sind UML-Klassendiagramme, in denen die gleiche Klasse in beliebig vielen Diagrammen vorkommen darf. Ein anderes Beispiel ist ein Editor, der beliebig viele Sichten auf einen Graphen erlaubt, wobei in jeder Sicht ein anderes Layout definiert werden kann. In diesem Fall müssen die Repräsentanten der entsprechenden Objekte explizit in der editierbaren Struktur modelliert sein, da jedem individuelle Layoutinformationen zugeordnet werden müssen. Für die Implementierung bedeutet dies, dass sich hier editierbare und semantische Struktur grundlegend unterscheiden.

Welche Strukturediteoren generiert DEViL?

Basierend auf dem oben vorgestellten Klassifikationsschema lassen sich Strukturediteoren und Generatoren für Strukturediteoren einordnen und vergleichen. Nachfolgend sollen die von DEViL generierten Strukturediteoren anhand dieses Schemas eingeordnet werden.

Art der konkreten Repräsentation: Mit DEViL können visuelle Strukturediteoren generiert werden, die als Spezialfall auch alle oben genannten „eingeschränkten“ Sichttypen enthalten können. Dreidimensionale und temporale

Repräsentationen sind nicht vorgesehen. Interaktionsbasierte Repräsentationen werden unterstützt und einige visuelle Muster enthalten dafür bereits konkrete Implementierungen.

Layoutfreiheiten: Mit DEViL können Darstellungen mit automatischem Layout und mit beiden Arten von Layoutfreiheiten realisiert werden. Allerdings bietet DEViL keine Unterstützung für ein initiales Layout von semantischen Strukturen mit Layoutfreiheiten. Das Konzept der visuellen Muster besitzt allerdings das Potenzial, hier nachzurüsten.

Interaktionsmechanismen: Der Interaktionsmechanismus von DEViL basiert stark auf objektspezifischen Einfügestellen und nutzt das *direct manipulation* Konzept. Darüber hinaus gibt es einen generischen Interaktionsmechanismus auf Basis von Kontextmenü-Operationen sowie einen generischen Mechanismus zur Tastaturnavigation.

Niveau der editierbaren Struktur: DEViL ist für Editoren konzipiert, bei denen editierbare und semantische Struktur das gleiche Niveau haben. Kleinere Unterschiede des Niveaus können durch spezifizierbare Anpassungen der Kopplung zwischen editierbarer und semantischer Struktur überbrückt werden. Zur Überbrückung größerer Niveauunterschiede sind allerdings keine adäquaten Hilfsmittel vorgesehen.

Schärfe der Editor-Syntax: Die Schärfe der Syntax entspricht ungefähr der von UML-Strukturdiagrammen, d.h. Baumstrukturen, Querbeziehungen und Kardinalitäten können syntaktisch modelliert werden. Weitere Einschränkungen können durch strukturelle Constraints definiert werden, die aber temporär beim Editieren verletzt werden dürfen. Als Spezialfall können mit DEViL auch Strukturediteren generiert werden, die sich auf die Baumstruktur beschränken.

Anzahl der Repräsentanten eines semantischen Objekts: DEViL wurde gezielt so entworfen, dass auch Editoren mit einer variablen Anzahl von Objekt-Repräsentanten realisiert werden können. Dabei wurde darauf geachtet, dass hierdurch kein zusätzlicher Aufwand zur Spezifikation von Editoren einfacherer Klassen entsteht.

2.3.3 Implementierung visueller Strukturediteren

Um näher auf die Implementierung von Strukturediteren eingehen zu können ist es notwendig, das Schema aus Abbildung 2.6 zu verfeinern. Sowohl

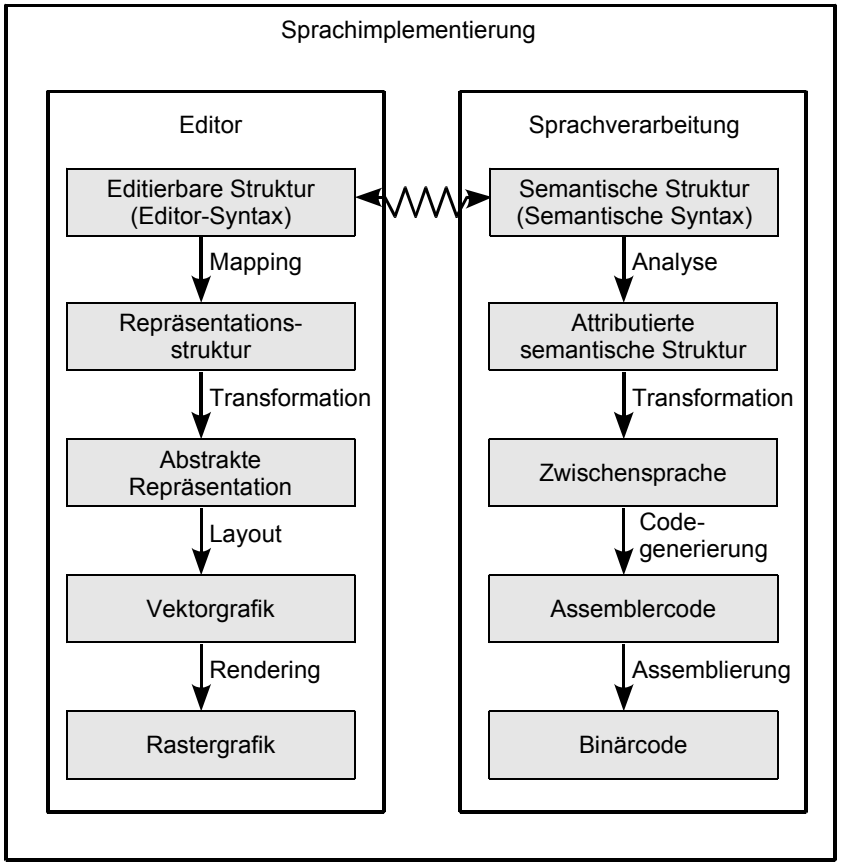


Abbildung 2.8: Verfeinertes Grundmodell zur Sprachimplementierung

die Abbildung der editierbaren Struktur auf die konkrete Repräsentation als auch die Abbildung der semantischen Struktur auf die Zielrepräsentation sind komplexe Aufgaben, die normalerweise in mehreren Schritten gelöst werden.

Abbildung 2.8 zeigt ein verfeinertes Modell zur Sprachimplementierung. Obwohl die dargestellten Zwischenprodukte und Abbildungsschritte nicht allgemeingültig sind, lassen sich vergleichbare Transformationsketten in vielen Sprachimplementierungen wiederfinden. Zumindest die Transformationskette zur Sprachübersetzung ist allgemein anerkannt und findet sich in Lehrbüchern zum Übersetzerbau. Die Transformationskette auf der linken Seite habe ich als „kleinsten gemeinsamen Nenner“ vieler Editorimplementierungen identifiziert. Als Beispiele für Sprachimplementierungen, die solche Transformationsketten verwenden, werden weiter unten *Proxima* und mit Eli generierte Unparser vorgestellt.

Die editierbare Struktur wird zunächst auf die Repräsentationsstruktur abge-

bildet, die ein Hilfsmittel zur Implementierung der grafischen Repräsentation ist. Die Repräsentationsstruktur hat ein ähnliches Niveau wie die editierbare Struktur, aber enthält bereits Methoden und Daten für die nachfolgende Transformation. Im DEViL-System sind der Repräsentationsstruktur visuelle Muster zugeordnet. In einer objektorientierten Handimplementierung würden die Objekte der Repräsentationsstruktur Berechnungen kapseln, die bestimmen, wie bestimmte Sprachkonstrukte dargestellt werden.

Während die Repräsentationsstruktur noch eine sprachspezifische Syntax hat, beschreibt die abstrakte Repräsentation wesentliche Darstellungseigenschaften in einem konzeptionell sprachunabhängigen Kalkül. In realen Systemen könnte dies z.B. die Eingabesprache eines Constraintsolvers sein. Im Allgemeinen enthält die abstrakte Darstellung eine qualitative Bildbeschreibung wie z.B. „der Kreis A muss innerhalb des Rechtecks B dargestellt werden“ oder „der Text C soll im Blocksatz dargestellt und eingerückt werden“. In anderen Ansätzen wird diese Ebene auch *spatial relations graph* genannt (siehe Abschnitt 2.5.5).

Durch das Layout werden den Layoutattributen der grafischen Objekte bestimmte Werte zugewiesen. Layoutattribute sind vor allem Größen und Positionen, können in anderen Fällen aber z.B. auch Farben, Schraffuren oder generierte Namen sein. Das Ergebnis des Layout-Schrittes ist eine Vektorgrafik, d.h. eine Menge grafischer Primitive wie Rechtecke, Linien oder Texte mit bestimmten Attributen wie Größe, Farbe oder Schriftart.

Rendering ist der Prozess, der eine Vektorgrafik in eine Rastergrafik überführt, so dass sie auf Ausgabegeräten wie Bildschirmen oder Drucker ausgegeben werden kann. Dieser Abbildungsschritt ist eine Standardaufgabe jedes Grafiksystems, weshalb darauf in dieser Arbeit nicht näher eingegangen wird.

Die auf der rechten Seite befindliche Verarbeitungskette habe ich aufgeführt, um die Korrespondenz beider Ketten zu zeigen. Die Repräsentationsstruktur und die attributierte semantische Struktur sind beides Hilfsstrukturen, um eine komplexe strukturelle Transformation durchzuführen. Die abstrakte Repräsentation und die Zwischensprache sind beide sprachunabhängige Abstraktionen, die schon recht nah am Übersetzungsziel liegen. Beim Layout bzw. der Codegenerierung wird jeweils ein Optimierungsproblem gelöst, bei dem eine möglichst gute Lösung unter Einhaltung bestimmter Randbedingungen zu finden ist. Schließlich ist das Rendering bzw. die Assemblierung eine eher technische Formatkonvertierung auf ein elementareres Niveau.

Zur Klarstellung sei betont, dass die oben beschriebene Korrespondenz nicht bedeutet, dass man die Aufgaben mit den gleichen Methoden lösen kann. Sie zeigt lediglich, dass die Transformationsketten nicht vollkommen willkürlich sind, sondern dass sie Instanzen eines übergeordneten Modells sind, das sich zur Sprachverarbeitung bereits als nützlich erwiesen hat.

Konkrete Ausprägungen der Verarbeitungskette In anderen Systemen finden sich Abbildungsketten, die der oben vorgestellten sehr ähnlich sind. *Proxima* [56] z.B. ist ein Editor, mit dem XML-Strukturen anhand einer grafischen Repräsentation editiert werden können. Er ist in der funktionalen Sprache *Haskell* [24] implementiert. Die Abbildung vom Dokument zur konkreten Darstellung wurde in mehrere Teilabbildungen zerlegt, um die Implementierung zu modularisieren. Es ergeben sich mehrere Zwischenrepräsentationen:

- Das *Dokument* (Originalbezeichnung *document*) repräsentiert die Daten der XML-Datei. In meiner Terminologie ist dies die editierbare Struktur.
- Das *bereicherte Dokument* (Originalbezeichnung *enrichted document*) enthält zusätzliche berechnete Informationen, wie z.B. Nummerierungen. In meiner Terminologie ist dies die Repräsentationsstruktur.
- Die *abstrakte Darstellung* (Originalbezeichnung *abstract presentation*) beschreibt die Struktur der Repräsentation, wobei noch vom absoluten Layout abstrahiert wird; die abstrakte Darstellung beschreibt lediglich die relative Positionierung. In meiner Terminologie ist dies die abstrakte Repräsentation.
- Das *Arrangement* (Originalbezeichnung *arrangement*) weist allen grafischen Elementen absolute Positionen zu und berücksichtigt auch Trennungen sowie Zeilen- und Seitenumbrüche. In meiner Terminologie ist dies die Vektorgrafik.
- Die *Wiedergabe* (Originalbezeichnung *rendering*) schließlich ist die fertige Darstellung als Pixelgrafik. In meiner Terminologie ist dies die Rastergrafik.

Auch das so genannte *Unparsing* textueller Sprachen lässt sich als Instanz dieser Transformationskette interpretieren. Das soll am Beispiel des Eli-Systems verdeutlicht werden, das automatisch Unparser aus konkreten Grammatiken

generieren kann. Die Eingabe des Unparsers ist im Fall von Eli der Strukturbaum eines Programms. Dieser entspricht hier der editierbaren Struktur.

Auf Basis der konkreten Grammatik wird eine attributierte Grammatik generiert, die den Strukturbaum in eine textuelle Repräsentation übersetzen kann. Der attributierte Strukturbaum spielt die Rolle der Repräsentationsstruktur, denn hierauf basiert der eigentliche Abbildungsprozess.

Im Falle von Eli werden während der Attributauswertung so genannte PTG-Funktionen [1] aufgerufen. PTG ist ein Werkzeug zur Generierung von strukturiertem Ausgabetext. Durch Aufruf von PTG-Funktionen wird ein Ausgabetext abstrakt beschrieben. Zeilenumbrüche und Einrückungen müssen z.B. noch nicht explizit festgelegt werden, sondern es kann eine maximale Zeilenbreite angegeben werden. Die Einrückung wird auf hohem Niveau durch Befehle wie „Einrückung erhöhen“ und „Einrückung verringern“ spezifiziert. In meiner Terminologie spielt die PTG-Zwischenstruktur die Rolle der abstrakten Repräsentation, da sie das Layout nur auf abstrakter Ebene definiert.

Das PTG-System generiert aus der Zwischenrepräsentation den eigentlichen Zieltext in Form einer Textdatei. Diese Textdatei spielt die Rolle der Vektorgrafik. Erst, wenn diese Datei angezeigt oder ausgedruckt wird, wird durch das Betriebssystem bzw. durch den Drucker die Transformation in eine Rastergrafik durchgeführt.

Vergleichbare nicht-funktionale Ansätze Auch der SRG-ASG-Ansatz zur Spezifikation visueller Sprachen (siehe Abschnitt 2.5.5) unterscheidet ähnliche strukturelle Abstraktionen.

- Der *abstract syntax graph* (ASG) ist ein Graph, der die logische Struktur des visuellen Programms repräsentiert. Da basierend auf diesem Graphen auch die strukturellen Editieroperationen definiert sind, ist dies in meiner Terminologie die editierbare Struktur.
- Der *spatial relations graph* (SRG) ist ein Graph, dessen Knoten die grafischen Objekte und dessen Kanten die räumlichen Relationen wie Berührung oder Enthaltensein beschreiben. In meiner Terminologie ist dies also die abstrakte Repräsentation.
- Unter *physikalischem Layout* wird eine Menge grafischer Grundelemente mit Eigenschaften wie Position oder Farbe verstanden, die unmittelbar die grafische Repräsentation beschreiben. In meiner Terminologie ist dies Vektorgrafik.

Auch hier sind also äquivalente Abstraktionen unterschiedlichen Niveaus zu erkennen. Die Kopplung von ASG und SRG ist in diesem Ansatz jedoch nicht funktional, sondern basiert auf gekoppelten Graphgrammatiken. Eine Besonderheit dieses Ansatzes ist, dass die Transformationskette auch in umgekehrter Richtung durchlaufen werden kann.

Akehurst [3] benutzt als Grundlage seiner Arbeit die aus dem SRG-ASG-Ansatz bekannten Begriffe, verfolgt aber einen anderen Spezifikationsansatz. Die Syntax des ASG und SRG werden in diesem Ansatz durch UML-Klassendiagramme modelliert. Der Zusammenhang zwischen den Strukturen wird durch OCL-Constraints beschrieben, wobei zur Vereinfachung der Spezifikation neue UML-Stereotypen definiert werden. Aus den Spezifikationen können Klassen generiert werden, die bei Verletzung der Constraints *Events* auslösen. Der Sprachentwickler muss Methoden implementieren, die auf diese *Events* reagieren und die Strukturen entsprechend anpassen.

2.4 Das VL-Eli System

Der Vorgänger von DEViL, das VL-Eli System, wurde in Kooperation mit Matthias Jung und Christian Schindler [29] entwickelt. Der Entwurf des Systems ist in der Dissertation von Jung [28], die Entwicklung der visuellen Muster in [54] beschrieben. Ab 2001 habe ich das System eigenverantwortlich weiterentwickelt [52, 32]. Wie schon in der Einleitung erwähnt war es mein Ziel ein Nachfolgesystem zu entwickeln, das die positiven Eigenschaften von VL-Eli übernimmt, seine Schwächen vermeidet und auch für anspruchsvolle visuelle Sprachen mit einer variablen Anzahl von Repräsentanten eines semantischen Objekts geeignet ist.

Nachfolgend möchte ich VL-Eli vorstellen und auf seine Stärken und Schwächen sowie die Unterschiede zu DEViL eingehen. Einerseits werden dadurch die Grundlagen zum Verständnis des DEViL-Systems gelegt, andererseits wird so deutlich, wie sehr sich DEViL von VL-Eli unterscheidet.

Abbildung 2.9 gibt einen Überblick über die Architektur von VL-Eli. Der VL-Generator generiert visuelle Strukturediteren aus Spezifikationen. Er basiert auf Werkzeugen zur Implementierung grafischer Oberflächen (Tcl/Tk [43] und Parcon [20]) und zur Implementierung von Sprachen im Allgemeinen (Eli [31]). Die oberste Schicht ist eine Bibliothek, die Implementierungen so genannter visueller Muster kapselt. Jede dieser Implementierungen realisiert

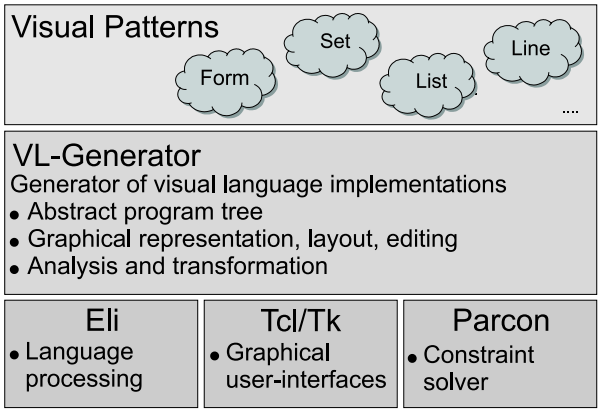


Abbildung 2.9: Architektur des VL-Eli Systems

ein bestimmtes visuelles Darstellungskonzept inklusive Interaktion und Layout und basiert auf den Spezifikationsmechanismen des VL-Generators.

Abbildung 2.10 zeigt die Struktur der von VL-Eli generierten Sprachimplementierungen. Die abstrakte Struktur der Sprache wird durch eine herkömmliche kontextfreie Grammatik beschrieben. Unter abstrakter Struktur ist hier sowohl die editierbare als auch die semantische Struktur zu verstehen, da VL-Eli nicht zwischen diesen unterscheidet. Auf Basis der abstrakten Struktur kann eine beliebige Anzahl voneinander unabhängiger attributierter Grammatiken definiert werden. Aus jeder attributierten Grammatik wird ein separater Attributauswerter generiert. Die attributierten Grammatiken definieren entweder visuelle Sichten oder Codegeneratoren. Im ersten Fall spielen deren Instanzen die Rolle der Repräsentationsstruktur, im zweiten Fall die der attributierten semantischen Struktur.

2.4.1 Spezifikation der abstrakten Struktur

Die Spezifikation der abstrakten Struktur basiert auf kontextfreien Grammatiken, die gewisse EBNF-Konstrukte enthalten dürfen. Abbildung 2.11 zeigt einen Ausschnitt aus einer abstrakten Syntax für Zustandsdiagramme. Beispielsweise besteht ein Statechart-Objekt aus einer Menge von `States` und einer Menge von `Transitions`. Ein `State` kann u.a. ein `SimpleState` oder ein `XORSuperstate` sein.

Zusätzlich zu der in Abbildung 2.11 gezeigen Grammatik können den Symbolen so genannte persistente Attribute zugeordnet werden, die z.B. Namen

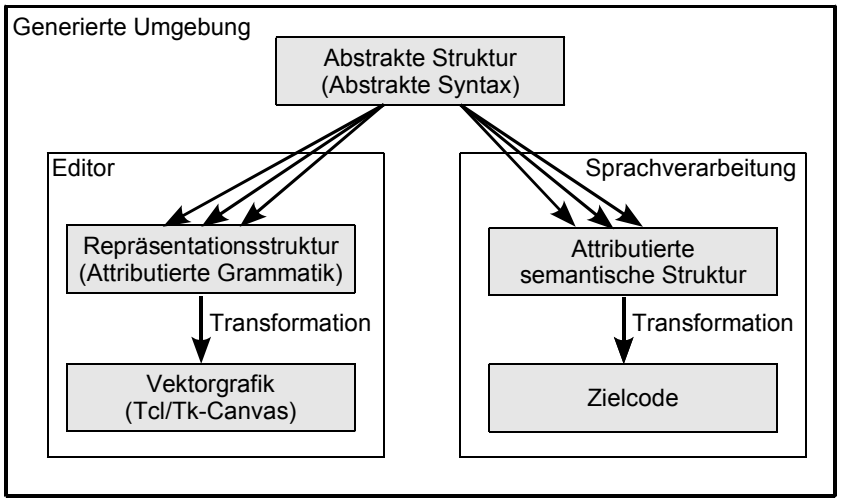


Abbildung 2.10: Struktur der von VL-Eli generierten Sprachimplementierungen

```
RULE: Statechart ::=  Statechart_TopState  Transitions     END;
RULE: Statechart_TopState LISTOF State                     END;
RULE: Transitions LISTOF Transition                        END;

RULE: State ::= XORSuperstate                              END;
RULE: XORSuperstate ::= XORSuperstate_Name XORStates       END;
RULE: XORSuperstate_Name ::=                               END;
RULE: XORStates LISTOF State                               END;

RULE: State ::= ANDSuperstate                              END;
RULE: ANDSuperstate ::= ANDSuperstate_Name ANDStateRegions END;
RULE: ANDSuperstate_Name ::=                               END;
RULE: ANDStateRegions LISTOF ANDStateRegion                END;
RULE: ANDStateRegion LISTOF State                          END;

RULE: State ::= SimpleState                                END;
RULE: SimpleState ::=                                      END;

RULE: State ::= HistoryState                               END;
RULE: HistoryState ::=                                     END;

RULE: State ::= InitialState                               END;
RULE: InitialState ::=                                     END;

RULE: State ::= FinalState                                 END;
RULE: FinalState ::=                                       END;

RULE: Transition ::= Transition_Label                      END;
RULE: Transition_Label ::=                                 END;
```

Abbildung 2.11: VL-Eli Grammatik für Zustandsdiagramme

oder andere Eigenschaften von Sprachobjekten speichern können. Den Symbolen `SimpleState` und `XORSuperstate` wird z.B. jeweils das persistente Attribut „name“ von Typ `VLString` zugeordnet.

Da eine kontextfreie Grammatik nur die „consists-of“ Relation zwischen Sprachkonstrukten definiert, gibt es spezielle persistente Attribute, die Querbeziehungen modellieren. Sie haben den Typ `Program_obj`. Solche Attribute referenzieren Verbindungsobjekte in einer Definitionstabelle. Zwei Programmkonstrukte sind über eine Querbeziehung miteinander verbunden, wenn Attribute ihrer Knoten das gleiche Verbindungsobjekt referenzieren. Ein `Transition`-Konstrukt ist mit zwei `State`-Konstrukten verbunden, also hat es die zwei Verbindungs-Attribute `from` und `to`. Ein `SimpleState`-Konstrukt hat das Attribut `Endpoint`. Es referenziert das gleiche Verbindungs-Objekt, das auch von `Transition`-Konstrukten referenziert wird, die mit diesem `SimpleState`-Konstrukt verbunden sind.

Durch diese Art der Spezifikation werden semantisch unsinnige Verbindungen nicht ausgeschlossen, z.B. könnte eine Transition fälschlicherweise mit einer anderen Transition verbunden werden. In VL-Eli lassen sich Einschränkungen von Verbindungen auf zweierlei Weise definieren. Durch Attributberechnungen können Gültigkeitsbedingungen geprüft und evtl. Fehlermeldungen generiert werden. In grafischen Sichten können die Editiermöglichkeiten eingeschränkt werden, so dass fehlerhafte Verbindungen überhaupt nicht konstruiert werden können. In den Implementierungen der visuellen Muster ist dies teilweise bereits vorgesehen.

Stärken und Schwächen Der Vorteil des Spezifikationsansatzes ist seine Einfachheit und die Äquivalenz zur Definition von textuellen Sprachen. Da in VL-Eli die editierbare und semantische Struktur nicht unterschieden wird, können allerdings Editoren mit einer variablen Anzahl von Objekt-Repräsentanten nur mit hohem Aufwand realisiert werden.

Sehr problematisch ist weiterhin, dass strukturelle Eigenschaften von Querbeziehungen nur nachträglich über Attributberechnungen definiert werden können. Da Querbeziehungen insbesondere keine Richtung und keine Kardinalität zugeordnet ist, werden zur sinnvollen Unterstützung von *Cut-and-Paste* im Allgemeinen Zusatzinformationen benötigt. Falls es mehrere Sichten gibt, entsteht ferner unnötige Redundanz, da in allen Sichten die gleichen Editiereinschränkungen spezifiziert werden müssen. Außerdem sind die Querbeziehungen auch für den Sprachentwickler unübersichtlich, da aus der

abstrakten Syntax nicht ersichtlich ist, welche Objekte wie miteinander verbunden werden können. Aus diesen Gründen wurden in DEViL typisierte Querreferenzen und Konsistenzbedingungen auf der Ebene der semantischen Struktur eingeführt, die durch die Interaktionsmechanismen in den Sichten automatisch berücksichtigt werden können.

Schließlich ist es problematisch, dass kontextfreie Grammatiken im Vergleich zu anderen Modellierungsmethoden wenig Strukturierungskonzepte unterscheiden. Relationen wie „part-of“, „inherits“ oder „subtype-of“ können zwar auf Grammatiken abgebildet werden, deren Unterscheidung geht dann allerdings verloren. In vielen Fällen ist diese Unterscheidung aber für das Editierverhalten sowie für den Umgang mit der editierbaren Struktur im Allgemeinen wichtig. Das führt dazu, dass in vielen Fällen Fallunterscheidungen zum Umgang mit Strukturen benötigt werden. Ob beim Löschen eines Objekts die übergeordnete Produktion ebenfalls gelöscht werden muss hängt z.B. davon ab, ob sie eine „subtype“-Relation oder ein eigenständiges Objekt modelliert. Auch Grammatik-Änderungen im Rahmen der Weiterentwicklung visueller Sprachen sind problematisch, denn sie führen schnell zu inkompatiblen Datenformaten bereits gespeicherter Strukturen. Aus diesem Grund basiert DEViL auf einer höheren Modellierungssprache, die wichtige Konzepte aus objektorientierten Ansätzen übernimmt und damit die genannten Probleme vermeidet.

2.4.2 Spezifikation der grafischen Darstellung

Die Spezifikation der visuellen Darstellung auf der Ebene des VL-Generators basiert auf attributierten Grammatiken. Dazu werden den Baumkontexten Berechnungen zugeordnet, die das Programm visualisieren (Abbildung 2.12). Die Berechnungen benutzen und definieren Attributwerte der Baumknoten und transportieren so Informationen. Die grafische Darstellung wird als Seiteneffekt in das zugehörige Sicht-Fenster gezeichnet.

Abbildung 2.13 zeigt eine vereinfachte Berechnung von Layoutinformationen. Zu jedem Knoten, der ein visuelles Objekt repräsentiert, werden zwei Attribute berechnet: `size`-Attribute charakterisieren die Minimalanforderung (Breite, Höhe, etc.) der grafischen Repräsentation des Teilbaums. `position`-Attribute definieren die Koordinaten, an denen der Teilbaum gezeichnet werden soll.

Die gezeigte Produktion hat drei Berechnungen. Das `size`-Attribut des

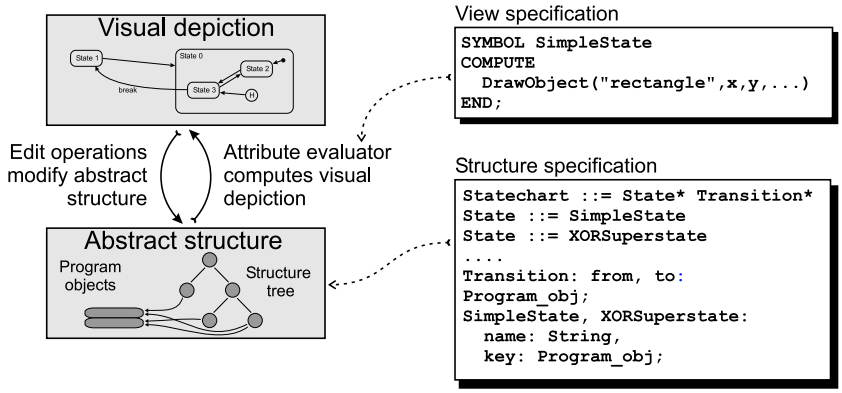


Abbildung 2.12: Konzept des VL-Generators in VL-Eli

```
RULE: ANDSuperstate ::= ANDSuperstate_Name ANDStateRegions COMPUTE
  ANDSuperstate.Size =
    calcSize (ANDSuperstate_Name.Size, ANDStateRegions.Size);

  ANDSuperstate_Name.Position =
    calcNamePosition (ANDSuperstate.Position, ANDSuperstate.Size);

  ANDStateRegions.Position =
    calcRegionsPosition (ANDSuperstate.Position, ANDSuperstate.Size);
END;
```

Abbildung 2.13: Layoutberechnungen für AND-Superstates in VL-Eli

`ANDSuperstate`-Konstrukts wird durch einen Funktionsaufruf berechnet, der die Größenanforderungen der zwei Unterbäume vereinigt. Die anderen beiden Berechnungen bestimmen die `position`-Attribute der Unterbäume auf Basis der `position`- und `size`-Attribute des `ANDSuperstates`. Die aufgerufenen Funktionen bestimmen die Details des Layouts und sind in diesem Beispiel so definiert, dass der `Name`-Knoten oberhalb des `ANDStateRegions`-Knotens dargestellt wird.

Andere Regelkontexte besitzen ähnliche Berechnungen. Die Größeninformation wird im Baum aufwärts propagiert und akkumuliert, während Positionierungsentscheidungen abwärts im Baum verfeinert werden. Basierend auf den so berechneten Positions- und Größenattributen werden schließlich grafische Primitive wie Rechtecke, Linien oder Texte in den Ausgabebereich gezeichnet. Des Weiteren werden auch unsichtbare Elemente und Kontextinformationen in die Grafik integriert, so dass der Benutzer mit der Grafik interagieren kann, um z.B. Elemente zu löschen oder neue einzufügen.

Auch in VL-Eli können mehrere Sichten auf ein Programm definiert werden.

Eine Sicht zeigt im Allgemeinen nicht das vollständige Programm, sondern lediglich einen bestimmten Teilbaum in einer bestimmten Perspektive. Jede Sicht wird in einem separaten Fenster angezeigt. Aus Effizienzgründen wird jeder Sichttyp durch eine separate attributierte Grammatik spezifiziert. Jedem Sichttyp ist ein Wurzelsymbol zugeordnet, das bestimmt, welche Teilbäume er visualisiert. Zur Berechnung der Darstellung werden nur Teilbäume durchlaufen, zu denen tatsächlich eine Sicht geöffnet ist.

Ein Sicht-Attributauswerter spielt in diesem Ansatz die Rolle der Repräsentationsstruktur. Die Syntax der Repräsentationsstruktur ist dabei identisch mit der Sprachstruktur, lediglich das Wurzelsymbol kann ein anderes sein.

Stärken und Schwächen Die 1:1-Relation zwischen Sprachstruktur und Repräsentationsstruktur ist für handgeschriebene Attributberechnungen durchaus ausreichend. Es hat sich jedoch herausgestellt, dass zur Anwendung visueller Muster mehr Flexibilität wünschenswert ist, denn im Allgemeinen muss die Struktur der Grammatik den anzuwendenden visuellen Mustern angepasst werden. Eine Grammatik wie in Abbildung 2.14a ist z.B. zur Modellierung der abstrakten Struktur vollkommen ausreichend. Um einen passenden Kontext zur Anwendung von Berechnungsrollen zu definieren, muss in VL-Eli aber eine Grammatik wie in Abbildung 2.14b verwendet werden. Solche Grammatiken zeichnen sich durch kontextspezifische Symbole wie z.B. `IfStmtExpr` und daraus resultierende Kettenproduktionen[2] wie „`IfStmtExp ::= Expr`“ aus. Die genauen Anforderungen bzgl. der Sichtspezifikation sind beim Entwurf der Sprachstruktur schwer zu überblicken und können sich noch dazu bei parallelen Sichten widersprechen. Aus diesem Grund erlaubt DEViL, die Syntax der Repräsentationsstrukturen individuell anzupassen (siehe Abschnitt 4.3).

2.4.3 Visuelle Muster

Visuelle Muster definieren eine höhere Abstraktionsebene zur Spezifikation visueller Darstellungen. Das zugrundeliegende Konzept und eine initiale Umsetzung wurde in [54] entwickelt.

Generell wird zwischen „abstrakten“ visuellen Mustern und konkreten Implementierungsvarianten unterschieden. Ein (abstraktes) visuelles Muster ist

[2]Kettenproduktionen sind Produktionen, die nur ein Nichtterminal auf der rechten Seite besitzen

```
IfStmt ::= Expr StmtList StmtList
StmtList LISTOF Stmt
```

(a) Einfache Grammatik

```
IfStmt ::= IfStmtExpr IfStmtTrueBranch IfStmtFalseBranch
IfStmtExpr ::= Expr
IfStmtTrueBranch LISTOF Stmt
IfStmtFalseBranch LISTOF Stmt
```

(b) Grammatik, die zur Anwendung visueller Muster geeignet ist

Abbildung 2.14: Beispiel zur Problematik des Grammatikentwurfs in VL-Eli

Abbildung 2.15: Unterschiedliche visuelle Muster zur Darstellung einer Folge

ein Darstellungskonzept für eine bestimmte Struktur. Die abstrakte Struktur „Folge von Elementen“ könnte z.B. durch das Listen-Muster dargestellt werden, indem die Elemente visuell entlang einer bestimmten Achse aufgereiht werden (Abbildung 2.15a). Nach einem anderen Muster könnte die gleiche Folge durch Nummerierung beliebig verteilter Elemente dargestellt werden (Abbildung 2.15b). Das Beispiel verdeutlicht schön das Wesen visueller Muster, allerdings sind in der Praxis vor allem Darstellungskonzepte relevant, die Einfluss auf das Layout haben, denn das Layout ist die komplexeste Aufgabe der Sichtberechnung. Aus diesem Grund gibt es für das zweitgenannte Muster weder in VL-Eli noch in DEViL eine vorgefertigte Implementierung.

In [54] wurden folgende (abstrakte) Muster definiert.

- Das *Formular-Muster* visualisiert ein abstraktes n-Tupel, indem die Tupelelemente in fester relativer Position zueinander angeordnet werden.

- Das *Registerkarten-Muster* visualisiert ein abstraktes n-Tupel, indem die Tupelelemente 2,5-dimensional übereinander gestapelt dargestellt werden.
- Das *Listen-Muster* visualisiert eine abstrakte Folge, indem die Elemente der Folge entlang einer bestimmten Raumrichtung aufgereiht werden.
- Das *Stapel-Muster* visualisiert eine abstrakte Folge, indem die Elemente als 2,5-dimensionaler Stapel angezeigt werden.
- Das *Tabellen-Muster* visualisiert eine Folge von Tupeln gleicher Struktur durch eine gitterförmige Anordnung, d.h. eine „visuelle“ Tabelle. Eine Tabellenzeile repräsentiert jeweils ein Tupel und eine Tabellenzelle ein Tupelelement.
- Das *Matrix-Muster* visualisiert eine abstrakte Matrix durch ein rechteckiges, gitterförmiges Schema, d.h. eine „visuelle“ Matrix.
- Das *Mengen-Muster* visualisiert eine abstrakte Menge durch einen zweidimensionalen Bereich, in dem die Mengenelemente beliebig angeordnet werden können.
- Das *Graph-Muster* visualisiert einen abstrakten Graphen durch eine Kombination von Linien- und Mengen-Muster.
- Das *Linien-Muster* visualisiert eine binäre Relation zwischen zwei Elementen durch eine Linienverbindung zwischen ihnen.
- Das *Attribut-Relations-Muster* visualisiert eine n-äre Relation dadurch, dass die Elemente in einem ihrer Attribute übereinstimmen. Solche Attribute können z.B. Namen, Farben, Formen oder Füllmuster sein.

Abstrakte visuelle Muster wurden vor allem deshalb definiert, um daraus konkrete Implementierungsvarianten abzuleiten. Solche Implementierungen konkretisieren das zugrundeliegende abstrakte Darstellungskonzept, was zwangsläufig mit Einschränkungen einhergeht. Dabei wird versucht, einen guten Kompromiss aus einfacher Anwendbarkeit und weitreichender Parametrisierbarkeit zu finden.

Im Einzelnen konkretisiert eine Muster-Implementierung folgende Eigenschaften.

- Die konkrete Darstellung

- Die Layoutmethode
- Die Interaktionsmechanismen

In allen Muster-Implementierungen sind diese Eigenschaften in gewissem Maße parametrisierbar. Beim Listen-Muster könnte z.B. die Darstellung des Rahmens um die Liste oder die Darstellung des Separators zwischen Listenelementen anpassbar sein. Beim Layout könnte wählbar sein, ob die Liste horizontal oder vertikal ausgerichtet sein soll. Bei den Interaktionsmechanismen könnte parametrisierbar sein, auf welche Weise Elemente eingefügt oder gelöscht werden können.

Natürlich hat jede Parametrisierbarkeit ihre Grenzen, z.B. erlaubt die Muster-Implementierung `SimpleList` in VL-Eli zwar horizontalen und vertikalen Listenverlauf, nicht aber diagonalen.

Muster-Implementierungen werden angewendet, indem Symbolen der Repräsentations-Grammatik Berechnungsrollen zugeordnet werden. Für das Listen-Muster wären dies z.B. `VPList` und `VPListElement`. Die Berechnungsrollen kapseln Attributberechnungen wie die in Abbildung 2.13. Durch Überdeckung der Repräsentationsstruktur mit solchen Berechnungsrollen wird eine vollständige Attributierung definiert, die die gewünschte Darstellung spezifiziert.

Stärken und Schwächen Das zweistufige Spezifikationskonzept aus allgemein verwendbaren Attributberechnungen und gekapselten visuellen Mustern hat sich bereits als sehr tragfähig erwiesen. Die Muster-Implementierungen in VL-Eli sind hinreichend parametrisierbar, so dass der überwiegende Teil einer Sprachspezifikation aus der Anwendung und Parametrisierung visueller Muster besteht. In den wenigen Fällen, in denen die Muster-Implementierungen nicht den Erfordernissen entsprechen, können diese durch Überschreiben oder Hinzufügen von Attributberechnungen ergänzt werden.

Noch unbefriedigend ausgearbeitet ist in VL-Eli das Modell zur Beschreibung der Kombinierbarkeit von Muster-Varianten. In dieser Arbeit führe ich daher eine einfache aber wirkungsvolle Methode zur Beschreibung der Kombinierbarkeit von Muster-Implementierungen ein und stelle darüber hinaus einige neue, methodisch interessante Muster-Implementierungen vor (siehe Abschnitt 4.1.3).

2.5 Andere Systeme zur Sprachimplementierung

Zur Generierung von Sprachimplementierungen wurden im Laufe der Zeit viele Systeme und Ansätze entwickelt, so dass an dieser Stelle nicht alle im Detail betrachtet werden können. Darum wurde ein repräsentatives Spektrum der für mich relevanten Ansätze ausgewählt.

Die nachfolgend vorgestellten Ansätze nähern sich der Aufgabe auf sehr unterschiedliche Weise und setzen auch sehr unterschiedliche Lösungsmethoden ein. Die Ansätze unterscheiden sich vor allem darin

- ob sie mit graph- oder baumbasierten Methoden arbeiten,
- ob sie Strukturediteren oder „intelligente“ freie Editoren generieren und
- wie eingeschränkt die grafische Repräsentation ist.

Nachfolgend werden zunächst die baumbasierten und dann die graphbasierten Systeme vorgestellt. PSG, GIGAS und VPE beschränken sich auf Bäume, während MetaEdit+ ein breites Spektrum an Syntax-Konstrukten für Baum- und Nicht-Baumkanten besitzt. Der SRG-ASG-Ansatz sowie DiaGen II basieren auf Graph-Grammatiken, die Baumkanten keine besondere Bedeutung beimessen. Die letzten beiden Ansätze sind auch für freie Editoren mit integriertem Parser geeignet, wohingegen alle anderen Systeme Strukturediteren implementieren.

Bezüglich der Einschränkungen der grafischen Darstellung gibt es große Unterschiede. PSG ist auf textuelle Sprachen spezialisiert. GIGAS und VPE eignen sich vor allem für schachtelungsbasierte Sprachen. MetaEdit+ schränkt die Darstellungsvielfalt auf graphartige, UML-ähnliche Sprachen ein. Der SRG-ASG-Ansatz und DiaGen II sind für eine relativ große Klasse visueller Sprachen geeignet.

2.5.1 PSG

PSG [6, 5] ist ein Generator, der interaktive Programmierumgebungen für textuelle Sprachen generiert. Die generierten Umgebungen besitzen einen Strukturediteor und optional einen Interpretierer. Obwohl die Strukturediteoren einen Modus haben, in dem Programmfragmente frei eingegeben werden

können, sind sie nicht mit „intelligenten“ Editoren (vgl. Abschnitt 2.1.2) vergleichbar, weil vor jeder weiteren Aktion das eingegebene Fragment in eine strukturelle Repräsentation umgewandelt werden muss.

Eine PSG-Spezifikation besteht aus den drei Teilen (1) Syntax, (2) Kontextbedingungen und (3) dynamische Semantik. Der erste Teil ist die wesentliche Grundlage für den Strukturediteor. Der zweite Teil beschreibt die statische Semantik der Sprache. Sie wird auch für den Strukturediteor benötigt, um semantische Fehler bei der Programmerstellung verhindern oder identifizieren zu können. Der dritte Teil wird nur für den Interpretierer benötigt. Nachfolgend gehe ich auf den ersten Spezifikationsteil genauer ein, da er besonders wichtig für meine Arbeit ist.

Die Definition der Syntax besteht aus (1) der lexikalischen Struktur, (2) der abstrakten Syntax, (3) der konkreten Syntax, (4) der Format-Syntax und (5) Spezifikationen für Beschriftungen und Menüs. Die abstrakte Syntax definiert die Struktur des abstrakten Strukturbaums, der als interne Programmrepräsentation dient. Die konkrete Syntax wird benötigt, um ein frei eingegebenes Programmfragment in einen Strukturbaum zu übersetzen. Die Format-Syntax spezifiziert, wie ein abstrakter Strukturbaum in eine konkrete Darstellung transformiert wird.

Der Kern der abstrakten Syntax ist eine Baumgrammatik, die aber um spezielle Konstrukte zur Klassen- und Listenbildung erweitert wurde. Es gibt drei verschiedene Arten von Grammatik-Regeln. NODE-Regeln wie z.B. „`NODE assign :: var expr`“ entsprechen „normalen“ Grammatikregeln. Im Strukturbaum hat ein `assign`-Symbol die Unterknoten `var` und `expr`. LIST-Regeln wie „`LIST statlist = stat+`“ spezifizieren, dass `statlist`-Knoten im Strukturbaum beliebig viele `stat`-Unterknoten besitzen können. Schließlich beschreiben CLASS-Regeln wie „`CLASS type = integer, real, bool`“ syntaktische Alternativen. Durch die Unterscheidung dieser drei Regelarten lässt sich das Editieren komfortabler gestalten.

Die Format-Syntax spezifiziert, wie ein abstrakter Strukturbaum in eine konkrete Darstellung transformiert wird, die als Schnittstelle zum Sprachanwender dient. Zu jeder Regel der abstrakten Syntax gibt es in der Format-Syntax eine Regel, die die Darstellung entsprechender Strukturfragmente beschreibt. Für die oben genannte abstrakte Regel „`NODE assign :: var expr`“ könnte die Format-Syntax z.B. die Regel „`assign => ! var ass expr`“ enthalten. Diese spezifiziert, dass vor einer Zuweisung ein Zeilenumbruch erfolgen soll („!“) und dass zwischen der Variablen `var` und dem Ausdruck `expr` ein Zuweisungszeichen (`ass`) stehen soll. Neben dem Ausrufe-

zeichen als Formatierungsanweisung für Zeilenumbrüche besitzt die Spezifikationssprache auch Konstrukte für Einrückungen und bedingte Formatierung.

Relevanz im Kontext dieser Arbeit Das PSG-System ist sozusagen ein Urahn von DEViL. Das Grundkonzept der Syntaxdefinition ist durchaus vergleichbar, lediglich Querbeziehungen lassen sich in PSG nicht syntaktisch modellieren. DEViL ist im Gegensatz zu PSG auf visuelle Repräsentationen spezialisiert. Das Konzept der Format-Syntax ist aber trotzdem wichtig für DEViL, denn DEViL enthält eine vergleichbare Spezifikationssprache, damit textuelle Teilrepräsentationen genau so einfach wie in PSG spezifiziert werden können (siehe Abschnitt 4.5).

2.5.2 GIGAS

Das GIGAS-System [15] beschränkt sich wie PSG auf baumstrukturierte Sprachen. Auch GIGAS basiert auf Baum-Grammatiken, erlaubt aber im Gegensatz zu PSG auch grafische Repräsentationen. Der Spezifikationsmechanismus unterscheidet zwei Ebenen: Zur Lösung elementarer Gleichungen grafischer Attribute werden attributierte Grammatiken verwendet. Die grundlegende Struktur der Darstellung wird auf einer höheren Ebene durch die Spezifikationssprache GSL (*graphical specification language*) definiert. Das Spezifikationsmodell von GSL basiert auf geschachtelten Rechtecken und wurde vom Textsatzsystem TeX [34] inspiriert.

Abbildung 2.16a zeigt die Spezifikation der grafischen Darstellung für die abstrakte Produktion „`Integral: Exp ::= Exp Exp Exp Var`“. Das Konzept der Spezifikation lässt sich an Abbildung 2.16b erkennen. Die Darstellung besteht auf oberster Ebene aus den zwei Rechtecken `Symbol` und `Operands`, die das Integral-Symbol und die Operanden enthalten. Das Rechteck `Operands` enthält wiederum Rechtecke für die obere und untere Integrations-Grenze sowie für den Mittelteil der Darstellung. Der Mittelteil besteht schließlich aus dem Ausdruck `Exp`, dem Buchstaben „D“ sowie der Integrationsvariablen `y`. Die Anordnung von Rechtecken gleicher Ebene wird durch die Layoutmuster „horizontal“, „vertikal“, „horizontal zentriert“ und „vertikal zentriert“ festgelegt. Die Layoutmuster werden intern auf eine Menge von Attributberechnungen abgebildet. Diese Berechnungen können je nach Bedarf durch benutzerdefinierte Berechnungen ergänzt oder überschrieben

werden. Solche Ergänzungen sind in Abbildung 2.16a in geschweiften Klammern notiert.

Relevanz im Kontext dieser Arbeit GIGAS basiert auf dem gleichen zweistufigen Spezifikationskonzept wie DEViL und dessen Vorgänger VL-Eli. GIGAS besitzt aber nur ein festes, sehr spezialisiertes Layoutmuster, während DEViL und VL-Eli eine große, erweiterbare Musterbibliothek bereitstellen.

Ein wichtiger Beitrag von GIGAS ist die Unterscheidung zwischen abstrakter Grammatik und Layout-Grammatik. Die Layout-Grammatik in GIGAS beschreibt die Schachtelungsstruktur der Rechtecke und entspricht der Repräsentationsstruktur in DEViL. VL-Eli fehlt diese Unterscheidung. Die Trennung dieser beiden Strukturen in DEViL wird in Abschnitt 4.3 behandelt.

2.5.3 VPE

Das VPE-System [18] hat ein ähnliches Funktionsspektrum wie GIGAS. Auch mit VPE können Strukturediteren für baumstrukturierte Sprachen realisiert werden. In der Spezifikation werden visuelle Grundobjekte beschrieben, die später vom Sprachanwender zu visuellen Ausdrücken zusammengesetzt werden können.

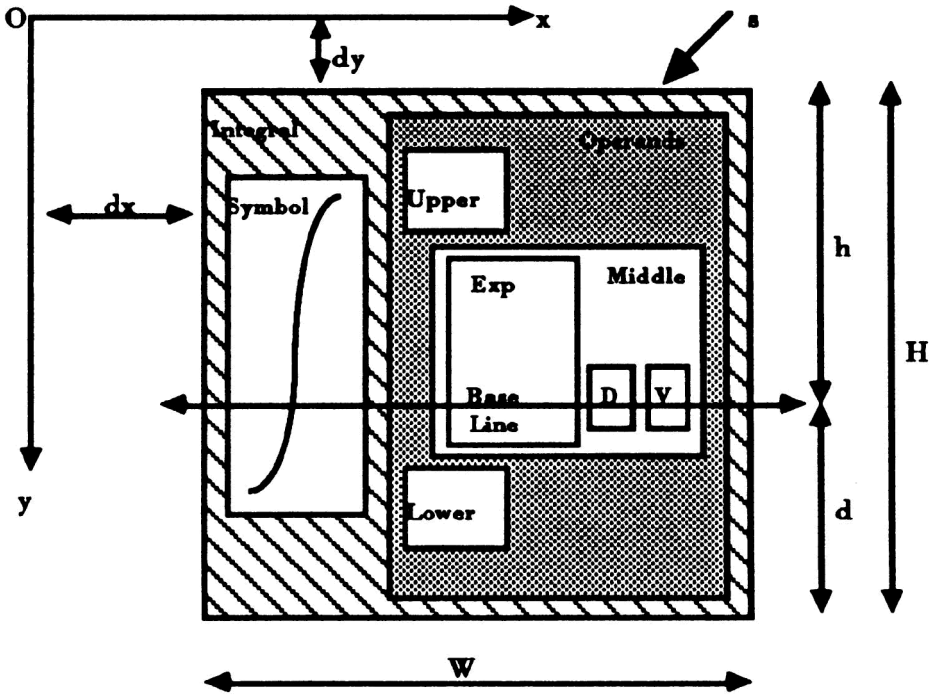
Abbildung 2.17 zeigt die Spezifikation der bedingten Anweisung in Nassi-Shneiderman Diagrammen. Die Spezifikation enthält Vektorgrafikprimitive und zusätzliche rechteckige Bereiche, so genannte Container. Zu jedem Container gehören Informationen, welche anderen visuellen Objekte der Sprachanwender dort einfügen darf und welche Layouteigenschaften daraus resultieren. Bezüglich der abstrakten Struktur entsprechen die Objektbeschreibungen den Produktionen einer Baum-Grammatik.

Eine interessante Eigenschaft des VPE-Systems ist das Prinzip des automatischen Layouts. Die linke Seite von Abbildung 2.18 zeigt zwei visuelle Grundobjekte, die zur Konstruktion von Nassi-Shneiderman Diagrammen benötigt werden. Wird die Schleife in die bedingte Anweisung eingefügt, ergibt sich die Darstellung auf der rechten Seite. Wie man erkennt, sind die visuellen Objekte trotz der Spezifikation mit festen Koordinaten nicht starr, sondern werden skaliert und gedehnt.

VPE-Spezifikationen sind recht kompakt und zumindest für geschachtelte Darstellungen ist die Ausdruckskraft des Spezifikationskonzepts recht hoch.

```
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 2.16: Spezifikation der grafischen Darstellung eines Integrals in GIGAS (aus [15])

```
(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 2.17: VPE-Spezifikation der bedingten Anweisung in Nassi-Shneiderman Diagrammen

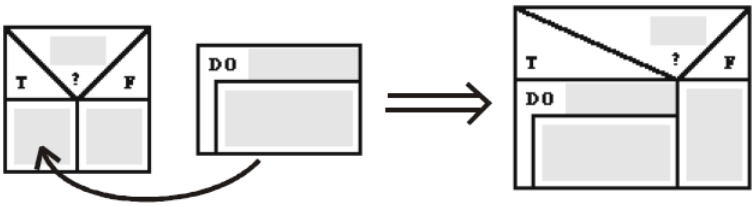


Abbildung 2.18: Layoutkonzept des VPE-Systems

Zur Spezifikation von Nassi-Shneiderman Diagrammen werden beispielsweise nur etwa 210 Zeilen Quelltext benötigt. Für den praktischen Einsatz sind aber vor allem die vom System bereitgestellten Editieroperationen ungenügend, da nicht einmal Listen explizit modelliert werden können. Stattdessen müssen sie durch Rekursion nachgebildet werden.

Relevanz im Kontext dieser Arbeit VPE hat den Entwurf der so genannten Generischen Vektorgrafik-Zeichnungen inspiriert, die in Abschnitt 4.4.1 vorgestellt werden. Im Gegensatz zu VPE werden Generische Vektorgrafik-Zeichnungen in DEViL allerdings visuell spezifiziert und basieren im Detail auf einem anderen Layoutmechanismus.

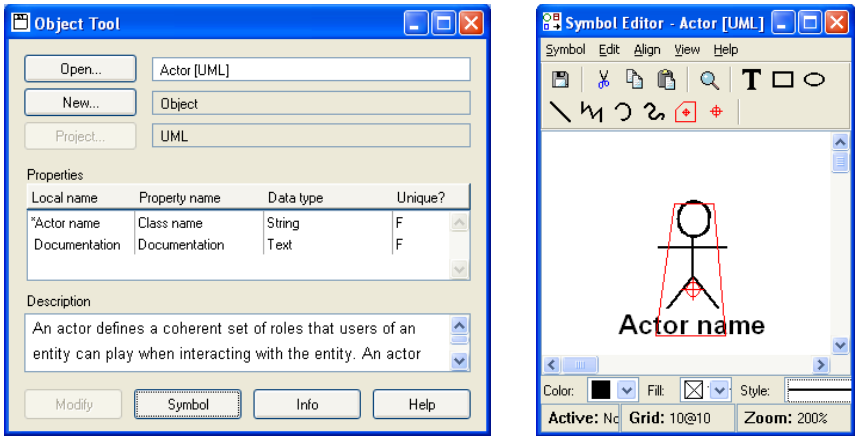
2.5.4 MetaEdit+

MetaEdit+ [10] ist ein kommerzielles Produkt der Firma *MetaCase*. Es vertritt die Klasse der so genannten Meta-CASE Werkzeuge, d.h. Systeme, die grafische CASE-Werkzeuge[3] relativ schnell zu entwickeln gestatten. CASE-Werkzeuge basieren auf textuellen oder visuellen Sprachen, mit denen Software modelliert oder spezifiziert werden kann.

MetaEdit+ basiert im Gegensatz zu den oben vorgestellten Werkzeugen auf einem Syntax-Kalkül, das nicht auf Bäume beschränkt ist. In MetaEdit+ wird das so genannte GOPRR-Modell verwendet. GOPRR steht für Graph-Object-Property-Relationship-Role. Diese fünf so genannten Metatypen bilden die Grundlage der Syntax-Spezifikation. Jedes Konstrukt der zu modellierenden Sprache basiert auf einem dieser Metatypen. Der Metattyp „Object“ modelliert Konstrukte, die mit anderen Konstrukten komplexe Beziehungen eingehen können. Der Metatyp „Relationship“ modelliert Beziehungen zwischen Objekten. Spracheigenschaften, die den Zusammenhang zwischen Objekten und Beziehungen betreffen, besitzen den Metatyp „Role“. Durch den Metatyp „Property“ werden Eigenschaften anderer Konstrukte modelliert und „Graph“ dient schließlich dazu, komplexere Strukturen zusammenzufassen, um z.B. Verfeinerungen und Dekompositionen ausdrücken zu können.

Wie die Erklärung des Akronyms GOPRR bereits vermuten lässt, ist diese Form der Syntaxbeschreibung stark auf graphartige Sprachen spezialisiert, die in CASE-Werkzeugen besonders häufig vorkommen. Für anders

[3] CASE steht für *Computer-Aided Software Engineering*



(a) Object Tool (b) Symbol Editor

Abbildung 2.19: Spezifikationswerkzeuge im MetaEdit+ System

strukturierte Sprachen ist diese Beschreibungsform weniger gut geeignet. Es gibt allerdings auch Meta-CASE Werkzeuge, die auf nicht so spezialisierten Strukturbeschreibungssprachen wie ER-Diagrammen oder UML-Klassendiagrammen basieren [13, 23].

Beim Entwurf von MetaEdit+ wurde Wert darauf gelegt, das System einfach anwendbar zu machen. Daher existiert eine grafische Benutzungsschnittstelle für den Sprachentwickler. Für alle Metatypen gibt es dialogbasierte Spezifikationswerkzeuge. Abbildung 2.19a zeigt das so genannte *Object Tool* zur Definition von Sprachkonstrukten basierend auf dem Metatyp „Object“. Mit dem zugehörigen *Symbol Editor* (siehe Abbildung 2.19b) können konkrete Repräsentationen für die Konstrukte definiert werden. Dazu können Vektorgrafik-Primitive, Platzhalter für Attribute und Konnektoren für Linienverbindungen grafisch zusammengestellt werden.

Relevanz im Kontext dieser Arbeit MetaEdit+ repräsentiert Systeme mit modellbasierten Syntax-Kalkülen, die nicht auf Baumstrukturen beschränkt sind. Sie schließen die Lücke zwischen Syntax-Kalkülen, die wie bei GIGAS und PSG auf Bäume beschränkt sind und solchen, die wie beim SRG-ASG-Ansatz und DiaGen II auf Graph-Grammatiken basieren. DEViL ist prinzipiell der Kategorie von MetaEdit+ zuzuordnen, wobei DEViL aber größeres Gewicht auf die Baumkanten legt.

MetaEdit+ ist auch interessant, da hier besonders auf die Benutzerfreundlichkeit geachtet wurde. Zugunsten einer einfacheren Benutzung wurde der reali-

sierbare Sprachumfang eingeschränkt und eine grafische Benutzungsschnittstelle zur Verfügung gestellt. Besonders zu erwähnen ist der oben vorgestellte *Symbol Editor*, der mit den in Abschnitt 4.4.1 beschriebenen Generischen Vektorgrafik-Zeichnungen verwandt ist.

Da MetaEdit+ auf Sprachen spezialisiert ist, die ähnlich wie UML komplexe Systeme beschreiben, können auch in MetaEdit+ semantische Objekte mehrfach in Sichten auftreten. Auch können mehrere gleichartige Sichten auf die semantische Struktur existieren, deren Layout unabhängig voneinander definiert werden kann. MetaEdit+ stellt daher auch Mechanismen zur Verfügung, um die Wirkung von Benutzeraktionen auf die semantische Struktur zu definieren und die Konsistenz der Sichten zu gewährleisten. Genauer wird hierauf in Abschnitt 3.3.2 eingegangen.

2.5.5 Der SRG-ASG-Ansatz

Sprachimplementierungen basierend auf dem so genannten SRG-ASG-Ansatz von Rekers und Schürr [48] erlauben sowohl freies als auch strukturiertes Editieren. Es werden drei strukturelle Abstraktionen unterschieden.

- Der *abstract syntax graph* (ASG) ist ein Graph, der die logische Struktur des visuellen Programms repräsentiert.
- Der *spatial relations graph* (SRG) ist ein Graph, dessen Knoten die grafischen Objekte und dessen Kanten die räumlichen Relationen wie Berührung oder Enthaltensein modellieren.
- Unter *physikalischem Layout* wird eine Menge grafischer Grundelemente mit Eigenschaften wie Position oder Farbe verstanden, die unmittelbar die grafische Repräsentation beschreiben.

Der Zusammenhang dieser Strukturen ist in Abbildung 2.20 dargestellt. Ein Beispiel für diese Strukturen ist in Abbildung 2.21 zu sehen.

Zur Spezifikation von ASG und SRG werden gekoppelte Graphgrammatiken verwendet. Durch die einzelnen Graphgrammatiken ist die Syntax der Graphen definiert. Durch die Kopplung wird der Zusammenhang beider Strukturen hergestellt. Die korrespondierenden Symbole werden dazu durch zusätzliche Kanten verbunden, die die „Repräsentierter-Repräsentant“-Beziehung

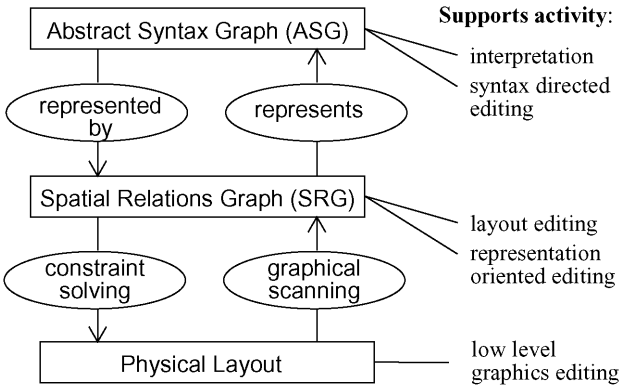
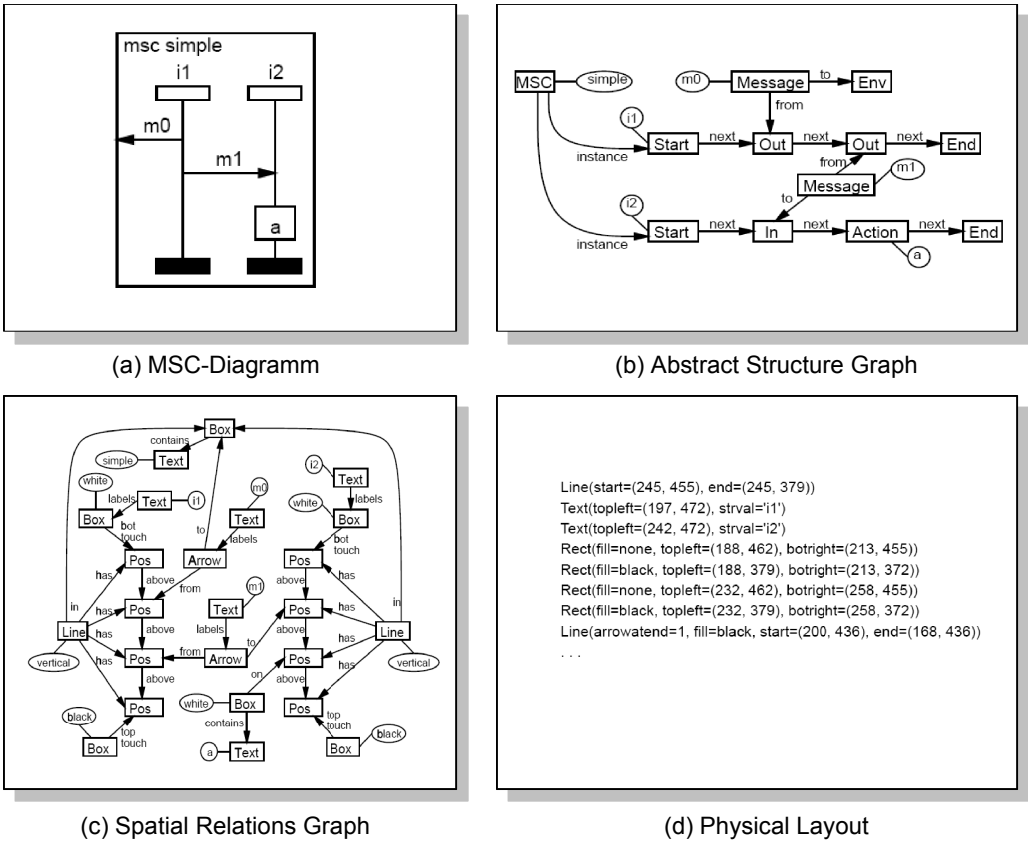


Abbildung 2.20: Die drei Repräsentationsebenen des SRG-ASG-Ansatzes (aus [48])



(a) MSC-Diagramm

(b) Abstract Structure Graph

(c) Spatial Relations Graph

(d) Physical Layout

Abbildung 2.21: Die drei Repräsentationsebenen des SRG-ASG-Ansatzes am Beispiel von *Message Sequence Charts* (aus [48])

modellieren. Basierend auf dieser Korrespondenz kann eine Änderung der einen Struktur auf die entspreched andere übertragen werden.

Der SRG kann als Constraint-Netzwerk betrachtet werden, auf dessen Grundlage das physikalische Layout berechnet werden kann. Ein Constraint-System besteht aus einer Menge von Variablen und einer Menge mathematischer Gleichungen und Ungleichungen über diesen Variablen. In der hier betrachteten Anwendung sind die Variablen Layoutattribute wie Positionen oder Objektgrößen und die Gleichungen und Ungleichungen ergeben sich aus den räumlichen Relationen des SRG wie Berührung oder Enthaltensein. Die Lösung eines Constraint-Netzwerks, die von so genannten Constraint-Solvern wie z.B. Parcon [20] oder QOCA [35] automatisch berechnet werden kann, liefert ein konkretes Layout für die Darstellung. In umgekehrter Richtung kann durch Analyse der räumlichen Relationen eines physikalischen Layouts auch ein SRG abgeleitet werden.

Zusammengenommen erlaubt der Ansatz sowohl strukturiertes als auch freies Editieren. Beim strukturierten Editieren wird der ASG geändert und die Änderung durch Graph-Kopplung und grafische Constraints auf das physikalische Layout übertragen. Bei einer freien Änderung des physikalischen Layouts baut ein grafischer Scanner einen SRG auf, der dann entsprechend der SRG-Grammatik parsiert wird. Basierend hierauf kann durch die Kopplung der Graphgrammatiken der ASG erstellt werden.

Der SRG-ASG-Ansatz wurde nicht vollständig implementiert. Es gibt aber einige Systeme, die diesen Ansatz zumindest teilweise umsetzen. Zwei bekannte Systeme, die Struktur-Editoren basierend auf diesem Ansatz generieren, sind PROGRES [59] und GenGEd [7]. In beiden Systemen werden Editieroperationen durch Graph-Transformationsregeln beschrieben und beide basieren auf Constraint-Netzwerken zur Layoutberechnung. Im Detail unterscheiden sich die Systeme. GenGEd basiert auf algebraischen Graphstrukturen und erlaubt es, Editoren interaktiv und grafisch zu spezifizieren.

Relevanz im Kontext dieser Arbeit Auch in diesem Ansatz finden sich die gleichen strukturellen Abstraktionen, wie sie in Abbildung 2.8 auf Seite 38 aufgeführt sind. Die Spezifikation der Syntax basiert beim SRG-ASG-Ansatz allerdings auf Graph-Grammatiken, die nicht direkt mit dem in DEViL eingesetzten Konzept verwandt sind.

Beachtenswert ist die Spezifikationsmethode zur Definition der grafischen Darstellung. Das hier verwendete Konzept, die grafische Darstellung durch

die Abbildung auf ein Constraint-Netzwerk zu beschreiben, ist prinzipiell bestechend und wird dementsprechend häufig verwendet. Allerdings ist dies, wie ich in Abschnitt 4.2.4 ausführen werde, mit Einschränkungen verbunden.

Interessant sind auch die Anmerkungen der Autoren zur Benutzerfreundlichkeit der resultierenden Strukturediteren. Sie führen aus, dass das in Abbildung 2.21 gezeigte Beispiel „Message Sequence Charts“ bei einigen Editieroperationen die Selektion von bis zu vier Kontexten erforderlich macht. Da ein entsprechender Strukturediteur nur schwer bedienbar wäre, führen sie eine abgeschwächte SRG-Grammatik ein, die auch syntaktisch inkonsistente Zwischenzustände erlaubt. Um aus solchen Programmen einen ASG abzuleiten, muss der SRG zunächst anhand der ursprünglichen SRG-Grammatik parsiert werden.

2.5.6 DiaGen II

DiaGen II [37] ist primär auf die Generierung von Sprachimplementierungen mit freiem Editor und integriertem Parser zugeschnitten. Die Editoren können durch Zusatzspezifikationen aber zu hybriden Editoren ausgebaut werden, die auch strukturierte Editieroperationen erlauben.

Die Struktur eines DiaGen-Editors ist in Abbildung 2.22 dargestellt. Rechtecke stellen funktionale Bestandteile und Ovale Datenstrukturen dar. Der wichtigste Verarbeitungspfad erstreckt sich vom Diagramm über das Hypergraphmodell, das reduzierte Hypergraphmodell und die Ableitungsstruktur bis zur semantischen Repräsentation. Die Kette realisiert die Übersetzung eines frei konstruierten Programms in eine semantische Struktur. Die in Abbildung 2.8 auf Seite 38 gezeigte Verarbeitungskette wird hier also in umgekehrter Richtung durchlaufen.

Unter „Diagramm“ ist hier eine Diagramm-Repräsentation zu verstehen, die frei positionierbare, aber sprachspezifische Grafikprimitive enthält. Diese Abstraktionsebene ist grob mit der Vektorgrafik-Repräsentation aus Abbildung 2.8 vergleichbar.

Das Hypergraphmodell repräsentiert das Diagramm auf direkte Weise als Hypergraph. Hier werden bestimmte Relationen der grafischen Primitive wie Berührungen oder Schachtelungen bereits strukturell repräsentiert. Die Repräsentation entspricht grob der abstrakten Repräsentation aus Abbildung 2.8. Den strukturellen Elementen können des Weiteren auch Attribute wie

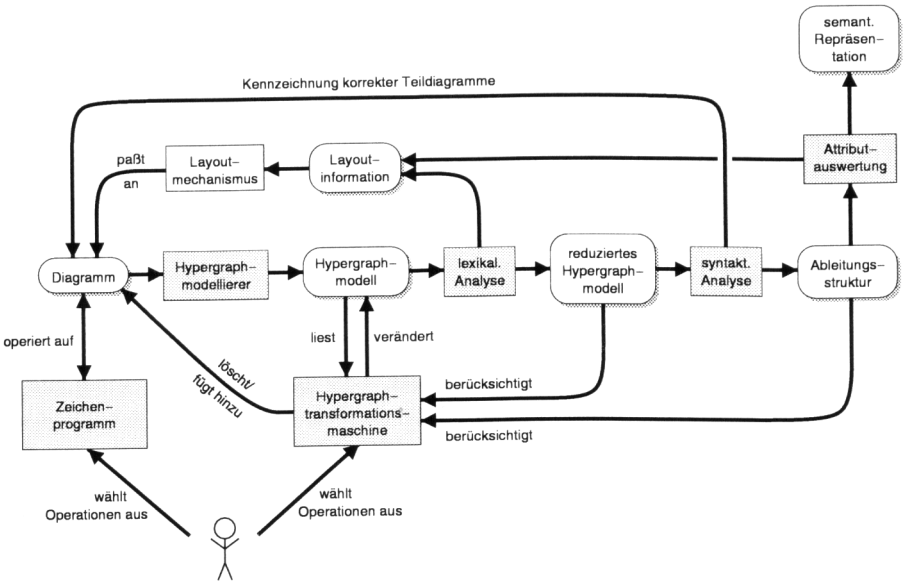


Abbildung 2.22: Die Struktur eines DiaGen Editors (aus [37])

Koordinaten zugeordnet sein, die weitere Layouteigenschaften der entsprechenden grafischen Primitive modellieren. Dies ist notwendig, um in einem späteren Schritt z.B. die lineare Ordnung von Elementen erkennen zu können.

Der Übergang vom Hypergraphmodell zum reduzierten Hypergraphmodell entspricht in gewisser Weise der lexikalischen Analyse bei der Verarbeitung textueller Programme. Prinzipiell werden in diesem Schritt bestimmte einfache „Muster“ im Hypergraphmodell erkannt und durch explizite Repräsentationen ersetzt. Dies wird durch so genannte Reduktionsregeln spezifiziert, die eng mit Graphtransformationsregeln verwandt sind. Durch die Reduktionsregeln können u.a. auch die vorher nur implizit vorhandenen Reihenfolge-Informationen durch eine strukturelle Repräsentation ersetzt werden.

Das reduzierte Hypergraphmodell kann basierend auf einem Hypergraph-Parser und Attributberechnungen in eine semantische Repräsentation überführt werden. Die Hypergraph-Grammatik beschreibt hierbei die Menge der syntaktisch gültigen Programme. Inkorrekte Teildiagramme werden im Editor automatisch gekennzeichnet.

Durch zusätzliche Spezifikationen können dem Editor strukturierte Editieroperationen hinzugefügt werden. Die Spezifikation der Editieroperationen erfolgt durch Hypergraph-Transformationsregeln, die durch Kontrollprogramme koordiniert werden. Die Hypergraph-Transformationsregeln arbeiten auf dem Hypergraphmodell, können aber auch auf die Informationen des

reduzierten Hypergraphmodells und die Ableitungsstruktur zugreifen. Damit nach einer strukturellen Änderung die Darstellung aktualisiert werden kann, muss weiterhin spezifiziert werden, wie das Layout eines Diagramms berechnet bzw. angepasst wird. Dies kann entweder durch Spezifikation von Layout-Constraints oder durch Programmierung eines anwendungsspezifischen Layoutmoduls erfolgen.

Relevanz im Kontext dieser Arbeit Interessant an diesem Ansatz ist die Umsetzung der in Abbildung 2.8 dargestellten Transformationskette in umgekehrter Richtung. Man erkennt, dass auch hier eine Reihe von Zwischenrepräsentationen verwendet werden, die natürlich aufgrund der umgekehrten Verarbeitungsrichtung anderen Randbedingungen unterliegen und daher nicht direkt mit denen in Abbildung 2.8 vergleichbar sind.

In Bezug auf DEViL sind vor allem die Spezifikationsmechanismen für das automatische Layout und insbesondere die daraus gewonnenen Erfahrungen interessant. Beispielsweise wird erwähnt, dass das Layout größerer Diagramme bei constraint-basierter Layoutspezifikation sehr lange dauern kann. Die constraint-basierte Spezifikationsmethode wird daher vor allem als Methode zur Umsetzung von Prototypen verstanden [37, S. 136]. Interessant ist auch die Bemerkung, dass ein „direct manipulation“ Interaktionsmechanismus im Vergleich zu der in DiaGen II eingesetzten Variante den Spezifikationsaufwand erheblich vergrößern würde [37, S. 160]. Dies unterstreicht die in dieser Arbeit vertretene These, dass visuelle Muster auch durch deren Fähigkeit, Interaktionsmechanismen zu kapseln, einen wertvollen Beitrag leisten.

2.6 Zusammenfassung

Im ersten Teil dieses Kapitels wurde ein Überblick über das Gebiet der visuellen Sprachen und deren Implementierungsformen gegeben. Vor allem wurde deutlich, dass es in visuellen Sprachen eine sehr große Bandbreite visueller Ausdrucksmittel gibt, die dem Sprachentwickler neue Möglichkeiten bieten und das Benutzen der Sprache einfacher und angenehmer machen können. Strukturediteren können die Benutzung einer Sprache erleichtern, indem sie die Programmkonstruktion und den Umgang mit komplexen Strukturen vereinfachen. Um hier kein Potenzial zu verschenken sollten Generatoren für Strukturediteren daher besondere Mechanismen zum Umgang mit Strukturen bereitstellen.

Mit dem in Abschnitt 2.3 eingeführten Modell konnten die Eigenschaften von Strukturediteren in verschieden Dimensionen klassifiziert werden und mit einem erweiterten Modell wurden die Grundlagen zur Implementierung von Strukturediteren gelegt. Besonders hervorzuheben sind die Abstraktionen „semantische Struktur“, „editierbare Struktur“ und „Repräsentationsstruktur“, die in ähnlicher oder gleicher Form auch in anderen Werkzeugen auftreten und in den folgenden Kapiteln dieser Arbeit eine zentrale Rolle spielen.

Im letzten Teil wurden verwandte Systeme vorgestellt, die für den Entwurf von DEViL von entscheidender Bedeutung sind. In VL-Eli basiert die Spezifikation der grafischen Darstellung auf visuellen Mustern, in GIGAS auf geschachtelten Rechtecken und in MetaEdit+ auf einer einfachen visuellen Beschreibungssprache. In VPE kommt eine anspruchsvollere textuelle Beschreibungssprache zum Einsatz, die auf Containern für Unterelemente basiert. Der SRG-ASG-Ansatz setzt genau wie DiaGen II Constraint-Netzwerke ein, und PSG hat eine Spezialsprache zur Spezifikation textueller Repräsentationen. All diese Techniken lassen sich in abgewandelter Form auch in DEViL wiederfinden und werden dort zu einem flexiblen Gesamtkonzept vereint.

3 Editierbare und semantische Struktur

Inhalt

Abstrakte Strukturen bilden die Grundlage eines jeden Strukturediteors. In diesem Kapitel werden die in DEViL realisierten Konzepte und Methoden zum Umgang mit abstrakten Strukturen vorgestellt. Bereits im zweiten Kapitel wurde in Abbildung 2.6 auf Seite 26 ein funktionales Modell zur Implementierung von Strukturediteoren vorgestellt. Dort wurden die beiden Abstraktionen „editierbare Struktur“ und „semantische Struktur“ eingeführt. Hier wird nun näher auf den Zusammenhang beider Strukturen eingegangen. Die Methode zur Kopplung beider Strukturen ist ein zentraler Teil dieses Kapitels.

Im funktionalen Modell in Abbildung 2.6 werden die abstrakten Strukturen strikt von der Umsetzung der grafischen Darstellung getrennt. Aus diesem Grund sind die nachfolgend vorgestellten Methoden unabhängig von der Spezifikationsmethode zur Beschreibung der grafischen Darstellung. Allerdings muss gewährleistet sein, dass die editierbare Struktur zu der grafischen Darstellung, der sie als Grundlage dient, passt. Das bedeutet z.B., dass jedes elementare Objekt der grafischen Darstellung durch ein Objekt der editierbaren Struktur repräsentiert wird, dass Schachtelung und andere Arten von Teil-Ganzes-Beziehungen durch dafür geeignete Relation repräsentiert werden, dass in der editierbaren Struktur benutzerdefinierte Layoutvorgaben gespeichert werden können und dass die umzusetzenden Editiermechanismen mit der Editor-Syntax zusammenpassen. Kurz: Die editierbare Struktur soll für die konkrete Repräsentation maßgeschneidert sein.

Damit dieses Entwurfsziel nicht mit anderen Entwurfszielen wie Redundanzfreiheit oder Stabilität der Syntax in Konflikt gerät, wurde zwischen der editierbaren und der semantischen Struktur differenziert. Die semantische Struktur modelliert den relevanten Informationsgehalt des Programms und braucht die konkrete Repräsentation in keiner Weise zu berücksichtigen. Allerdings ergibt sich durch die Trennung beider Strukturen die Aufgabe, sie konsistent zu halten.

Bevor die Kopplung von Strukturen diskutiert werden kann, wird im ersten Abschnitt eine Sprache vorgestellt, um die Syntax von Strukturen zu definieren. Hierauf basieren alle weiteren Teile der Arbeit. Die Sprache wurde speziell für das in dieser Arbeit vorgestellte Gesamtkonzept entwickelt und vereinigt die Vorzüge kontextfreier Grammatiken, UML-basierter Modellierungskonzepte und objektorientierter Programmiersprachen.

Das Kalkül für die Syntax-Spezifikation ist sowohl für die editierbare als auch für die semantische Struktur geeignet. Im Kontext der editierbaren Struktur

ist allerdings die Wechselwirkung mit der Benutzungsschnittstelle zu berücksichtigen. Hierauf wird in Abschnitt zwei genauer eingegangen.

Im dritten Abschnitt wird schließlich die Methode zur Kopplung von editierbarer und semantischer Struktur vorgestellt. Auch diese Methode wurde speziell für den hier betrachteten Anwendungszweck entworfen. Die Herausforderung bestand aber darin, einen Kopplungsmechanismus zu entwerfen, der für typische Anwendungsfälle den Spezifikationsaufwand minimiert aber trotzdem nicht bei komplexeren Problemstellungen versagt.

Zum Entwurf des Kopplungsmechanismus wurden typische Einsatzszenarien erarbeitet, indem reale visuelle Sprachen wie UML untersucht wurden. Im vierten Abschnitt werden diese Szenarien vorgestellt und es wird gezeigt, wie sie sich basierend auf dem vorher eingeführten Kopplungsmechanismus umsetzen lassen. Diese Szenarien sind über diese Arbeit hinaus auch für zukünftige Arbeiten nützlich, um die Wirksamkeit alternativer Kopplungsmechanismen zu evaluieren.

Den Abschluss dieses Kapitels bilden einige allgemeine Bemerkungen zur hier vorgestellten Methode. Sie betreffen das Einsatzspektrum, sinnvolle Erweiterungen sowie die Grenzen des Mechanismus.

3.1 Spezifikation abstrakter Strukturen in DEViL

Nachfolgend wird beschrieben, wie die Syntax abstrakter Strukturen in DEViL definiert wird. Der Begriff „abstrakte Struktur“ ist hier als Überbegriff von editierbarer und semantischer Struktur zu verstehen. Die nachfolgend vorgestellte Spezifikationssprache ist für beide Strukturen gleichermaßen geeignet.

Zunächst werden die Anforderungen an das Spezifikationskonzept diskutiert. Dann wird die Spezifikationssprache DSSL (*DEViL Structure Specification Language*) eingeführt, mit der die Syntax abstrakter Strukturen spezifiziert werden kann. Ferner wird eine Notation für Pfadausdrücke vorgestellt, die sowohl den nachfolgenden Abschnitten als Grundlage dient als auch die Spezifikation von Konsistenzbedingungen erlaubt. Den Abschluss bildet eine Beschreibung der Funktionen zur Änderung der Struktur, die ebenfalls in nachfolgenden Teilen der Arbeit benötigt werden.

3.1.1 Anforderungen an das Spezifikationskonzept

Es gibt viele Methoden zur Syntax-Spezifikation. Einige der wichtigsten sind kontextfreie Grammatiken, UML-Klassendiagramme, XML-Schema-Definitionen und Graph-Grammatiken. Um den Entwurf des Spezifikationskonzepts nachvollziehbar beschreiben zu können, sollen zunächst die Anforderungen und Randbedingungen diskutiert werden, die für diese Arbeit eine Rolle spielten. Zu den wichtigsten Anforderungen zählten

- die Abbildbarkeit der Syntax auf Baum-Grammatiken, die die Anwendung visueller Muster erlauben,
- die Kompatibilität des Konzepts zu grafischen und interaktiven Anforderungen der Sprachimplementierung,
- die Eignung für visuelle *und* textuelle Repräsentationen,
- die Generierbarkeit eines Strukturediteors aus der „nackten“ Syntax,
- die Förderung von Stabilität und Aufwärtskompatibilität spezifizierter Strukturen,
- die Übersichtlichkeit der Syntax für den Sprachentwickler und
- die Beschränkung der Modellierungssprache auf wenige Grundkonzepte, um sowohl die Definition als auch den Umgang mit der Syntax zu vereinfachen.

Das wichtigste Ziel dieser Arbeit ist die Weiterentwicklung der Methode der visuellen Muster. Daher fordert der erste Spiegelpunkt, dass das Konzept der Syntax-Spezifikation mit der Umsetzungsmethode der visuellen Muster kompatibel ist. Aus den in Kapitel 4 dargestellten Gründen habe ich mich für Baum-Grammatiken als Realisierungsgrundlage für visuelle Muster entschieden. Daher ist es wichtig, dass sich die abstrakte Syntax leicht auf Baum-Grammatiken abbilden lässt.

Beim zweiten Punkt geht es um die Syntax im Kontext der editierbaren Struktur. In diesem Kontext beschreibt die Syntax die Grundbausteine visueller Repräsentationen sowie deren Kombinierbarkeit. Darum ist es wichtig, dass diese Modellierung „kompatibel“ mit den grafischen und interaktiven Eigenschaften von Repräsentationen ist, dass z.B. jedes elementare grafische Objekt

durch einen Knoten in der abstrakten Struktur modelliert werden kann und dass sich die grafischen Interaktionsmechanismen direkt auf strukturelle Änderungsfunktionen abbilden lassen. Beispielsweise spielen lineare Ordnungen in visuellen Repräsentationen eine wichtige Rolle, so dass in der Syntax ein Konstrukt zur Listenbildung vorhanden sein sollte. Auf die Beziehung zwischen Syntax und Benutzungsschnittstelle wird in Abschnitt 3.2 explizit eingegangen.

Da auch in visuellen Sprachen textuelle Teilrepräsentationen vorkommen, fordert der dritte Punkt, dass die Syntax für beide Varianten geeignet ist. In anderen Ansätzen werden unterschiedliche Kalküle für die beiden Darstellungsvarianten verwendet. Ich halte aber eine einheitliche Strukturbeschreibungssprache für wichtig, die auch in diesem Punkt von der konkreten Repräsentation abstrahiert.

Der vierte Punkt spiegelt die Erfahrung wider, dass manchmal bereits mit der Umsetzung der semantischen Analyse und Codegenerierung begonnen werden soll, bevor die Implementierung der grafischen Darstellung abgeschlossen ist. Daher sollte auch aus der „nackten“ Syntax bereits ein angemessener Strukturediteor generierbar sein.

Da die Entwicklung von Sprachimplementierungen normalerweise ein inkrementeller Vorgang ist, sollte das Konzept der Syntax-Definition ferner die Stabilität und Aufwärtskompatibilität von Strukturen fördern. „Stabilität“ bedeutet hier, dass eine Änderung der Syntax möglichst wenig ausstrahlt, also lokal begrenzt bleibt. „Aufwärtskompatibilität“ bedeutet in diesem Zusammenhang, dass gespeicherte Strukturen möglichst auch nach Syntax-Änderungen noch weiter verwendbar sein sollten. Das ist wichtig, da gespeicherte Strukturen Programme der visuellen Sprache repräsentieren, in denen häufig erheblicher Aufwand steckt. Falls Strukturen alter Syntax nicht direkt wiederverwendbar sind, sollten sie wenigstens möglichst einfach an die neue Syntax anpassbar sein.

Die Syntax sollte ferner dem Sprachentwickler eine schnelle und gute Übersicht über die modellierte Sprache bieten. Das ist vor allem wichtig, um sich schnell in fremde Spezifikationen einarbeiten zu können.

Schließlich soll sich das Spezifikationskonzept auf möglichst wenige, orthogonale Konzepte beschränken. Das erleichtert nicht nur die Implementierung des Generators, sondern vor allem auch den Umgang mit der Syntax, da weniger Sonder- und Spezialfälle zu berücksichtigen sind.

```
CLASS Statechart {
  states: SUB State*;
  transitions: SUB Transition*;
}

ABSTRACT CLASS State {
  name: VAL VLString;
}

CLASS SimpleState INHERITS State {
}

CLASS XORSuperstate INHERITS State {
  substates: SUB State*;
}

CLASS Transition {
  from: REF State;
  to: REF State;
  label: VAL VLString;
}
```

Abbildung 3.1: DSSL-Syntax für Zustandsdiagramme

3.1.2 Die Spezifikationssprache DSSL

In Abbildung 3.1 ist ein Beispiel für die Spezifikationssprache DSSL (*DEViL Structure Specification Language*) zu sehen. Die Syntax modelliert eine eingeschränkte Klasse von Zustandsdiagrammen. Sofort ins Auge sticht die Ähnlichkeit zu objektorientierten Programmiersprachen wie Java. Hierdurch bekommt der Sprachentwickler direkt einen zutreffenden ersten Eindruck von der Semantik der Sprache.

DSSL basiert auf objektorientierten Modellierungskonzepten wie Klassen, Attributen und Vererbung. Jede nicht-abstrakte Klasse modelliert ein bestimmtes Sprachkonstrukt. Eine Instanz der Klasse repräsentiert das Auftreten dieses Sprachkonstrukts an einer bestimmten Stelle im Programm. Den Klassen können Attribute zugeordnet sein, die Eigenschaften des Sprachkonstrukts modellieren. Beispielsweise besitzt die Klasse `Statechart` die Attribute `states` und `transitions`, die die Menge von Zuständen bzw. Transitionen eines Zustandsdiagramms modellieren.

Es gibt drei verschiedene Arten von Attributen, die durch die Schlüsselwörter VAL, SUB und REF unterschieden werden.

- VAL-Attribute speichern einen Wert. Als Datentyp von VAL-Attributen können sowohl vordefinierte Datentypen wie `VLString` oder `VLInt` als

auch benutzerdefinierte Datentypen verwendet werden. Durch ein Fragezeichen hinter dem Datentyp kann spezifiziert werden, dass der Wert auch undefiniert bleiben kann.

- SUB-Attribute speichern Unterstrukturen, also andere Sprachkonstrukt-Knoten, die dem speichernden Objekt logisch untergeordnet sind. Das Attribut modelliert also Baumkanten der abstrakten Struktur. Als Unterobjekte sind nur Objekte erlaubt, die einem Untertyp der angegebenen Klasse angehören. Beispielsweise kann das Attribut `states` sowohl `SimpleState` als auch `XORSuperstate`-Objekte speichern, jedoch keine `Transition`-Objekte. Hinter der Typangabe wird die Kardinalität notiert. Ein Stern bedeutet Kardinalität „0..*“, ein Fragezeichen Kardinalität „0..1“ und ein Ausrufezeichen bzw. Prozentzeichen Kardinalität „1“. Im Fall der Kardinalität „Stern“ speichert das Attribut eine Liste von Objekten, so dass den Objekten gleichzeitig eine Reihenfolge zugeordnet ist. Der Unterschied zwischen den Kardinalitäten „Ausrufezeichen“ und „Prozentzeichen“ besteht darin, dass der Editor im Fall des Ausrufezeichens auch zulässt, das das Attribut temporär leer ist.
- REF-Attribute speichern Referenzen zu anderen Objekten. Das Attribut modelliert also eine Querrelation im Baum. Es dürfen nur Objekte referenziert werden, die einem Untertyp der angegebenen Klasse angehören. Durch ein Fragezeichen hinter der Typangabe kann spezifiziert werden, dass die Referenz optional ist.

Durch das Schlüsselwort INHERITS wird die Unterklassen-Beziehung zwischen Klassen definiert. Wie aus objektorientierten Programmiersprachen bekannt, wird dadurch einerseits die Untertyp-Relation zwischen Klassen und andererseits die Vererbung von Attributen spezifiziert. Die Untertyp-Relation ermöglicht, dass an allen Stellen, an denen Objekte einer bestimmten Klasse erlaubt sind, auch Objekte von Unterklassen verwendet werden können. Durch die Vererbung besitzen Unterklassen zusätzlich zu selbst definierten Attributen auch alle Attribute ihrer Oberklassen.

In DSSL ist Mehrfachvererbung erlaubt. Eine Besonderheit ist aber, dass grundsätzlich nur von abstrakten Klassen geerbt werden kann. Hierdurch soll eine „saubere“ Modellierung gefördert werden. Würde anstatt der in Abbildung 3.2a gezeigten Modellierung die Modellierung in Abbildung 3.2b verwendet, könnten der Klasse `SimpleState` keine Eigenschaften zugeordnet werden, die nicht von `XORState` geerbt würden. Da die in der Syntax definierten Vererbungsbeziehungen auf alle darauf aufbauenden Sicht-, Analyse-

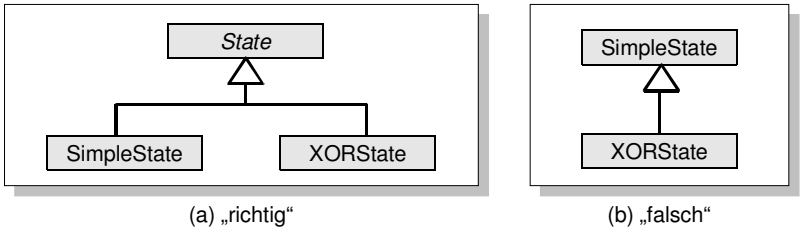


Abbildung 3.2: Modellierung von Gemeinsamkeiten durch Vererbungsbeziehungen

und Übersetzungsspezifikationen übertragen werden, ist es schwer absehbar, ob dies an bestimmten Stellen zu Problemen führt. Diese Randbedingung schränkt die Ausdrucksstärke von DSSL nicht ein, da die Vererbungsstruktur immer durch Einführen einer abstrakten Klasse in eine gültige Modellierung überführt werden kann.

Lässt man die REF-Attribute außer Acht, beschreiben DSSL-Spezifikationen eine Baumstruktur. Die SUB-Attribute definieren hierbei die Baumkanten. Wenn also zukünftig von der Baumstruktur bzw. von Kind- oder Vaterknoten die Rede ist, ist diese Interpretation der Struktur gemeint.

Jede vollständige DSSL-Spezifikation muss die Klasse „Root“ enthalten, die die Wurzel des Baums modelliert. Wenn wie in Abbildung 3.1 keine Root-Klasse enthalten ist, so ist das Beispiel als Teil einer größeren Spezifikation zu verstehen. In der Regel ist in diesem Fall die erste Klasse des Beipiels die Wurzel der betrachteten Teilspezifikation.

DSSL-Grammatiken lassen sich aufgrund der oben beschriebenen Baum-Interpretation relativ leicht auf Baum-Grammatiken abbilden. Prinzipiell entspricht eine DSSL-Klasse einer Grammatik-Produktion und die DSSL-Attribute repräsentieren die rechte Seite dieser Produktion. Im Detail gibt es aber mehrere Varianten zur Abbildung auf Grammatiken. Hierauf wird in Kapitel 4 (Abschnitt 4.1.2) genauer eingegangen.

Diskussion des Entwurfs DSSL-Grammatiken sind eng mit Baum-Grammatiken, UML-basierten Strukturmodellierungen und objektorientierten Programmiersprachen verwandt. Beim Sprachentwurf habe ich versucht, die Vorzüge dieser drei Konzepte zu vereinen. Nachfolgend möchte ich auf die Unterschiede eingehen und hieran wichtige Entwurfsentscheidungen aufzeigen.

Im Vergleich zu kontextfreien Grammatiken haben DSSL-Grammatiken folgende Besonderheiten.

- Sie enthalten mehr Strukturierungselemente wie z.B. Vererbung oder Listenbildung
- Sie ermöglichen es, die Rollen der Symbole auf der rechten Seite einer Produktion zu benennen
- Sie modellieren auch Querbeziehungen im Baum

Der erste Punkt ist wichtig für die Forderung, dass die Struktur kompatibel mit grafischen Repräsentationen und Interaktionsmechanismen sein soll. Die durch die Vererbung etablierte Untertyp-Relation entspricht der Eigenschaft konkreter Repräsentationen, dass Objekte unterschiedlichen Typs an der gleichen Stelle eingefügt werden können. Aus diesem Grund benutzen auch Ansätze, die auf kontextfreien Grammatiken basieren, erweiterte Grammatik-Kalküle. Sowohl VL-Eli (siehe Abschnitt 2.4) als auch PSG (siehe Abschnitt 2.5.1) haben Konstrukte zur Listenbildung und PSG hat ferner ein Konstrukt zur Klassenbildung.

Hintergrund des zweiten Punktes ist die Tatsache, dass Baum-Grammatiken häufig schwer interpretierbar sind, da die Rollen der Symbole auf der rechten Seite einer Produktion nicht erkennbar sind. Beispielsweise ist aus der Produktion „`IfStmt ::= Expr Stmt Stmt`“ nicht zu ersehen, welches `Stmt`-Symbol für den *true*-Zweig und welches für den *false*-Zweig einer bedingten Anweisung steht. Im Gegensatz hierzu haben die Attribute in DSSL Namen, mit denen ihre Rolle benannt werden kann. Dies hat den zusätzlichen Vorteil, dass die Aufwärtskompatibilität abstrakter Strukturen damit wesentlich verbessert wird. Wenn zu einer Grammatik-Produktion ein weiteres Symbol auf der rechten Seite hinzugefügt wird (z.B. ein weiteres `Stmt`), ist nicht klar, wie eine Instanz der alten Produktion im Kontext der neuen zu interpretieren ist. Im Gegensatz dazu lässt sich bei einer DSSL-Syntax einfach erkennen, welche Attribute hinzugekommen sind.

Schließlich werden Querbeziehungen zu anderen Sprachobjekten in kontextfreien Grammatiken nicht modelliert. VL-Eli führt zur Repräsentation von Querbeziehungen einen zusätzlichen Attributtyp ein. Allerdings kann in VL-Eli weder die Klasse der erlaubten Relationsteilnehmer noch deren Kardinalität modelliert werden, so dass die Grammatik in diesem Punkt nicht gut die Intention des Sprachentwicklers widerspiegelt. Außerdem ist es aufgrund

dieses Defizits schwierig, angemessene Strukturediteren aus der „nackten“ Syntax zu generieren, da aufgrund der mangelnden Strukturinformation kein sinnvoller Auswahlmechanismus für Querrelationen bereitgestellt werden kann.

Im Vergleich mit UML-Klassendiagrammen sind folgende Unterschiede zu verzeichnen.

- Querbeziehungen sind in DSSL binär und haben im Gegensatz zu UML immer einen Besitzer
- DSSL kommt mit wesentlich weniger Sprachkonzepten aus

Querbeziehungen werden in UML *Assoziationen* genannt. In UML sind sowohl binäre als auch n-äre Assoziationen erlaubt und alle Teilnehmer an Assoziationen sind zunächst gleichberechtigt. In DSSL sind die Daten einer Relation einem der Relationspartner zugeordnet und die Kardinalität der Relation aus Sicht des Besitzers ist auf „1“ oder „0..1“ beschränkt. Das bedeutet allerdings nicht, dass die Relationen nur in eine Richtung navigierbar sind. Zum effizienten Zugriff in der Umkehrrichtung sind in der generierten Implementierung Datenstrukturen vorgesehen, die transparent für den Benutzer die Umkehrrelation speichern. Die Zugehörigkeit von Relationen zu einem der Relationsenden lässt sich daher nicht mit den in UML spezifizierbaren Navigationsrichtungen vergleichen.

Die Asymmetrie von DSSL-Relationen ist ein wichtiger Entwurfsaspekt, denn sie passt sehr gut zu wichtigen interaktiven Eigenschaften von Strukturediteren. Ein Beispiel für den Nutzen dieser Asymmetrie ist die Beziehung zwischen Funktionsdefinition und Funktionsaufruf. Hier gehört das Attribut zum Funktionsaufruf, was sich z.B. durch unterschiedliches Verhalten beim Duplizieren von Objekten äußert. Wird ein Funktionsaufruf dupliziert, ruft er die selbe Funktion auf wie das Original. Wird eine Funktionsdefinition dupliziert, steht sie zunächst mit keinem Aufruf in Beziehung. Genauer wird hierauf in Abschnitt 3.2.3 eingegangen.

Bezüglich des zweiten Spiegelpunktes ist zu vermerken, dass DSSL im Vergleich zu UML auf Sichtbarkeiten, Assoziationsklassen, Zugriffsrichtungen, n-ären Assoziationen, Stereotypen usw. verzichtet. Durch diese Reduktion der Modellierungsfreiheiten lassen sich die Strukturen wesentlich einfacher und einheitlicher handhaben und die Flut alternativer Entwurfsvarianten wird eingeschränkt.

In Bezug auf objektorientierte Programmiersprachen gibt es folgende Unterschiede.

- DSSL enthält Konstrukte für Kardinalitäten und Listenbildung
- In DSSL-Strukturen sind Relationen in beiden Richtungen navigierbar
- DSSL erlaubt Mehrfachvererbung, aber verbietet die Spezialisierung konkreter Klassen
- In DSSL wird zwischen Baum- und Querrelationen unterschieden

Zur Begründung des ersten Punkts gelten die bereits oben vorgebrachten Argumente.

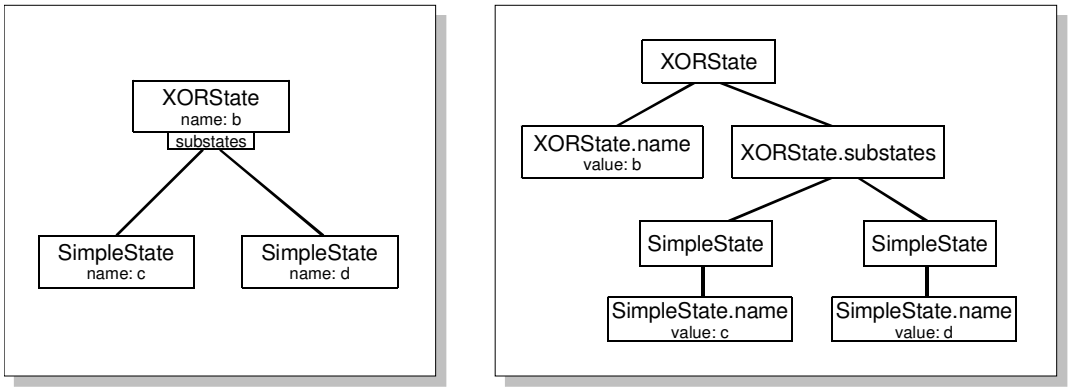
Der zweite Punkt drückt aus, dass eine Relation durchaus auch in umgekehrter Richtung durchwandert werden kann. Eine Funktionsdefinition kann demnach z.B. auf ihre Aufrufstellen zugreifen. Dies ist wichtig, um flexibel mit DSSL-Strukturen umgehen zu können. Wie bereits oben erwähnt sind in der generierten Implementierung Datenstrukturen vorgesehen, die einen effizienten Zugriff in der Umkehrrichtung gestatten.

Der dritte Punkt ist die Konsequenz der Tatsache, dass DSSL keine Implementierungs-, sondern eine Modellierungssprache ist. Mehrfachvererbung ist hier im Gegensatz zu Programmiersprachen unproblematisch. Der Grund, warum konkrete Klassen nicht spezialisiert werden dürfen, wurde bereits oben genannt.

Der vierte Punkt – die Unterscheidung zwischen Baum- und Querrelationen – ist natürlich entscheidend, um die Syntax auf Baum-Grammatiken abbilden zu können.

3.1.3 Zugriffsfunktionen und Pfadausdrücke

Der Umgang mit der abstrakten Struktur ist eine Kernaufgabe jedes Strukturediteors. Daher ist es wichtig, dass man mit DSSL-Strukturen sowohl konzeptionell als auch praktisch einfach umgehen kann. Im Folgenden werden einige Notationen zum Durchwandern von Strukturen beschrieben, die auch in den nachfolgenden Abschnitten dieses Kapitels benötigt werden. Besonders zu nennen sind hier die so genannten Pfadausdrücke. Pfadausdücke dienen sowohl der konzeptuellen Beschreibung bestimmter Mechanismen wie der



(a) Logische Struktur (b) Implementierte Datenstruktur

Abbildung 3.3: Implementierung von DSSL-Strukturen

strukturellen Kopplung (vgl. Abschnitt 3.3) als auch zur Definition struktureller Konsistenzbedingungen (vgl. Abschnitt 3.1.4).

Als Grundlage zur Definition von Pfadausdrücken muss zunächst genauer auf die Repräsentation von DSSL-Instanzen eingegangen werden. In Abbildung 3.3 ist eine logische DSSL-Instanz und die zugehörige implementierte Datenstruktur dargestellt. Man erkennt, dass es in der implementierten Datenstruktur sowohl für Klassen-Instanzen als auch für Attribut-Instanzen eigenständige Objekte gibt. Diese Trennung ist sowohl für die Definitionen von Pfadausdrücken als auch für die praktische Implementierung von entscheidender Bedeutung. Nachfolgend nenne ich die Instanzen von Klassen *Sprachkonstrukt-Knoten* und die Instanzen von Attributen *Attribut-Knoten*. Attribut-Knoten lassen sich je nach Art weiter in *SUB-Knoten*, *VAL-Knoten* und *REF-Knoten* unterteilen. Beachtenswert ist, dass die implementierte Datenstruktur ein bipartiter Graph ist, denn die Kinder von Sprachkonstrukt-Knoten sind Attribut-Knoten und die Kinder von SUB-Knoten bzw. die Werte von REF-Knoten sind Sprachkonstrukt-Knoten.

Im Rest dieses Abschnitts werden DSSL-Strukturen wie in Abbildung 3.3b dargestellt, um das Verständnis der nachfolgenden Definition zu erleichtern. Im weiteren Verlauf der Arbeit wird allerdings der Übersichtlichkeit halber die Notation in Abbildung 3.3a verwendet.

Die folgenden Zugriffsfunktionen bilden die Grundlage für Pfadausdrücke:

- Sei R ein Knoten. Dann bezeichne $R.parent$ den Vaterknoten in der implementierten Datenstruktur. Man beachte, dass der Vater eines

Sprachkonstrukt-Knotens ein Attribut-Knoten und der Vater eines Attribut-Knotens ein Sprachkonstrukt-Knoten ist.

- Sei R ein Blatt-Knoten, d.h. entweder ein VAL-Knoten oder ein REF-Knoten. Dann bezeichne $R.\mathit{value}$ den von R gespeicherten Wert. Falls R ein REF-Knoten ist, ist $R.\mathit{value}$ also ein Sprachkonstrukt-Knoten.
- Sei R ein Sprachkonstrukt-Knoten und a der Name eines Attributs von R. Dann bezeichne $R.a$ den Attribut-Knoten von R mit Namen a.
- Sei R ein Sprachkonstrukt-Knoten, der einem SUB-Attribut S untergeordnet ist. Dann bezeichne $R.\mathit{prev}$ und $R.\mathit{next}$ den Unterknoten von S, der in der Listenordnung vor bzw. nach R kommt.
- Sei R ein Knoten und T der Typ eines Knotens der implementierten Datenstruktur. Dann bezeichne $R.\mathit{including}_T$ den nächstgelegenen R übergeordneten Knoten des Typs T. (Die Knoten in Abbildung 3.3 haben z.B. die Typen `XORState` oder `SimpleState.name`.)
- Sei R ein Knoten. Dann bezeichne $R.\mathit{children}$ die Menge der direkten Unterknoten in der implementierten Datenstruktur. Ist R ein Sprachkonstrukt-Knoten, so besteht die Menge $R.\mathit{children}$ also aus Attribut-Knoten. Ist R ein SUB-Knoten, so besteht die Menge aus Sprachkonstrukt-Knoten. Ist R ein VAL- oder REF-Knoten, so ist die Menge leer.
- Sei R ein Sprachkonstrukt-Knoten. Dann bezeichne $R.\mathit{ivalue}$ die Menge aller REF-Knoten S, für die gilt $S.\mathit{value} = R$. Die Relation „ivalue“ („*inverse value*“) bildet somit die Umkehrrelation der Relation „value“.
- Sei R eine Menge von Knoten und T der Typ eines Knotens in der implementierten Datenstruktur. Dann bezeichne $R\ [T]$ die Teilmenge der Knoten aus R, die den Typ T besitzen.

Mit Hilfe der oben definierten Zugriffsfunktionen kann die Datenstruktur bereits durchwandert werden. Ist beispielsweise S einer der `SimpleState`-Knoten aus Abbildung 3.3, so ist $S.\mathit{parent}.\mathit{parent}.\mathit{name}.\mathit{value}$ der Name des übergeordneten `XORState`-Knotens, also „b“.

Die Zugriffsfunktionen werden nachfolgend *Selektionen* genannt. Die meisten Selektionen bilden Knoten auf Knoten ab. Die letzten drei Selektionen sind jedoch Besonderheiten, da hier die Funktionsergebnisse *Mengen* von Knoten

sind. Um beliebige Selektionen hintereinander ausführen zu können, müssen demnach alle Selektionen auch auf Mengen anwendbar sein. Dazu werden die Selektionen kanonisch erweitert. Sei also M eine Knotenmenge und S eine Selektion, dann gelte

- $MS := \{mS | m \in M\}$, falls S Knoten auf einzelnen Knoten abbildet.
- $MS := \bigcup_{m \in M} mS$, falls S Knoten auf eine Knotenmenge abbildet.

Ein Pfadausdruck ist eine Selektion, die durch Hintereinanderausführung der Elementarselektionen gebildet wird. Diese Definition von Pfadausdrücken hat im Gegensatz zu andersartigen Definitionen den Vorteil, dass die Struktur eines Pfadausdrucks bestimmt, ob der Ergebnistyp ein Einzelwert oder eine Menge ist. Insbesondere macht dies die Formeln in Abschnitt 3.3 einfacher, wo Pfadausdrücke verwendet werden, um Konsistenzbedingungen zwischen gekoppelten Strukturen zu definieren. Zur Implementierung von Pfadausdrücken ist es allerdings zweckmäßig, die beiden Fälle explizit zu unterscheiden. Daher beinhaltet DEViL zwei Funktionen zum Auswerten von Pfadausdrücken. Die eine Funktion liefert eine Menge und erlaubt alle Selektionen, wohingegen die andere Funktion einen einzelnen Wert liefert, aber die verwendbaren Selektionen einschränkt.

Bis jetzt wurde nur eine Verknüpfung zwischen Selektionen betrachtet, nämlich die Hintereinanderausführung. Pfadausdrücke dürfen aber auch andere Operatoren wie „*“ (0..n-fache Wiederholung), „+“ (1..n-fache Wiederholung) und „|“ (Alternative) enthalten. Die Semantik dieser Operatoren entspricht denen in regulären Ausdrücken. Sind S und R Selektionen und M eine Knotenmenge, so lässt sich die Semantik wie folgt definieren:

- $MS* := M \cup MS \cup MSS \cup MSSS \cup \ldots$
- $MS+ := (MS)S*$
- $M(S|R) := MS \cup MR$

Für die nachfolgenden Betrachtungen werden schließlich noch folgende Definitionen benötigt:

- Seien R und S Knoten. Dann bezeichne $TGV(R, S)$ den tiefsten gemeinsamen Vorgänger von R und S im Baum.

- Seien R und S Sprachkonstrukt-Knoten. Dann bedeute $R \triangleleft S$, dass R und S Kinder des gleichen SUB-Knotens sind und R in der Listenordnung von S kommt, d.h. dass gilt $R \in S.prev+$

Anwendungsbeispiel Trotz der etwas kompliziert wirkenden Definition sind Pfadausdrücke einfach zu verstehen und anzuwenden. Zur Demonstration soll nachfolgend ein Pfadausdruck zur Berechnung aller direkter und indirekter Oberklassen einer gegebenen Klasse in Klassendiagrammen vorgestellt werden. Um das Beispiel realistischer und anspruchsvoller zu machen, sei die Vererbungsrelation nicht als Attribut von `Class`-Knoten, sondern als eigenständige Klasse modelliert (siehe Abbildung 3.4a).

Zunächst betrachten wir einen Pfadausdruck, der die Menge aller *direkten* Oberklassen eines `Class`-Knotens S berechnet (siehe Abbildung 3.4b). Zur Illustration enthält die Abbildung auch eine Beispielstruktur, in der ein entsprechender Pfad eingetragen ist. Ausgehend vom `Class`-Knoten S liefert die Selektion *.ivalue* alle REF-Knoten, die auf S verweisen. Die Selektion *[Inheritance.subclass]* schränkt die gefundene Knotenmenge auf Attributknoten namens `subclass` im Kontext eines `Inheritance`-Knotens ein. Die Selektion *.parent* liefert die Menge der entsprechenden `Inheritance`-Knoten und *.superclass* liefert die Menge der REF-Attribute namens `superclass`. Die Selektion *.value* liefert schließlich die Menge der Werte dieser Attribute, also die Menge der direkten Oberklassen von S.

Abbildung 3.4c zeigt, wie sich der in Abbildung 3.4b vorgestellte Pfadausdruck auf indirekten Oberklassen erweitern lässt. Dieser muss dazu lediglich um den Wiederholungsoperator „+“ ergänzt werden.

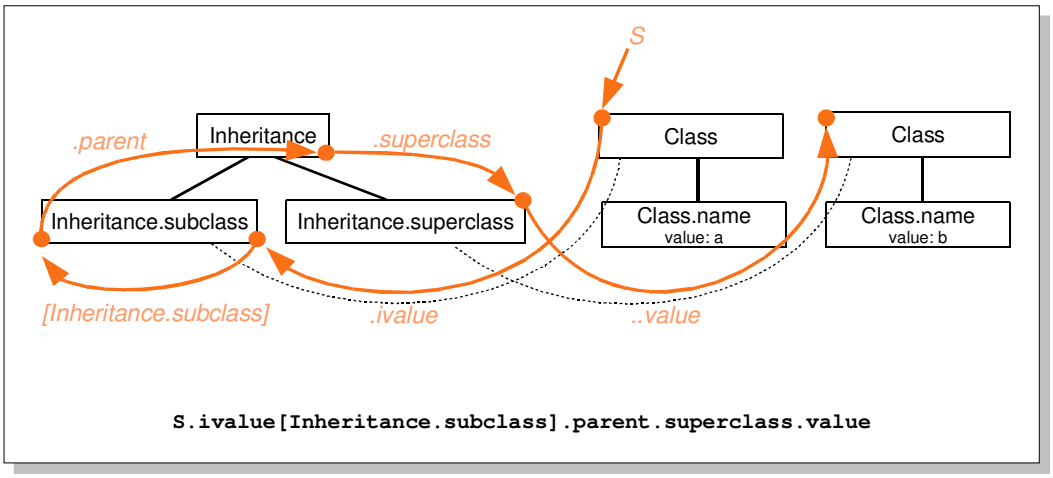
3.1.4 Spezifikation von Konsistenzbedingungen

Im Allgemeinen beschreibt die DSSL-Syntax nur eine Obermenge der korrekten Programme. Normalerweise werden zusätzliche Konsistenzbedingungen benötigt, die sich aus der statischen Semantik des Programms ergeben. DEViL stellt verschiedene Mechanismen zur Verfügung, solche Konsistenzbedingungen zu formulieren. Die Mechanismen unterscheiden sich darin, wann bzw. wie oft die Konsistenz geprüft wird und welche Auswirkungen die Bedingungen auf die grafische Benutzungsschnittstelle haben.

Prinzipiell lassen sich drei Varianten unterscheiden:

```
CLASS Class {
  name: VAL VLString;
}

CLASS Inheritance {
  subclass: REF Class;
  superclass: REF Class;
}
```

(a) Syntax



(b) Pfadausdruck zur Bestimmung direkter Oberklassen

```
S(.ivalue[Inheritance.subclass].parent.superclass.value)+
```

(c) Pfadausdruck zur Bestimmung direkter und indirekter Oberklassen

Abbildung 3.4: Berechnung von Oberklassen durch Pfadausdrücke

- Vor einer Editieroperation wird geprüft, ob das Ergebnis im Sinne der Konsistenzbedingung gültig ist. Falls das Ergebnis ungültig wäre, darf der Benutzer die Operation nicht durchführen.
- Die Konsistenzbedingung wird nach jeder Editieroperation geprüft und auftretende Verstöße werden ggf. grafisch gekennzeichnet.
- Die Konsistenzbedingung wird erst geprüft, wenn der Benutzer die Analyse, Simulation bzw. Übersetzung des Programms startet. Evtl. auftretende Verstöße werden dann ähnlich wie im zweiten Fall gemeldet.

All diese Varianten haben Vor- und Nachteile. Die erste Variante garantiert die Korrektheit der konstruierten Programme, schränkt aber den Sprachbenutzer in seiner Handlungsfreiheit ein. Daher ist diese Variante nur für lokale Eigenschaften sinnvoll. Die zweite Variante schränkt den Benutzer nicht ein, erlaubt aber auch inkonsistente Programme.

Die ersten beiden Methoden sind vor allem deswegen problematisch, weil nach jeder Änderung alle Prüfungen erneut durchgeführt werden müssen, was aufwändig sein kann und so die Reaktionszeit des Systems verlängert. Im Gegensatz dazu ist die dritte Variante auch für aufwändige Prüfungen geeignet. Der Nachteil ist allerdings, dass der Benutzer erst spät erfährt, ob das Programm statisch korrekt ist.

Da alle drei Alternativen ihre Daseinsberechtigung haben, werden alle von DEViL unterstützt. Welche Alternative verwendet wird, ist eine Entwurfsentscheidung des Sprachentwicklers. Der Spezifikationskontext einer Konsistenzbedingung entscheidet über die Art der Prüfung.

Wird eine Prüfung im Kontext eines VAL- und REF-Attributs durchgeführt, werden Zuweisungen an das entsprechende Attribut nur dann durchgeführt, wenn sie die Konsistenzbedingungen nicht verletzen. Wird z.B. in einem Eingabedialog eine ungültige Zeichenkette eingegeben, wird eine Fehlermeldung ausgegeben und das Schließen des Dialogs verhindert. In Auswahllisten für Referenzen werden grundsätzlich nur Werte angezeigt, die bzgl. dieser Art von Konsistenzbedingungen erlaubt sind. Auch beim *Drag-and-Drop* in visuellen Sichten lassen sich nur Zielpunkte selektieren, die entsprechend dieser Prüfung gültig sind.

Eine Konsistenzbedingung der zweiten Art wird im Kontext eines Sprachkonstrukt-Knotens spezifiziert. Fehlermeldungen dieser Art werden in einer speziellen Sicht, der so genannten Fehler-Sicht angezeigt. Diese wird

nach jeder Änderung automatisch aktualisiert. Klickt man mit der Maus auf eine darin enthaltene Fehlermeldung, so wird das Sprachkonstrukt, auf das sich die Meldung bezieht, in einer dafür geeigneten Sicht hervorgehoben. Zudem lassen sich Sichttyp-Spezifikationen auch so erweitern, dass Fehler durch spezifische Markierungen kenntlich gemacht werden.

Die letzte Art von Konsistenzbedingungen wird im Kontext von Verarbeitungsprozessoren definiert. Verarbeitungsprozessoren dienen zur Analyse oder Übersetzung visueller Programme und werden in DEViL durch attributierte Grammatiken spezifiziert. Zur Implementierung von Verarbeitungsprozessoren stehen die gleichen Hilfsmittel zur Verfügung, die auch in Eli [31] zur Analyse und Übersetzung textueller Programme verwendet werden können, z.B. Module zur Typanalyse oder Operator-Identifikation. Um statische Fehler zu melden, lassen sich während der Attributberechnung Meldungsfunktionen aufrufen, denen als Parameter ein Kontext und eine Klartext-Fehlermeldung übergeben wird. Die so generierten Meldungen werden gesammelt und ggf. in einem Meldungsfenster angezeigt. Der Vorteil dieser Vorgehensweise ist, dass Fehler häufig sowieso als „Abfallprodukt" der Übersetzung erkannt werden. Da bei der Übersetzung typisierter Ausdrücke z.B. generell eine Operatoridentifikation durchgeführt werden muss, wird ganz nebenbei der Fall erkannt, dass die Operanden inkompatible Typen besitzen.

Spezifikationsvarianten für Konsistenzbedingungen Konsistenzbedingungen der ersten und zweiten Art werden durch Funktionen definiert, die Attributen oder Klassen zugeordnet werden. Die Funktionen müssen im Fehlerfall eine Klartext-Fehlermeldung liefern, die ggf. dem Benutzer angezeigt wird. Die Spezifikation der Konsistenzbedingungen durch separate Funktionen hat den Vorteil, dass die Prüfungen individuell und bedarfsgerecht durchgeführt werden können.

Zur Spezifikation der Prüfungen gibt es zwei Notationen, die in Abbildung 3.5 gegenübergestellt sind. Abbildung 3.5a zeigt eine direkt implementierte Konsistenzprüfung, die dem VAL-Attribut `Person.numberOfChildren` zugeordnet wurde. Im Beispiel wird eine Fehlermeldung zurückgeliefert, wenn der Attributwert kleiner als Null ist.

Die Spezifikation in Abbildung 3.5b hat die gleiche Funktionalität. Dort wird die Prüfung durch eine CHECK-Klausel im Rahmen der Klassendefinition spezifiziert. Die Parameter der dort aufgerufenen Funktion `greaterOrEqual` werden schrittweise gebunden. Der erste Parameter – in

```
addCheck Person.numberOfChildren {
  if { $value < 0 } {
    return "Value must be positive"
  } else {
    return ""
  }
}
```

(a)

```
CLASS Person {
  numberOfChildren: VAL VLInt
    CHECK "greaterOrQual 0";
}
```

(b)

Abbildung 3.5: Zwei Varianten zur Spezifikation von Konsistenzbedingungen

```
addCheck Class {
  set superclassSet [c::getList {
    $obj(.IVALUE[Inheritance.subclass].parent.superclass.value)+ }]
  if { [elementOf $obj $supereclassSet] } {
    return "This class is part of a cyclic inheritance relation"
  } else {
    return ""
  }
}
```

Abbildung 3.6: Spezifikation zur Erkennung zyklischer Vererbungsbeziehungen

diesem Fall „0“ – steht bereits in der CHECK-Klausel. Der zweite Parameter ergibt sich in diesem Fall aus dem aktuellen Attributwert. In DEViL sind bereits eine Reihe solcher Funktionen vordefiniert, so dass einfache Konsistenzbedingungen direkt bei der Klassendefinition formuliert werden können.

Zur Implementierung komplexerer, struktureller Konsistenzbedingungen können die in Abschnitt 3.1.3 eingeführten Pfadausdrücke verwendet werden. Um z.B. zu überprüfen, ob in einem Klassendiagramm zyklische Vererbungsketten existieren, lässt sich der in Abbildung 3.4c gezeigte Pfadausdruck zur Berechnung aller Oberklassen einer Klasse S verwenden. S ist genau dann an einer zyklischen Vererbungskette beteiligt, wenn S eine (indirekte) Oberklasse von sich selbst ist. Die Spezifikation der entsprechenden Konsistenzprüfung ist in Abbildung 3.6 dargestellt. Wichtig ist, dass DEViL die Ergebnismenge jedes Pfadausdrucks auch dann vollständig berechnen kann, wenn die Datenstruktur Zyklen aufweist.

```
createObject(type)
deleteObject(objectNode)

insertObject(contextNode, newObjectNode)
extractObject(objectNode)

setValue(valNode, newValue)
setReference(refNode, newValue)
```

Abbildung 3.7: DSSL-Änderungsoperationen

3.1.5 Änderungsfunktionen

Sowohl zur Implementierung konkreter Editieroperationen als auch zur Implementierung der strukturellen Kopplung müssen Strukturänderungen durchgeführt werden. Als elementare Schnittstelle für solche Änderungen dienen die in Abbildung 3.7 gezeigten Funktionen, die nachfolgend kurz erläutert werden sollen.

Die Funktion `createObject` erzeugt einen neuen Sprachkonstrukt-Knoten der angegebenen Klasse und die Funktion `deleteObject` löscht Sprachkonstrukt-Knoten.

Die Funktion `insertObject` fügt den Knoten `newObjectNode` in einen neuen Kontext ein. Der Parameter `contextNode` bestimmt hierbei die genaue Einfügestelle. `newObjectNode` wird zwischen die Knoten *contextNode.prev* und *contextNode* in den SUB-Knoten *contextNode.parent* eingefügt. Um Knoten auch am Listenende einfügen zu können, besitzt jeder SUB-Knoten einen Spezialknoten, der das Ende der Liste repräsentiert.

Die Funktion `extractObject` entfernt das übergebene Objekt aus dem SUB-Knoten *objectNode.parent*. Nach dieser Operation gilt $objectNode\,.\,parent = null$.

Die Funktionen `setValue` und `setReference` setzen schließlich den neuen Wert eines VAL- bzw. REF-Attributs.

Man beachte, dass durch die angegebenen Funktionen die Änderungsschnittstelle für DSSL-Strukturen vollständig beschrieben ist. Die Einfachheit und Redundanzfreiheit dieser Schnittstelle vereinfacht sowohl die Implementierung als auch die Anwendung der Änderungsfunktionen. In den Abschnitten 3.2 und 4.1.4 wird gezeigt, wie sich konkrete Editierfunktionen auf diese Operationen abbilden lassen. In Abschnitt 3.3 wird diese Schnittstelle benutzt, um die Kopplung von editierbarer und semantischer Struktur zu spezifizieren.

3.2 Konsequenzen für die Benutzungsschnittstelle

Das im letzten Abschnitt beschriebene Modellierungskonzept für abstrakte Strukturen hat Auswirkungen auf die Benutzungsschnittstelle des Strukturediteors, denn DSSL wird auch zur Definition der Editor-Syntax verwendet, die wiederum eng mit den visuellen Sichten in Beziehung steht. Nachfolgend soll der Zusammenhang zwischen editierbarer Struktur und konkreter Darstellung sowie den dort verwendeten Interaktionsmechanismen genauer herausgestellt werden.

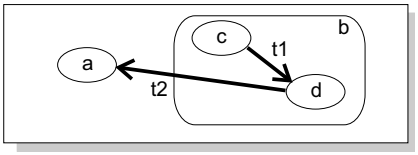
3.2.1 Zusammenhang von Struktur und Repräsentation

Wie bereits oben erwähnt ist die editierbare Struktur ein Datenmodell für die konkrete Repräsentation. Das bedeutet, dass jedes elementare Objekt der visuellen Darstellung durch ein Objekt der editierbaren Struktur repräsentiert wird und dass die umzusetzenden Editiermechanismen zur Editor-Syntax passen müssen. Wenn man also eine visuelle Sprache sowie die zur Bearbeitung anwendbaren Editieroperationen kennt, lässt sich hieraus relativ eindeutig eine Editor-Syntax ableiten.

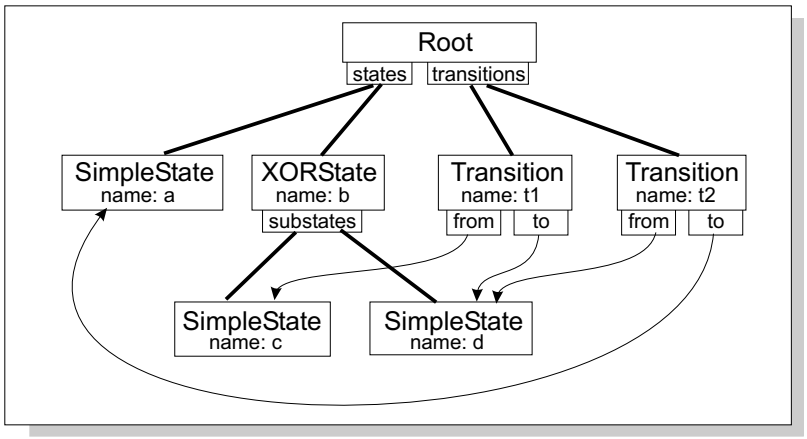
Ein Beispiel für den Zusammenhang zwischen konkreter Repräsentation und editierbarer Struktur ist in Abbildung 3.8 dargestellt. Man erkennt, dass alle grafischen Grundobjekte der Sprache durch Knoten der editierbaren Struktur repräsentiert sind. Da eine Transition und ihre Beschriftung untrennbar miteinander verbunden sind, sind sie als *ein* grafisches Grundobjekt zu verstehen und werden dementsprechend durch einen einzigen Knoten repräsentiert. Da DSSL-Instanzen Bäume sind, wird zur Vervollständigung der Struktur ein Wurzelknoten benötigt, der das Diagramm als Ganzes repräsentiert. Dieser Wurzelknoten besitzt als Unterelemente die äußeren Zustände und die Transitionen.

3.2.2 Editieroperationen

Wie bereits oben erwähnt war es ein wichtiges Entwurfsziel von DSSL, dass bereits aus der „nackten“ Syntax ein angemessener Editor generiert werden kann. Weiterhin ist wichtig, dass sich die in visuellen Sichten verwendeten Editiermechanismen möglichst direkt auf die editierbare Struktur abbilden lassen.



(a) Konkrete Repräsentation



(b) Zugehörige Instanz des DSSL-Strukturmodells

Abbildung 3.8: Die Struktur eines UML-Zustandsdiagramms

Betrachtet man die Änderungsschnittstelle der abstrakten Struktur in Abbildung 3.7 auf Seite 86 wird klar, dass bereits die folgenden Grundoperationen zum Editieren der Struktur ausreichen:

- Erzeugen eines neuen Sprachkonstrukt-Knotens als Unterelement eines SUB-Knotens
- Erzeugen eines neuen Sprachkonstrukt-Knotens vor oder nach einem anderen Sprachkonstrukt-Knoten
- Setzen des neuen Wertes eines REF-Knotens
- Setzen des neuen Wertes eines VAL-Knotens
- Löschen eines Sprachkonstrukt-Knotens

Als Benutzungsschnittstelle dieser Editieroperationen dient eine Baum-Sicht, wie sie in Abbildung 3.9 dargestellt ist. Das Beispiel basiert auf der Syntax in Abbildung 4.4a auf Seite 136 und zeigt die Struktur des Nassi-Shneiderman Diagramms aus Abbildung 4.5. Die Knoten des Baums entsprechen denen der implementierten Datenstruktur, wie sie in Abbildung 3.3b auf Seite 78

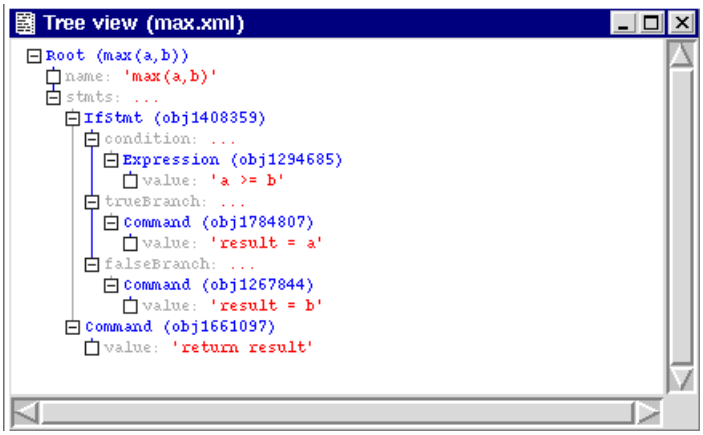


Abbildung 3.9: Baum-Sicht zum Editieren der abstrakten Struktur

illustriert ist. Die oben genannten Editieroperationen sind aus dem Kontextmenü der entsprechenden Baumknoten aufrufbar. Die erlaubten Klassen zur Erzeugung eines neuen Sprachkonstrukt-Knotens lassen sich aus dem Typ des entsprechenden SUB-Knotens ableiten. Im Kontext von REF-Attributen kann durch eine Auswahlliste der Wert der Referenz gesetzt werden. Auch hier kann basierend auf dem Typ des REF-Attributs eine sinnvolle Vorauswahl getroffen werden. Im Kontext eines VAL-Knotens kann ein Eingabedialog geöffnet werden, der dem Typ des VAL-Attributs entspricht.

Der Editor erlaubt keine Editieroperationen, die die Kardinalitäten von SUB-Attributen verletzen. Wenn ein Knoten mit einem SUB%-Attribut eingefügt wird, wird automatisch ein entsprechender Unterknoten für dieses Attribut erzeugt. Damit die Klasse des zu erzeugenden Knotens eindeutig ist, muss der Typ des SUB%-Attributs eine nicht-abstrakte Klasse sein. Schließlich benutzt die Baumsicht einen speziellen Automatismus, um sie übersichtlicher zu machen: Wenn ein Sprachkonstrukt ein `name`-Attribut besitzt, so wird dieser Name als Beschriftung des entsprechenden Knotens verwendet.

Die meisten der in visuellen Sichten verwendeten Editieroperationen sind lediglich andere Ausdrucksformen der oben beschriebenen abstrakten Operationen. Wird z.B. ein Objekt durch *direct manipulation* an eine andere Stelle verschoben, so entspricht dies dem Entfernen eines Sprachkonstrukt-Knotens und dem Einfügen in einen anderen Kontext. In visuellen Sichten wird beim *Drag-and-Drop* grundsätzlich die nächstgelegene gültige Einfügestelle hervorgehoben, in die ein Sprachobjekt laut Syntax eingefügt werden kann. Die hierzu benötigten Informationen ergeben sich auf die gleiche Weise, wie sie auch zur Realisierung der Baumsicht abgeleitet werden. Ein anderes Beispiel ist

das „Umhängen“ eines Linienendpunktes durch *Drag-and-Drop*. Wenn ein Linienendpunkt bewegt wird, werden wiederum die gültigen Linienendpunkte grafisch hervorgehoben. Auch die hierzu verwendeten Informationen ergeben sich primär aus der Syntax, denn im abstrakten Sinne entspricht das „Umhängen“ der Linie lediglich der Wertänderung eines REF-Knotens.

Die abstrakten Änderungsoperationen aus Abbildung 3.7 passen sehr gut zu den genannten konkreten Interaktionsmechanismen, weil sie mit sehr wenigen Kontextinformationen auskommen. Hierauf wird genauer in Abschnitt 4.1.4 eingegangen.

Dass die abstrakten Änderungsoperationen mit so wenig Kontextinformation auskommen ist nicht selbstverständlich. In Systemen, die auf Graphgrammatiken basieren, werden für Editieroperationen häufig viel mehr Kontextinformationen benötigt. In solchen Systemen werden Strukturen geändert, indem Kontexte ausgewählt und darauf geeignete Graphgrammatikproduktionen angewendet werden. Beispielsweise müssen zum Einfügen eines neuen Objektes mindestens zwei weitere Kontexte ausgewählt werden, nämlich der Vorgänger und der Nachfolger des neuen Objektes. Zum Löschen von Teilstrukturen werden Grammatikproduktionen in umgekehrter Richtung angewendet, wobei wiederum zunächst die entsprechenden Kontexte ausgewählt werden müssen. Natürlich lassen sich auch in solchen Systemen andersartige Editiermechanismen realisieren, aber dies ist häufig mit erheblichem Zusatzaufwand bei der Spezifikation verbunden.

3.2.3 Cut-and-Paste

Von Strukturediteren wird im Allgemeinen erwartet, dass Knoten kopiert oder ausgeschnitten werden können, um dann an einer neuen Stelle eingefügt werden zu können. Dieser Mechanismus wird häufig als *Cut-and-Paste* bezeichnet. Aus historischen Gründen ist *Cut-and-Paste* als Oberbegriff von Kopier- und Verschiebeoperationen zu verstehen. Die Kopieroperation wird aus den gleichen Gründen manchmal auch *non-destructive cut* genannt.

Dank der baumbasierten Modellierung lässt sich der *Cut-and-Paste* Mechanismus zu großen Teilen automatisch und ohne zusätzlichen Spezifikationsaufwand realisieren. Da beim Verschieben von Sprachelementen normalerweise all ihre Relationen erhalten bleiben sollen, ist die Verschiebe-Operation sehr einfach zu realisieren. Es ist lediglich eine `extractObject` und eine `insertObject` Operation notwendig.

Beim Kopieren von Sprachkonstrukt-Knoten gibt es mehr zu bedenken. Hier ist zu entscheiden, wie mit Querrelationen umzugehen ist. Kopiert man beispielsweise einen Teilbaum, der Funktionsdefinitionen enthält, erwartet man, dass die Kopie zunächst mit keinen Aufrufstellen in Beziehung steht. Kopiert man dagegen einen Teilbaum mit Funktionsaufrufen, erwartet man, dass die Kopien die selben Funktionen referenzieren wie die Originale. Wird schließlich ein Teilbaum mit Funktionsaufrufen und entsprechenden Funktionsdefinitionen kopiert, erwartet man, dass sich die Kopien der Funktionsaufrufe auf die Kopien der Funktionsdefinitionen beziehen.

Um diese Eigenschaft zu erreichen, wird in DEViL der folgende Algorithmus zum Kopieren eines Teilbaums verwendet:

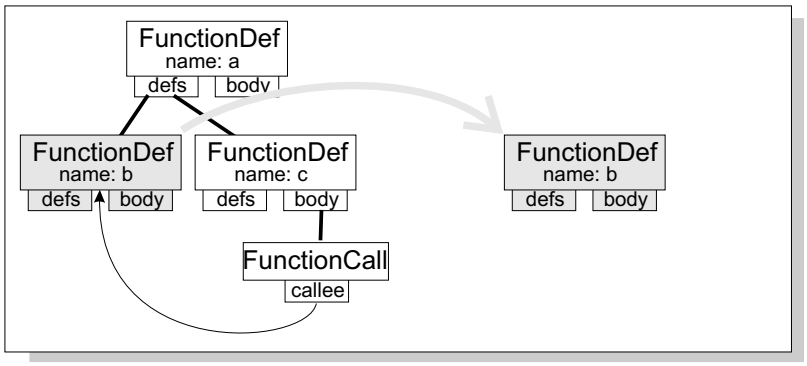
1. Kopiere die Baumstruktur rekursiv, wobei die Werte von REF- und VAL-Attributen übernommen werden.
2. Ersetze die Werte aller REF-Attribute, die sich auf Knoten des kopierten Teilbaums beziehen durch eine Referenz auf ihre Kopie.

Die Konsequenzen des Algorithmus sind in Abbildung 3.10 am Beispiel von Funktionsaufrufen illustriert. In allen drei Fällen entspricht das Ergebnis dem gewünschten Verhalten.

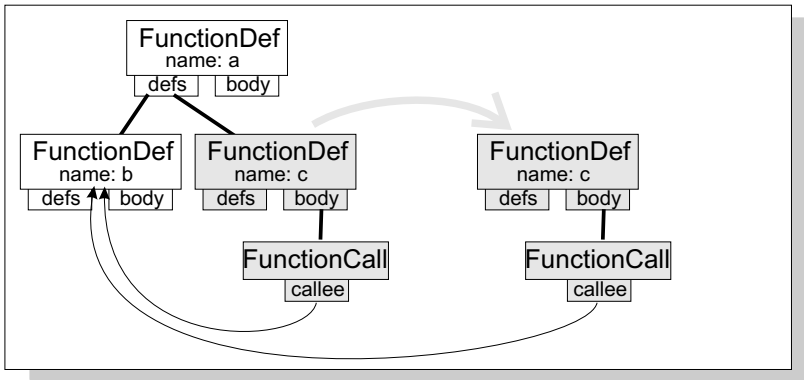
Cut-and-Paste in Zustandsdiagrammen In vielen Fällen entspricht der vorgestellte Algorithmus den Erwartungen des Benutzers, so dass keine zusätzlichen Spezifikationen notwendig sind. Allerdings kann es in Einzelfällen zu unerwünschten Ergebnissen kommen, z.B. wenn das Ergebnis semantischen Randbedingungen widerspricht. In diesen Fällen können zusätzliche Spezifikationen helfen, um die Benutzerfreundlichkeit zu verbessern.

Ein Beispiel hierfür liefert die in Abbildung 3.8 gezeigte Struktur eines Zustandsdiagramms. Selektiert und kopiert man den `XORState` „b“, so werden zwar die ihm untergeordneten Zustände c und d mit kopiert, nicht aber die Transition t1. Dies liegt daran, dass t1 zwar visuell, nicht aber strukturell zu „b“ gehört.

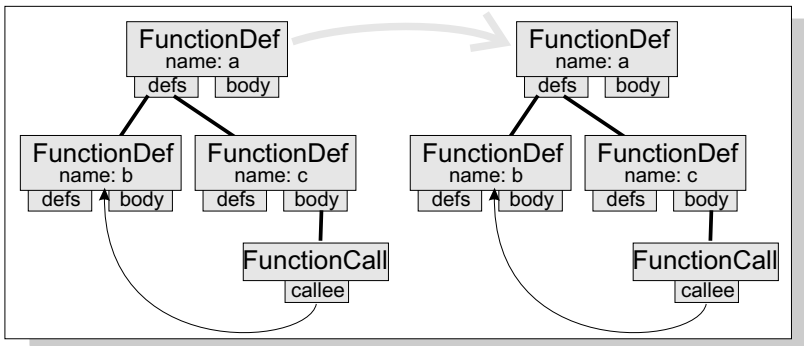
Um das Kopierverhalten zu verbessern ist es nahe liegend, die editierbare Struktur so zu ändern, dass sie besser zu den Eigenschaften der visuellen Darstellung passt. Eine Transition von A nach B sollte dementsprechend



(a) Kopieren einer Funktionsdefinition



(b) Kopieren eines Funktionsaufrufs



(c) Kopieren eines zusammengehörenden Definition-Aufruf-Paares

Abbildung 3.10: Behandlung von Querrelationen beim Kopieren von Teilstrukturen

nicht dem Zustandsdiagramm an sich, sondern dem sie umschließenden Zustand, d.h. dem Knoten TGV(A,B) zugeordnet sein. Dazu müsste der Klasse `XORState` ein weiteres Attribut hinzugefügt werden, das eine Liste von `Transition`-Elementen speichert. Da dies für den Benutzer transparent bleiben soll, muss nach jeder Änderungsoperation dafür gesorgt werden, dass weiterhin alle Transitionen dem richtigen Knoten zugeordnet sind. Die dafür notwendige Berechnung und Umstrukturierung lässt sich in DEViL in der Phase realisieren, in der auch die Kopplung zwischen semantischer und editierbarer Struktur erfolgt (s.u.). Wird die Struktur entsprechend angepasst, führen *Cut-and-Paste* Operationen zum gewünschten Ergebnissen.

3.3 Kopplung von semantischer und editierbarer Struktur

Bereits in Kapitel 2 habe ich erwähnt, dass der Zusammenhang zwischen semantischer und editierbarer Struktur im Allgemeinen recht kompliziert sein kann. Es gibt Fälle, bei denen weder in der einen noch in der anderen Richtung eine funktionale Abhängigkeit vorliegt. In diesen Fällen lassen sich die Abhängigkeiten zwischen den Strukturen nur durch bestimmte Konsistenzbedingungen ausdrücken. Andererseits ist in der Praxis der Zusammenhang häufig wesentlich einfacher. Sehr häufig kommt es z.B. vor, dass die editierbare Struktur (fast) ein Teilbaum der semantischen ist, wobei die editierbare Struktur evtl. um zusätzliche Layoutattribute erweitert sein kann. In anderen Fällen, in denen keine Layoutattribute benötigt werden, können editierbare und semantische Struktur vollkommen identisch sein.

Hieraus wird klar, dass der Mechanismus zur Kopplung von Strukturen einerseits bei sehr ähnlichen Strukturen keinen großen Spezifikationsaufwand verursachen darf, er andererseits aber auch bei komplizierten, nichtfunktionalen Abhängigkeiten einsetzbar sein muss. Nachfolgend werden anhand eines praktisch relevanten Beispiels zunächst die Anforderungen diskutiert, bevor das Spezifikationskonzept vorgestellt wird.

3.3.1 Anforderungen

Aus der Tatsache, dass sich in vielen Fällen editierbare und semantische Struktur kaum unterscheiden, lässt sich bereits die erste Anforderung ablei-

```
CLASS Root {
  classes: SUB Class*;
  associations: SUB Association*;
}

CLASS Class {
  name: VAL VLString;
  attributes: SUB Attribute*;
}

CLASS Association {
  from: REF Class;
  to: REF Class;
}
```

Abbildung 3.11: DSSL-Syntax für Klassendiagramme

ten: Gleiche Strukturen sind als Normalfall und Abweichungen als Ausnahme zu betrachten. Demnach sollte der Spezifikationsaufwand desto kleiner sein, je ähnlicher die zu koppelnden Strukturen sind.

Die weiteren Anforderungen möchte ich anhand einer vereinfachten Variante von UML-Klassendiagrammen diskutieren. An diesem sowohl realistischen als auch anspruchsvollen Beispiel lassen sich viele Aspekte verdeutlichen, die beim Entwurf des Spezifikationskonzepts zu berücksichtigen waren.

Abbildung 3.11 zeigt eine stark vereinfachte semantische Syntax für UML-Klassendiagramme. Semantisch werden Klassendiagramme durch eine Menge von Klassendefinitionen und eine Menge von Assoziationen zwischen Klassen repräsentiert. Des Weiteren besitzen Klassendefinitionen Unterstrukturen, was hier dadurch angedeutet ist, dass sie Attributdefinitionen enthalten.

Diese semantische Struktur kann laut UML-Standard [41] durch mehrere sich ergänzende Klassendiagramme repräsentiert werden. In einem Klassendiagramm müssen daher nicht alle Klassen vorkommen. Andererseits darf die gleiche Klasse laut UML-Standard in einem Diagramm durchaus mehrfach vorkommen. Das ist nützlich, um übermäßig lange Vererbungs- oder Assoziationslinien zu vermeiden. Natürlich müssen die verschiedenen Vorkommen der gleichen Klasse konsistent zueinander sein.

Um den Unterschied zwischen Klassen in der semantischen Struktur und dem Auftreten von Klassen in Klassendiagrammen deutlich zu machen, nenne ich die letzteren im Folgenden Klassenrepräsentanten. Nach dieser Sprechweise kann also ein Klassendiagramm mehrere Repräsentanten der gleichen Klasse enthalten. Abbildung 3.12 zeigt ein Beispiel für diese Situation. Die

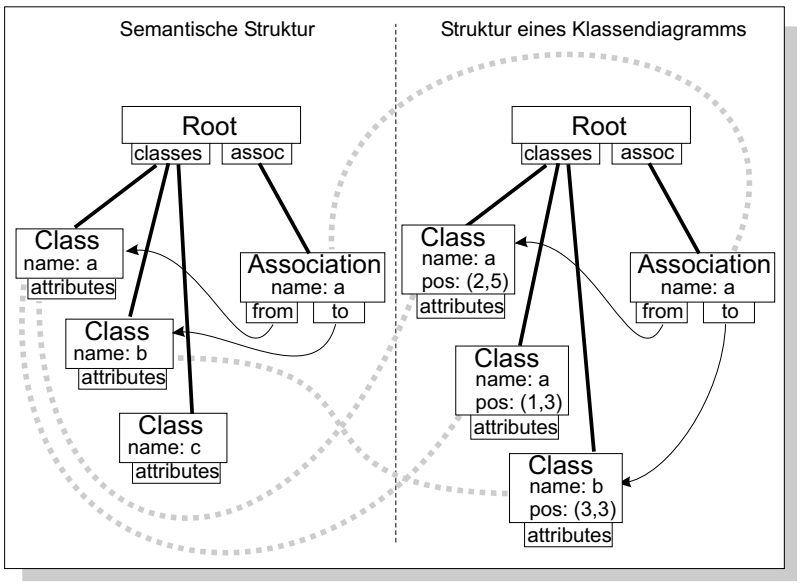


Abbildung 3.12: Zusammenhang zwischen semantischer und editierbarer Struktur am Beispiel von Klassendiagrammen

Beziehung zwischen den Repräsentanten und den repräsentierten Objekten ist durch gepunktete Linien dargestellt. Die semantische Struktur enthält die drei Klassen a, b und c sowie eine Assoziation zwischen den Klassen a und b. Die editierbare Struktur gehört zu einem Klassendiagramm, das zwei Repräsentanten der Klasse a und einen Repräsentanten der Klasse b enthält. Klasse c kommt darin nicht vor. Ferner enthält die editierbare Struktur einen Assoziations-Repräsentanten. Es ist semantisch irrelevant, mit welchem der beiden a-Repräsentanten der Assoziations-Repräsentant verbunden ist, aber natürlich nicht irrelevant für die grafische Darstellung.

An diesem Beispiel lassen sich einige wichtige Beobachtungen machen. Zunächst erkennt man, dass die editierbare Struktur sowohl kontinuierliche als auch diskrete Layoutinformationen enthält. Eine kontinuierliche Layoutinformation ist die Position der Klassen-Repräsentanten. Diskrete Layoutinformationen sind z.B. die Anzahl der Repräsentanten einer Klasse oder die Information, mit welchen Klassen-Repräsentanten ein Assoziations-Repräsentant verbunden ist. Trotz dieser Tatsache ist die Syntax beider Strukturen sehr ähnlich. Fügt man der Syntax aus Abbildung 3.11 noch Layoutattribute für die Position von Klassen hinzu, kann sie als Syntax für die editierbare Struktur dienen. Das Auftreten mehrerer Repräsentanten verursacht Redundanz in der editierbaren Struktur. Die semantische Struktur enthält dagegen keine Redundanz.

Zwischen semantischer und editierbarer Struktur gelten offenbar bestimmte

Konsistenzbedingungen. Wird die semantische Struktur geändert, kann die Konsistenz wiederhergestellt werden, indem die editierbare Struktur der semantischen angepasst wird. Dazu muss lediglich der neue Zustand der semantischen Struktur, aber nicht die Änderungsoperation betrachtet werden. Hat z.B. ein Klassenrepräsentant der editierbaren Struktur kein Gegenstück in der semantischen Struktur, muss er offenbar gelöscht werden.

Nach einer Änderung der editierbaren Struktur lässt sich die Konsistenz im Allgemeinen *nicht* auf diese Art wiederherstellen. Wird beispielsweise der Wert des `name`-Attributs eines Klassen-Repräsentanten geändert, ist die editierbare Struktur widersprüchlich, d.h. es ist nicht klar, ob und wie die semantische Struktur geändert werden muss. Vielmehr muss in diesem Fall die Änderungsoperation selbst betrachtet werden, da nur sie die Intention des Sprachbenutzers ausdrückt. Durch die Änderung des Namens möchte der Sprachbenutzer offenbar den Namen der repräsentierten Klasse ändern, d.h. dieser muss auch in der semantischen Struktur sowie im Kontext der anderen Repräsentanten geändert werden.

Ein weiterer ähnlicher Fall liegt vor, wenn der Sprachanwender einen Klassenrepräsentanten löscht. Das könnte zum einen bedeuten, dass tatsächlich nur der Repräsentant gelöscht werden soll, während die Klasse in der semantischen Struktur sowie evtl. weitere Repräsentanten erhalten bleiben sollen. Andererseits könnte der Sprachanwender aber auch die Klasse selbst mit allen noch vorhandenen Repräsentanten löschen wollen. Eine dritte Interpretation wäre, dass die Klasse in der semantischen Struktur genau dann gelöscht werden soll, wenn es sich bei dem gelöschten Klassen-Repräsentanten um den letzten seiner Art handelt. Im Allgemeinen ist die Interpretation eine Entwurfsentscheidung des Editors, die also im Kontext des Kopplungsmodells spezifizierbar sein muss.

Aus der vorangegangenen Diskussion ergibt sich die Erkenntnis, dass das Kopplungsmodell unsymmetrisch sein sollte. Bei Änderungen der editierbaren Struktur muss die Änderungsoperation betrachtet werden, um die Intention des Sprachbenutzers zu erkennen. Diese muss auf die semantische Struktur übertragen werden. Bei Änderungen der semantischen Struktur müssen bestimmte Konsistenzbedingungen zwischen beiden Strukturen wiederhergestellt werden. Dabei ist es unwichtig, wie die semantische Struktur geändert wurde.

3.3.2 Spezifikation der Kopplung

Nachfolgend wird das in DEViL realisierte Kopplungskonzept vorgestellt. Da es nicht auf die Kopplung von editierbarer und semantischer Struktur beschränkt ist, werden die beteiligten Strukturen im Folgenden *Basis-Struktur* und *abgeleitete Struktur* genannt. Im Fall der Kopplung von editierbarer und semantischer Struktur spielt die semantische Struktur die Rolle der Basis-Struktur und die editierbare Struktur die Rolle der abgeleiteten Struktur. Ein Beispiel dafür, dass die Kopplung nicht auf diesen Anwendungsfall beschränkt ist, wird in Abschnitt 3.4.7 vorgestellt.

Die Begriffe *Basis-Struktur* und *abgeleitete Struktur* verdeutlichen die Asymmetrie der Kopplung: Wird die Basis-Struktur verändert, wird die abgeleitete Struktur an diese Veränderung angepasst. Eine Veränderung der abgeleiteten Struktur kann sich lediglich durch direkte Übertragung der Änderungsoperation auf die Basis-Struktur auswirken.

Zur Spezifikation gekoppelter Strukturen müssen die folgenden Dinge spezifiziert werden.

- Die Syntax beider Strukturen,
- die Anpassung der abgeleiteten Struktur an die Basis-Struktur und
- die Übertragung von Änderungen an abgeleiteten Strukturen an die Basis-Struktur.

Spezifikation der Syntax Prinzipiell lässt sich sowohl die Syntax der Basis-Struktur als auch die Syntax der abgeleiteten Struktur direkt mittels DSSL spezifizieren. Ein Ziel der Kopplungsmethode ist jedoch, dass wenig Spezifikationsaufwand entsteht, wenn Basis- und abgeleitete Struktur ähnlich oder gleich sind. Daher wird in DEViL auch die Syntax der abgeleiteten Struktur aus der Syntax der Basis-Struktur abgeleitet, falls keine anderslautenden Spezifikationen existieren. Für den Fall der editierbaren und semantischen Struktur heißt dies, dass die semantische Struktur vollständig in DSSL spezifiziert wird, während lediglich dann Definitionen für Klassen der editierbaren Struktur angegeben werden müssen, wenn sie sich von denen der semantischen Struktur unterscheiden. Abbildung 3.13a zeigt die Syntax-Definition einer abgeleiteten Struktur für Klassendiagramme. Die Struktur-Definition basiert auf der Syntax in Abbildung 3.11.

```
DERIVED ClassDiagram ROOT Root {

  CLASS Class {
    name: VAL VLString;
    attributes: SUB Attribute*;
    position: VAL VLPoint;
  }

  Root.classes.
    createMissingRep = 0;
  Root.classes.
    removeMultipleRep = 0;
  Root.associations.
    createMissingRep = 0;
  Root.associations.
    removeMultiplerep = 0;
}
```

(a) Spezifikation der abgeleiteten Struktur

```
CLASS ::ClassDiagram::Root {
  classes: SUB ::ClassDiagram::Class*;
  associations:
    SUB ::ClassDiagram::Association*;
}

CLASS ::ClassDiagram::Class {
  name: VAL VLString;
  attributes:
    SUB ::ClassDiagram::Attribute*;
  position: VAL VLPoint;
}

CLASS ::ClassDiagram::Association {
  from: REF ::ClassDiagram::Class;
  to: REF ::ClassDiagram::Class;
}
```

(b) Die sich aus (a) ergebende Syntax

Abbildung 3.13: Definition der Editor-Syntax für Klassendiagramme

Im einfachsten Fall wird in einer solchen Spezifikation lediglich die Wurzel-Klasse der Basis-Syntax und der Namensraum der abgeleiteten Syntax spezifiziert. In Abbildung 3.13a ist `Root` die Wurzel-Klasse der Basis-Syntax und `ClassDiagram` der Namensraum der abgeleiteten Syntax. Ohne anderslautende Spezifikation werden zu jeder Klasse der Basis-Syntax äquivalente Klassen im Namensraum der abgeleiteten Syntax erzeugt. In unserem Fall enthielte die abgeleitete Struktur demnach die Klassen `ClassDiagram::Root`, `ClassDiagram::Class` und `ClassDiagram::Association`. Die korrespondierenden Klassen haben aber keineswegs den gleichen Inhalt wie die entsprechenden Basis-Klassen. Während das Attribut `from` der Klasse `::Association` Objekte der Klasse `::Class` referenziert, referenziert das Attribut `from` der Klasse `::ClassDiagram::Association` Objekte der Klasse `::ClassDiagram::Class`.

Die in Abbildung 3.13a angegebenen Klassendefinitionen überschreiben die automatisch abgeleiteten Klassen. Die sich ergebende abgeleitete Syntax ist in Abbildung 3.13b dargestellt. In diesem Fall wurden den Klassen lediglich weitere Attribute hinzugefügt. Auf die gleiche Weise lassen sich auch Attribute entfernen, Attributtypen ändern und ganze Klassen neu hinzufügen. So können Strukturen definiert werden, die beliebig stark von der Basis-Syntax abweichen. Anwendungsbeispiele für all diese Fälle werden im Abschnitt 3.4 vorgestellt.

Kopplung Basis-Struktur → abgeleitete Struktur Wann immer die Basis-Struktur sich ändert wird ein Mechanismus in Gang gesetzt, der die abgeleiteten Strukturen an die neue Basis-Struktur angleicht. Durch die in Abbildung 3.13a angegebenen Optionen wie `Root.classes.createMissingRep=0` können die Details dieser Anpassung beeinflusst werden. Je nach Optionen ist die abgeleitete Struktur ein Spiegelbild der Basis-Struktur oder es werden nur bestimmte Konsistenzbedingungen sichergestellt.

Prinzipiell werden zwei Strukturobjekte genau dann gekoppelt, wenn sie in *Repräsentierter-Repräsentant*-Beziehung (oder kurz RR-Beziehung) stehen. Jeder Sprachkonstrukt-Knoten der abgeleiteten Struktur hat zur Speicherung dieser Relation eine Referenz auf den von ihm repräsentierten Sprachkonstrukt-Knoten der Basis-Struktur. In Abbildung 3.12 ist diese Beziehung durch die gepunkteten Linien dargestellt. Ist S ein Sprachkonstrukt-Knoten der abgeleiteten Struktur, bezeichne $S.\mathit{base}$ den von ihm repräsentierten Sprachkonstrukt-Knoten in der Basis-Struktur.

Die RR-Relation lässt sich auf Attribut-Knoten erweitern. Zwei Attribut-Knoten stehen genau dann in RR-Relation, wenn deren Väter (Sprachkonstrukt-Knoten) in RR-Beziehung stehen und die Attribut-Knoten den gleichen Namen haben. Ist S ein Attribut-Knoten der abgeleiteten Struktur, wird der korrespondierende Knoten in der Basis-Struktur ebenfalls mit $S.\mathit{base}$ bezeichnet. Wenn das Attribut S den Namen a hat, ist $S.\mathit{base}$ also gleichbedeutend mit $S.\mathit{parent}.\mathit{base}.a$.

Zur automatischen Anpassung der abgeleiteten Struktur werden auf die Knoten der abgeleiteten Struktur bestimmte Anpassungsoperationen angewandt. Die sieben in DEViL vorhandenen Anpassungsschemata sind in Tabelle 3.1 zusammengefasst. Jedes Schema ist nur auf Knoten einer bestimmten Art anwendbar. Das Schema `RemoveAbandoned` lässt sich z.B. nur auf Sprachkonstrukt-Knoten anwenden.

Ein Anpassungsschema bewirkt, dass ein bestimmter struktureller Zusammenhang zwischen Basis- und abgeleiteter Struktur erzwungen wird. Wenn diese Eigenschaft nicht bereits erfüllt ist, wird die abgeleitete Struktur mit einem fest vorgegebenen Algorithmus entsprechend geändert.

- `RemoveAbandoned(S)` stellt sicher, dass der Sprachkonstrukt-Knoten S mit einem Knoten des Basis-Modells in RR-Beziehung steht. Falls dies nicht gilt wird S gelöscht.

Tabelle 3.1: Anpassungsschemata zur Kopplung von Strukturen

Name	Anwendbar auf	Erzwungene Eigenschaft
`RemoveAbandoned`	`SK-Knoten`	$S.\mathit{base} \neq \mathit{null}$
`MoveToCorrectList`	`SK-Knoten`	$S.\mathit{parent}.\mathit{base} = S.\mathit{base}.\mathit{parent}$
`RemoveMultipleRep`	`SUB-Knoten`	$\forall x, y \in S.\mathit{children} : x \neq y \Rightarrow x.\mathit{base} \neq y.\mathit{base}$
`CreateMissingRep`	`SUB-Knoten`	$\forall x \in S.\mathit{base}.\mathit{children} : \exists y \in S.\mathit{children} : y.\mathit{base} = x$
`OrderRep`	`SUB-Knoten`	$\forall x, y \in S.\mathit{children} : x.\mathit{base} \lhd y.\mathit{base} \Rightarrow x \lhd y$
`SyncVal`	`VAL-Knoten`	$S.\mathit{value} = S.\mathit{base}.\mathit{value}$
`SyncRef`	`REF-Knoten`	$S.\mathit{value}.\mathit{base} = S.\mathit{base}.\mathit{value}$

- `MoveToCorrectList(S)` stellt sicher, dass der Sprachkonstrukt-Knoten S Kind eines „passenden“ SUB-Knotens ist. Ein SUB-Knoten ist dann „passend“, wenn er den SUB-Knoten repräsentiert, in dem $S.\mathit{base}$ enthalten ist. Falls diese Eigenschaft nicht erfüllt ist, wird S an eine in diesem Sinne „passende“ Stelle verschoben. Gibt es mehrere passende Stellen, so wird der Kandidat K bevorzugt, bei dem der Abstand zwischen S und $TGV(K, S)$ minimal ist. Das Auswahlkriterium stellt sicher, dass Quelle und Ziel der Verschiebeoperation möglichst dicht beisammen sind, d.h. die „kleinstmögliche“ Änderung durchgeführt wird. (Zur Begründung dieses Details siehe Abschnitt 3.4.4.) Enthält die abgeleitete Struktur keine passende Stelle wird S gelöscht.
- `RemoveMultipleRep(S)` stellt sicher, dass S nicht mehrere Repräsentanten des gleichen Objekts enthält. Falls doch, werden alle bis auf einen gelöscht.
- `CreateMissingRep(S)` stellt sicher, dass der SUB-Knoten S Gegenstücke zu allen Elementen von $S.\mathit{base}$ besitzt. Falls dies nicht gilt werden die fehlenden Repräsentanten neu erstellt.
- `OrderRep(S)` stellt sicher, dass die Reihenfolge der Elemente des SUB-Knotens S denen ihrer Repräsentanten in $S.\mathit{base}$ entspricht. Elemente in S, die keine Elemente aus $S.\mathit{base}$ repräsentieren, werden dabei nicht berücksichtigt. Falls die Eigenschaft nicht erfüllt ist, werden die Elemente in S umsortiert.
- `SyncVal(S)` stellt sicher, dass die VAL-Knoten S und $S.\mathit{base}$ den gleichen Wert enthalten. Falls dies nicht gilt wird S der Wert von $S.\mathit{base}$ zugewiesen.

- `SyncRef(S)` stellt sicher, dass die Referenz S ein „passendes“ Objekt referenziert. Ein Objekt ist „passend“, wenn es mit dem Wert von $S.base$ in RR-Beziehung steht. Wenn die Bedingung nicht erfüllt ist, wird ein passender Repräsentant ausgewählt und der Wert von S geändert. Gibt es mehrere passende Repräsentanten, so wird analog zu `MoveToCorrectList` der Kandidat K bevorzugt, bei dem der Abstand zwischen S und $TGV(K, S)$ minimal ist. (Auch hierzu findet sich die Begründung in Abschnitt 3.4.4.) Gibt es keinen passenden Repräsentanten, wird S der Wert *null* zugewiesen.

Die Anpassungoperationen einzelner Klassen oder Attribute können individuell deaktiviert werden. Beispielsweise wird durch `Root.classes.createMissingRep=0` spezifiziert, dass die Anpassungoperation `CreateMissingRep` des SUB-Attributs `Root.classes` deaktiviert werden soll.

Die Wirkung jeder einzelnen Anpassungsoperation lässt sich unabhängig von anderen Anpassungsoperationen betrachten. Das Gesamtergebnis hängt aber von der Reihenfolge der Operationen ab. Es macht z.B. keinen Sinn `CreateMissingRep` nach `OrderRep` auszuführen, weil damit die durch `OrderRep` hergestellte strukturelle Eigenschaft wieder zerstört werden könnte. In DEViL werden die Operationen in der Reihenfolge angewandt, in der sie in Tabelle 3.1 aufgeführt sind. Durch die Reihenfolge ist sichergestellt, dass einmal hergestellte Eigenschaften erhalten bleiben und dass die Anpassungsoperationen die gewünschte Wirkung haben.

Die Auswirkungen der Anpassungsschemata sehen auf den ersten Blick kompliziert aus. Der Sprachentwickler braucht aber normalerweise nicht alle Details der Realisierung zu kennen. Für ihn genügt es z.B. zu wissen, dass das Anpassungsschema `CreateMissingRep` fehlende Sprachkonstrukte in der abgeleiteten Struktur ggf. neu erzeugt. Sie sind so entworfen, dass sie weitgehend den Erwartungen des Sprachentwicklers bzw. des Sprachanwenders entsprechen.

Als Voreinstellung sind zunächst alle Anpassungsoperationen aktiviert. In diesem Fall werden alle in RR-Relation stehenden Strukturen vollständig gekoppelt. Es lässt sich zeigen, dass dann die abgeleitete Struktur ein Spiegelbild der Basis-Struktur ist, falls deren Grammatiken gleich sind (siehe Abschnitt 3.3.3). Durch Deaktivierung einzelner Anpassungsoperationen lassen sich die Strukturen stellenweise entkoppeln. Hierdurch lässt sich z.B. sehr einfach ausdrücken, dass Klassendiagramme nicht alle Klassen und Assoziatio-

```
proc user::deleteObject(objectNode) {
  basic::deleteObject(objectNode)
  if(objectNode.base != null) {
    basic::deleteObject(objectNode.base)
  }
}
```

Abbildung 3.14: Pseudocode für eine Funktion zur Änderung der editierbaren Struktur

nen des semantischen Modells enthalten müssen. Tatsächlich ist durch die Spezifikation in Abbildung 3.13a die Kopplung für das in Abschnitt 3.3.1 diskutierte Einführungsbeispiel vollständig spezifiziert. In komplizierteren Fällen können die Anpassungsoperationen durch handimplementierte Versionen ersetzt werden.

Kopplung abgeleitete Struktur → Basis-Struktur Editiert der Sprachanwender die abgeleitete Struktur, beabsichtigt er damit unter Umständen, die Basis-Struktur zu ändern. In diesem Fall muss also nicht nur die eigentliche Änderung, sondern auch eine entsprechende Änderung in der Basis-Struktur durchgeführt werden. Nach dem Entwurfsprinzip „Kopplung ist der Normalfall, Entkopplung die Ausnahme“ wird ohne Zusatzspezifikation jede Änderung falls möglich direkt auf die Basis-Struktur übertragen.

Auch dieses Standard-Verhalten kann überschrieben werden. Dazu gibt es die in Abbildung 3.7 auf Seite 86 gezeigten elementaren Änderungsfunktionen in zwei Varianten. Die Varianten im Namespace `user` werden von der Benutzungsschnittstelle als Reaktion auf Benutzeroperationen aufgerufen. Die Funktionen im Namespace `basic` führen entsprechende Strukturänderungen durch.

Die Standard-Implementierungen der `user`-Funktionen bilden die Editieroperationen auf die entsprechenden Operationen im Namespace `basic` ab und übertragen sie falls möglich auch auf die Basis-Struktur. In Abbildung 3.14 ist die Implementierung der Standard-Funktion `user::deleteObject` zu sehen. Diese löscht nicht nur den angegebenen Sprachkonstrukt-Knoten, sondern - falls vorhanden - auch den repräsentierten Knoten.

Durch das Überschreiben der vordefinierten Implementierungen lässt sich die Kopplung in dieser Richtung flexibel anpassen. Dies ist z.B. für das Löschen von Klassen-Repräsentanten sinnvoll. Wenn der Sprachanwender einen Klassen-Repräsentanten löscht, sind wie oben bereits erwähnt drei verschie-

dene Interpretationen denkbar. (1) Es soll tatsächlich nur der Repräsentant gelöscht werden. (2) Es soll auch das semantische Objekt zusammen mit allen anderen Repräsentanten gelöscht werden. (3) Das semantische Objekt soll nur dann gelöscht werden, wenn es sich um den letzten Repräsentanten dieses Objektes gehandelt hat. Die in DEViL enthaltene Standardimplementierung fragt den Benutzer nach seiner Intention. Durch das Überschreiben der Lösch-Operation kann das Verhalten des Editors bzgl. dieses Aspekts angepasst werden. In MetaEdit+ (siehe Abschnitt 2.5.4) lässt sich z.B. global eine der drei Interpretationen auswählen.

3.3.3 Vollständigkeit der Anpassungsschemata

Die Menge der Anpassungsschemata ist insofern vollständig, als dass bei Aktivierung aller Anpassungsschemata die gekoppelten Strukturen isomorph sind. Die Voraussetzung dafür ist allerdings, dass die zugrunde liegenden Grammatiken gleich sind.

Den Beweis für diese Aussage möchte ich nachfolgend grob darstellen, ohne auf alle Einzelheiten genau einzugehen.

Die obige Aussage lautet exakt wie folgt.

> *Satz 1:* Sei D eine abgeleitete Struktur und B die entsprechende Basis-Struktur, wobei $\mathit{Nodes}(D)$ und $\mathit{Nodes}(B)$ die jeweiligen Knotenmengen der implementierten Datenstruktur seien. Wir setzen weiter voraus, dass die Wurzeln von D und B in RR-Beziehung stehen und dass die Klassen aller in RR-Beziehung stehender Knoten gleich sind. Ferner seien alle in Tabelle 3.1 aufgeführten Eigenschaften erfüllt. Dann definiert die Funktion $f : \mathit{Nodes}(D) \rightarrow \mathit{Nodes}(B)$, $f(x) := x.\mathit{base}$ einen Isomorphismus zwischen D und B.

Isomorphie zweier Graphen D und B bedeutet formal, dass es eine bijektive Abbildung $f : \mathit{Nodes}(D) \rightarrow \mathit{Nodes}(B)$ gibt, so dass $E(x, y)$ in D genau dann gilt, wenn $E(f(x), f(y))$ in B gilt. $E(x, y)$ bedeutet hierbei, dass x und y mit einer Kante verbunden sind.

DSSL-Instanzen sind allerdings keine einfachen Graphen, sondern sie enthalten verschiedenartige Kanten. Isomorphie von DSSL-Strukturen bedeutet demnach, dass die oben genannte Eigenschaft für alle nachfolgend aufgezählten Relationen E gilt.

- *Member*$_a(x, y)$ bedeute, dass y ein Attribut des Sprachkonstrukt-Knotens x mit Namen a ist, dass also gilt $x.a = y$.
- *Child*(x, y) bedeute, dass y Unterknoten des SUB-Knotens x ist, dass also gilt $y.\mathit{parent} = x$.
- *Next*(x, y) bedeute, dass x und y im gleichen SUB*-Attribut enthalten sind und dass y in der Listenfolge direkt nach x kommt, dass also gilt $x.\mathit{next} = y$.
- *Ref*(x, y) bedeute, dass x ein REF-Attribut mit Wert y ist, dass also gilt $x.\mathit{value} = y$.

Zum Beweis von Satz 1 muss gezeigt werden, dass f tatsächlich bijektiv ist und dass alle oben genannten Relationen E invariant bzgl. f sind, d.h. dass jeweils gilt $E(x, y) \Leftrightarrow E(f(x), f(y))$.

Dazu benötigen wir zunächst einen Hilfssatz.

> *Hilfssatz 1:* Seien d und b zwei in RR-Relation stehende SUB-Knoten und es seien für alle Knoten in $\{d\} \cup d.\mathit{children}$ alle in Tabelle 3.1 aufgeführten Eigenschaften erfüllt. Dann ist die Funktion $g : d.\mathit{children} \rightarrow b.\mathit{children}$, $g(x) := x.\mathit{base}$ bijektiv und die *Next*-Relation invariant bzgl. g.

Zum Beweis betrachten wir zunächst die Eigenschaften der Funktion g. Sie hat tatsächlich den Bildbereich $b.\mathit{children}$, da `MoveToCorrectList` gilt. Sie ist total, da `RemoveAbandoned` gilt. Sie ist injektiv, da `RemoveMultipleRep` gilt. Und sie ist surjektiv, da `CreateMissingRep` gilt. Hieraus folgt, dass g bijektiv ist. Aus `OrderRep` zusammen mit der Bijektivität folgt schließlich, dass g invariant gegen die *Next*-Relation ist.

Mit Hilfe des Hilfssatzes 1 kann nun der Satz 1 bewiesen werden. Der Beweis erfolgt durch vollständige Induktion über die Knotentiefe in den gekoppelten Bäumen. Die Induktionsannahme lautet wie folgt.

> Seien *Nodes*$_n(D)$ und *Nodes*$_n(B)$ die Mengen der Knoten von D bzw. B, deren Abstand zur Wurzel höchstens n ist. Sei die Funktion $f_n : \mathit{Nodes}_n(D) \rightarrow \mathit{Nodes}_n(B)$ definiert durch $f_n(x) := x.\mathit{base}$. Dann ist f_n eine bijektive Funktion und die Relationen *Member*$_a$, *Child* und *Next* sind invariant bzgl. f_n.

Als Induktionsanfang betrachten wir $n = 0$. $Nodes_n(D)$ und $Nodes_n(B)$ sind in diesem Fall einelementige Mengen, die laut Voraussetzung des Satzes 1 in RR-Beziehung stehen. Damit ist f_0 eine bijektive Funktion und die Aussage für diesen Fall bewiesen.

Sei n nun beliebig aber fest und sei die Aussage für n bereits bewiesen. Ist n gerade, so kommen durch den Induktionsschritt Attribut-Knoten hinzu. Da nach Voraussetzung des Satzes 1 die Klassen der in RR-Relation stehenden Sprachkonstrukt-Knoten gleich sind, kommen auf beiden Seiten sich entsprechende Attribut-Knoten hinzu, die nach Definition der *base*-Funktion auch in RR-Relation stehen. Daher ist auch f_{n+1} bijektiv und alle Relationen (insbesondere die $Member_a$-Relationen) sind auch bzgl. f_{n+1} invariant.

Ist n ungerade, so kommen durch den Induktionsschritt die Kinder von SUB-Knoten hinzu. Nach Hilfssatz 1 sind die dort definierten Funktionen g Bijektionen, die invariant bzgl. der *Next*-Relation sind. Da die Werte- und Bildbereiche von f_n sowie der g-Funktionen disjunkt sind, ergibt die Vereinigung der Funktionen wieder eine bijektive Funktion. Die Vereinigung dieser Funktionen entspricht der Funktion f_{n+1}. Aus den Eigenschaften der g-Funktionen folgt, dass die Relationen (insbesondere *Child* und *Next*) auch bzgl. f_{n+1} invariant sind. Somit ist die Induktionsaussage auch für diesen Fall gültig.

Durch die Induktion haben wir nun gezeigt, dass f eine bijektive Abbildung ist und dass die Relationen $Member_a$, *Child* und *Next* invariant bzgl. f sind. Es fehlt noch die Invarianz der Relation *Ref*. Diese folgt aber unmittelbar aus der Eigenschaft `SyncRef`. Damit ist Satz 1 endgültig bewiesen.

Es ist anzumerken, dass im Beweis das Anpassungsschema `SyncVal` nicht benötigt wird, da Satz 1 keine Aussage über die Werte von VAL-Attributen macht. Durch das Hinzfügen von `SyncVal` wird erreicht, dass die Strukturen nicht nur isomorph zueinander sind, sondern dass in RR-Relation stehende VAL-Knoten auch gleiche Werte besitzen.

Wie gezeigt lässt sich die Vollständigkeit der Anpassungsschemata allein auf Basis der geforderten strukturellen Eigenschaften beweisen. Um die totale Korrektheit der Kopplung zu begründen müsste aber zusätzlich gezeigt werden, dass die Hintereinanderausführung der Einzeloperationen zu einem Ergebnis führt, bei dem am Ende alle strukturellen Eigenschaften gelten. Dies hängt natürlich davon ab, *wie* die strukturellen Eigenschaften durch Änderung der Struktur herstellt werden. Zum Beweis könnten die Anpassungsschemata als Graphtransformationsregeln modelliert werden. Eine ausführ-

liche Betrachtung dieses Themas würde jedoch den Rahmen dieser Arbeit sprengen.

3.4 Anwendungsbeispiele für gekoppelte Strukturen

Der vorgestellte Kopplungsmechanismus ermöglicht es, die Implementierung konkreter Interaktionsmechanismen und die Spezifikation der abstrakten Wirkung vollständig zu trennen. Die Kopplung zwischen Basis-Struktur und abgeleiteter Struktur braucht so in der Sicht-Implementierung nicht berücksichtigt zu werden.

Im Abschnitt 3.3.1 wurde bereits ein Anwendungsbeispiel für den Kopplungsmechanismus vorgestellt, der dessen Entwurf begründet. Nachfolgend sollen weitere praxisrelevante Einsatzszenarios diskutiert werden, um zu zeigen, dass die Methode wirksam und zweckmäßig ist. Jedes der folgenden Beispiele steht für eine Klasse von Phänomenen, die in vielen visuellen Sprachen auftreten. Jedes der Phänomene würde Probleme bereiten, wenn die beteiligten Strukturen nicht separat repräsentiert und bedarfsgerecht gekoppelt würden.

Fast alle Beispiele behandeln die Kopplung von semantischen und editierbaren Strukturen. Das letzte Beispiel aber beschreibt einen Anwendungsfall, in dem auch die abgeleitete Struktur semantisch relevant ist.

3.4.1 Graphen mit mehreren Layouts

Häufig unterscheiden sich semantische und editierbare Strukturen nur dadurch, dass die editierbare Struktur zusätzliche Attribute enthält, die das Layout der grafischen Darstellung beschreiben. Ein Beispiel dafür sind Sichten auf Graphen. In der semantischen Struktur werden Graphen durch eine Liste von Knoten und eine Liste von Kanten modelliert. Zur Darstellung müssen die Knoten auf einer zweidimensionalen Arbeitsfläche positioniert werden. Während die semantische Struktur lediglich ein Attribut für den Namen des Knotens enthält, müssen Knoten in der editierbaren Struktur also zusätzlich ein Attribut mit den Layout-Koordinaten besitzen.

Soll es nur eine Sicht auf den Graphen geben ist es vertretbar, das Attribut für die Koordinaten in die semantische Struktur zu übernehmen und so die

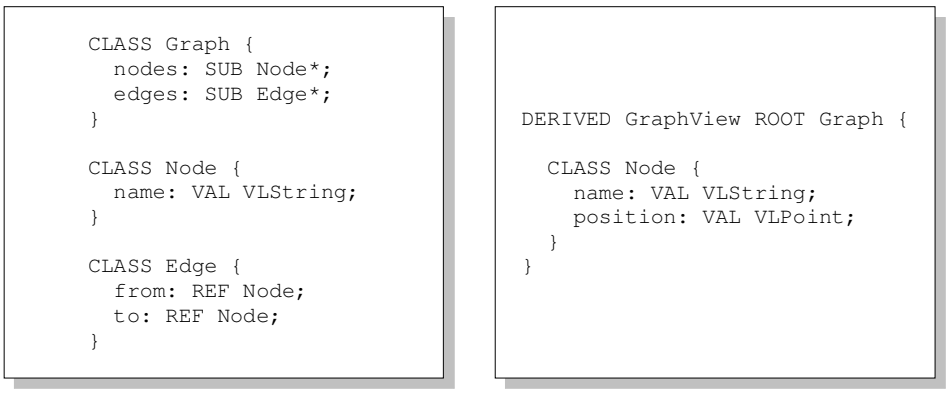
editierbare und semantische Struktur zu vereinheitlichen. Der einzige Nachteil wäre, dass die semantische Struktur, die z.B. als Schnittstelle zur Weiterverarbeitung dient, irrelevante Informationen enthielte. Soll das visuelle Programm mehrere Sichten auf einen Graphen enthalten und soll jede Sicht ein individuelles Layout besitzen dürfen, ist die Vereinigung aber nicht mehr so einfach möglich.

Abbildung 3.15 zeigt die Spezifikation getrennter semantischer und editierbarer Strukturen. Es ist zu erkennen, dass nur wenig Mehraufwand erforderlich ist. Es müssen weder Anpassungsoperationen deaktiviert noch Standard-Implementierungen von Änderungsoperationen überschrieben werden.

Die Spezifikation bewirkt, dass die Syntax der semantischen und editierbaren Struktur bis auf das zusätzliche `position`-Attribut übereinstimmt. Die Anpassungsoperationen und Standard-Änderungsoperationen sorgen dafür, dass die Strukturen bis auf das zusätzliche `position`-Attribut vollständig gekoppelt werden. Wird beispielsweise ein neuer Knoten-Repräsentant in eine Sicht eingefügt, wird durch die Standardimplementierung der Einfügeoperation ein entsprechender semantischer Knoten in die semantische Struktur eingefügt. Im Verlauf der anschließenden Anpassung werden durch das `CreateMissingRep` Anpassungsschema auch in allen anderen Sichten Repräsentanten für den neuen Knoten erzeugt. Der Wert des `name`-Attributs wird dann durch das `SyncVal` Anpassungsschema auf den richtigen Wert gesetzt. Der Wert des `position`-Attributs bleibt in anderen Sichten zunächst undefiniert. Die Wahl eines passenden Wertes ist Aufgabe der Sicht-Implementierung. Da alle Sichten somit ungekoppelte `position`-Attribute besitzen, ist die Position von Knoten-Repräsentanten individuell bestimmbar. Der gleiche Mechanismus synchronisiert auch das Erzeugen von Kanten, das Verbinden von Kanten mit Start- und Endknoten sowie das Löschen von Strukturobjekten.

Auf diese Weise können beliebig viele Graph-Sichten mit individuellem Layout angelegt werden, indem neue `::GraphView::Graph`-Knoten erzeugt und in RR-Beziehung mit dem zu repräsentierenden Graphen gesetzt werden.

Es gibt viele Anwendungsbeispiele dieser Art, bei der die editierbare Struktur lediglich um individuelle Layouteigenschaften erweitert wird. Solche Layouteigenschaften sind nicht auf Positions- und Größenangaben beschränkt. Es ist auch vorstellbar, dass verschiedene Sichten individuelle Farben oder Beschreibungstexte enthalten, um bestimmte Aspekte des Programms hervorzuheben.



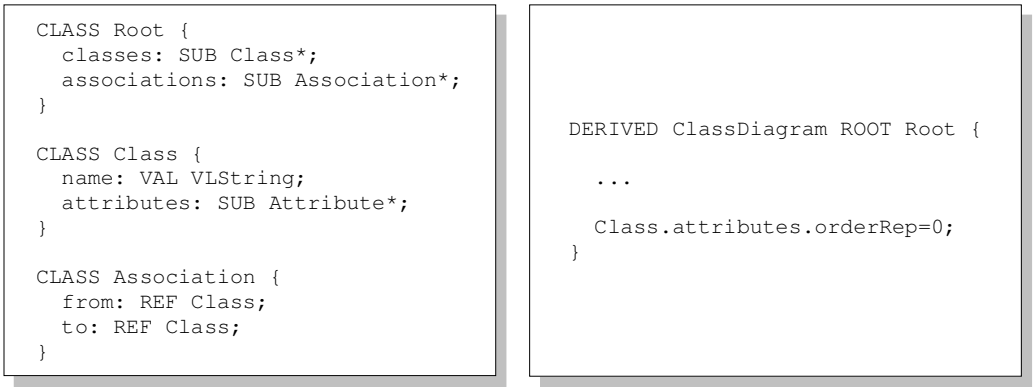
(a) Semantische Syntax (b) Spezifikation der abgeleiteten Struktur

Abbildung 3.15: Definition der Editor-Syntax für Graphen

3.4.2 Individuelle Reihenfolge von Attributen in UML

Häufig werden Objektmengen in grafischen Repräsentationen Reihenfolgen zugeordnet, die lediglich dem Layout dienen. In diesem Fall ist es evtl. erwünscht, dass in verschiedenen Repräsentationen die Reihenfolge individuell gewählt werden kann. Ein Beispiel ist die Menge von Attributen einer UML-Klasse. Gibt es mehrere Repräsentanten der gleichen Klasse, könnte die Sprachimplementierung es zulassen, dass die Reihenfolge überall individuell gewählt werden kann.

Konzeptionell ähnelt dieser Fall dem Graph-Beispiel aus Abschnitt 3.4.1, denn in beiden Fällen besitzen die Repräsentanten zusätzliche Layoutinformationen. Der technische Unterschied ist, dass die Attribut-Menge durch das DSSL-Listenkonstrukt modelliert wird, das den Elementen auch eine Reihenfolge zuordnet. In diesem Fall besitzen also bereits die Attribut-Knoten in der semantischen Struktur (ungewollt) eine Reihenfolge. Durch die Kopplung wird diese Reihenfolge standardmäßig auf die Repräsentanten übertragen. Damit die Reihenfolge bei allen Repräsentanten individuell gewählt werden kann, muss demnach kein Attribut hinzugefügt werden, sondern es muss durch Deaktivierung des `OrderRep`-Schemas die Kopplung abgeschwächt werden. Die dazu notwendige Spezifikation ist in Abbildung 3.16 dargestellt. Da alle anderen Anpassungsoperationen weiterhin aktiv sind, ist die Attributliste jedes Repräsentaten garantiert vollständig.



(a) Semantische Syntax (b) Spezifikation der abgeleiteten Struktur

Abbildung 3.16: Definition einer Editor-Syntax mit entkoppelter Reihenfolge von Attributen

3.4.3 Kommentare in UML-Diagrammen

Es kommt vor, dass Sichten grafische Elemente enthalten, die kein semantisches Objekt repräsentieren. Ein Beispiel dafür sind Kommentare in UML-Diagrammen.

Kommentare werden in UML durch ein Rechteck mit abgeknickter Ecke symbolisiert, können an beliebiger Stelle im Diagramm platziert und mittels Linien mit beliebigen anderen Objekten verbunden werden. Im Sinne der editierbaren Struktur ist ein Kommentar also kein Attribut eines bestimmten Programmkonstrukts, sondern ein eigenständiges Objekt. Laut UML-Standard [41] hängt die Repräsentation eines Kommentars in der semantischen Struktur vom Inhalt des Kommentars ab. Hier soll lediglich der Fall betrachtet werden, dass ein Kommentar eine informelle Beschreibung enthält und daher überhaupt kein Gegenstück in der semantischen Struktur besitzt.

Um eine Klasse für Kommentar-Symbole in der Editor-Syntax zu definieren, muss diese Klasse lediglich der entsprechenden Definition für die abgeleitete Struktur hinzugefügt werden. Zusätzlich muss die `RemoveAbandoned` Anpassungsoperation für diese Klasse deaktiviert werden. Weitere Spezifikationen sind nicht notwendig.

Da es keine Klasse `::Comment` in der semantischen Struktur gibt, wird beim Erzeugen eines Kommentars kein Gegenstück in der semantischen Struktur erzeugt und keine RR-Beziehung hergestellt. Dies ist auch der Grund, warum das `RemoveAbandoned` Anpassungsschema deaktiviert werden muss. Da

S.base nicht existiert, würde sonst ein neu erstellter Kommentar-Knoten bei der folgenden Strukturanpassung sofort wieder gelöscht.

Nach Green und Petre [19] können solche und ähnliche sekundären Ausdrucksmittel ein wichtiges Mittel darstellen, um die Absichten des Programmierers auszudrücken. Nicht-semantische visuelle Elemente können eingesetzt werden, um das Repertoire an sekundären Ausdrucksmitteln zu erweitern. Auf diese Weise lassen sich nicht nur Kommentare, sondern z.B. auch strukturierende visuelle Objekte wie Gruppen oder Absätze realisieren.

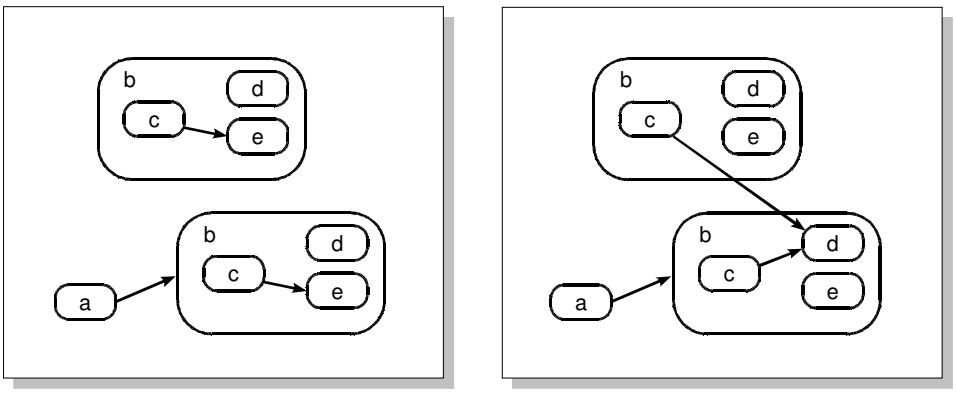
3.4.4 Zustände in Zustandsdiagrammen

Wie am Beispiel der Klassendiagramme demonstriert, können in einem Diagramm mehrere Repräsentanten des gleichen semantischen Objekts vorkommen. Diese Spracheigenschaft ist in UML hauptsächlich dafür gedacht, sehr lange Linienverbindungen zwischen weit entfernten Objekten zu vermeiden und so das Layout zu verbessern. Nachfolgend wird die Umsetzung dieser Spracheigenschaft anhand von UML-Zustandsdiagrammen diskutiert, denn diese weicht in einem wichtigen Punkt vom Klassendiagramm-Beispiel ab.

Analog zu Klassendiagrammen darf es in UML-Zustandsdiagrammen laut UML-Standard [41] mehrere Repräsentanten des gleichen Zustands geben. Sie sind am übereinstimmenden Namen zu erkennen. Ein Unterschied zum Klassendiagramm-Beispiel ist allerdings, dass Zustandsdiagramme grundsätzlich die vollständige Spezifikation des Zustandsautomaten enthalten. Es ist nicht vorgesehen, dass ein Zustandsautomat durch die Vereinigung mehrerer Diagramme spezifiziert werden kann.

Um den beschriebenen Effekt zu erzielen, muss das Anpassungsschema `RemoveMultipleRep` von SUB-Attributen, die Zustände speichern, deaktiviert werden. Zusätzlich müssen natürlich Interaktionsmittel vorhanden sein, die das Erstellen neuer Repräsentanten für bereits existierende Objekte gestatten. Auf struktureller Ebene bedeutet dies, dass ein neuer Repräsentanten-Knoten erzeugt wird und mit einem bereits existierenden Knoten der semantischen Struktur in RR-Beziehung gesetzt wird.

Wie gewohnt funktioniert die Anpassung aller Repräsentanten automatisch. Da das `CreateMissingRep` Anpassungsschema nicht deaktiviert ist, enthält jeder Zustands-Repräsentant mindestens einen Repräsentanten für jeden Unterzustand. Besitzt beispielsweise ein `XORSuperstate` zwei Unterzustände, ist dies an jedem seiner Repräsentanten zu erkennen. Somit zeigt jeder



(a) Ausgangszustand (b) Falsche Darstellung nach Änderung

Abbildung 3.17: Zustandsdiagramm mit mehreren Zustands-Repräsentanten

Zustands-Repräsentant das vollständige Verhalten des Zustands. Werden Eigenschaften eines Repräsentanten geändert, wird diese Änderung durch die Standard-Änderungsoperationen auf das semantische Objekt und durch die Anpassungsoperationen anschließend auf alle anderen Repräsentanten übertragen.

An diesem Beispiel lässt sich auch begründen, warum bei den Anpassungsschemata `MoveToCorrectList` und `SyncRef` bei alternativen Anpassungsmöglichkeiten die in Abschnitt 3.3.2 beschriebene TGV-Regel angewendet wird. Diese Regel besagt, dass die Anpassung der abgeleiteten Struktur möglichst lokal durchgeführt wird. Wird z.B. in der zu Abbildung 3.17a gehörenden semantischen Struktur das Ziel der in „b“ enthaltenen Transition von „e“ auf „d“ geändert, sollte jeder Transitions-Repräsentant das nächstgelegene „d“, und nicht wie in Abbildung 3.17b ein fremdes „d“ referenzieren. Das gleiche Argument gilt auch für das Verschieben von Sprachkonstrukt-Knoten in der semantischen Struktur.

Mit Klassen- und Zustandsdiagrammen wurden jetzt schon zwei Beispiele für das Auftreten mehrerer Repräsentanten eines semantischen Objekts im gleichen Diagramm diskutiert. Das Phänomen tritt aber nicht nur in UML, sondern auch in anderen Sprachen auf. Weiter unten werde ich auf Eigenschaften der Sprache *Streets* eingehen, die ebenfalls mit diesem Phänomen zusammenhängen.

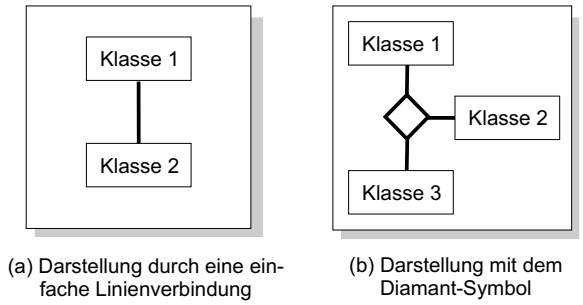


Abbildung 3.18: Darstellungsvarianten für Assoziationen in UML

3.4.5 Zwei Darstellungsarten für Assoziationen in UML

Es kommt vor, dass es alternative Darstellungsvarianten für bestimmte Sprachelemente gibt. In der Regel entscheidet der Sprachanwender bei der Erstellung des Programms, welche der Darstellungsvarianten verwendet werden soll. Diese Information ist eine Layoutentscheidung, die gespeichert werden muss. Darstellungsvarianten basieren häufig auf unterschiedlichen visuellen Konzepten, weshalb oft auch unterschiedliche Layoutinformationen gespeichert werden müssen. Um dies zu erreichen ist es zweckmäßig, für jede Darstellungsvariante eine eigene Klasse in der editierbaren Struktur einzuführen.

Ein relativ anspruchsvolles Beispiel für dieses Phänomen ist die Darstellung von Assoziationen in UML. Abbildung 3.18 zeigt die beiden im UML-Standard [41] beschriebenen Alternativen. In 3.18a ist die Assoziation durch eine einfache Linienverbindung dargestellt. Diese Variante eignet sich nur für binäre Assoziationen. In Abbildung 3.18b ist eine Darstellung mit einem Diamant-Symbol als Repräsentant der Assoziation zu sehen. Die teilnehmenden Klassen sind durch Linien mit dem Diamant-Symbol verbunden. Diese Darstellungsform eignet sich auch für Assoziationen mit drei oder mehr beteiligten Klassen.

Die editierbare Struktur zu Abbildung 3.18a besteht aus drei Objekten, nämlich zwei Klassenrepräsentanten und einem Objekt, das die Verbindungslinie repräsentiert. Demgegenüber hat die editierbare Struktur zu Abbildung 3.18b sieben Objekte. Neben den drei Klassenrepräsentanten gibt es ein Objekt für den Diamanten und drei weitere für die Linienverbindungen.

Diese Zerlegung des Diagramms in Objekte ist natürlich nicht die einzig sinnvolle. Sie hat aber den Vorteil, dass die visuelle Darstellung sehr einfach aus der editierbaren Struktur abgeleitet werden kann, da sich den vorkom-

```
CLASS Root {
  classes: SUB Class*;
  associations: SUB Assoc*;
}

CLASS Class {
  name: VAL String;
}

CLASS Assoc {
  name: VAL String;
  connections: SUB AssocEnd*;
}

CLASS AssocEnd {
  type: REF Class;
}
```

(a) Semantische Syntax

```
DERIVED ClassDiagram ROOT Root {

  ABSTRACT CLASS Assoc {
    name: VAL VLString;
  }

  CLASS MultAssoc INHERITS Assoc {
    position: VAL VLPoint;
    connections: SUB MultAssocEnd*;
  }

  CLASS BinAssoc INHERITS Assoc {
    from: SUB BinAssocEnd%;
    to: SUB BinAssocEnd%;
  }

  ABSTRACT CLASS AssocEnd {
    type: REF Class;
  }

  CLASS MultAssocEnd INHERITS AssocEnd {}
  CLASS BinAssocEnd INHERITS AssocEnd {}
}
```

(b) Spezifikation der abgeleiteten Struktur

Abbildung 3.19: Definition einer Editor-Syntax für Klassendiagramme mit alternativen Darstellungsformen für Assoziationen

menden Objektarten *Klasse*, *Assoziationslinie*, *Diamant* und *Diamant-Konnektor* einfache, einheitliche Darstellungen zuordnen lassen. Die zu speichernden Layoutdaten wie Positionen oder Zwischenpunkte lassen sich in natürlicher Weise den Objekten zuordnen. Außerdem entspricht diese Modellierung den vom Sprachbenutzer erwarteten Editiereigenschaften.

Abbildung 3.19 zeigt die Spezifikation eines entsprechenden semantischen und editierbaren Strukturmodells. Semantisch besitzen Assoziationen eine Liste von Endpunkten, die jeweils einen Klassen-Knoten referenzieren. Die editierbare Struktur der Darstellungsform 3.18b entspricht bis auf die Layoutattribute der semantischen Struktur. Die Struktur der Darstellungsform 3.18a weicht stärker ab. Statt der Liste von Assoziationsendpunkten gibt es dort die Attribute `from` und `to`. Dafür wird in diesem Fall kein Layout-Attribut benötigt.

Um die editierbare mit der semantischen Struktur zu koppeln, müssen Änderungsoperationen der editierbaren Struktur und Anpassungsoperationen überschrieben werden. Wird in der editierbaren Struktur ein Objekt der Klasse `BinAssoc` erzeugt, müssen in der semantischen Struktur ein Knoten der Klasse `Assoc` und zwei der Klasse `AssocEnd` angelegt werden. Wird in der editierbaren Struktur ein Objekte der Klasse `MultAssoc` oder

`MultAssocEnd` angelegt, entspricht dies dem Anlegen von `Assoc` bzw. `AssocEnd` in der semantischen Struktur.

Des Weiteren müssen einige Operationen zur Anpassung der editierbaren Struktur von Hand implementiert werden. Entsteht z.B. in der semantischen Struktur eine neue Assoziation, muss entschieden werden, auf welche Weise sie dargestellt werden soll. Es kann auch vorkommen, dass aus einer binären Assoziation eine ternäre wird. In diesem Fall muss die Anpassungsoperation den `BinAssoc`-Knoten der editierbaren Struktur durch einen Knoten der Klasse `MultAssoc` ersetzen.

Das Beispiel macht deutlich, dass die Kopplung von semantischer und visueller Strukturen keineswegs immer geradlinig ist. Die Abbildung der semantischen auf die editierbare Struktur kann wie hier relativ komplizierte strukturelle Entscheidungen erfordern. Der Algorithmus für solche Entscheidungen ist keineswegs eindeutig sondern ein Entwurfsaspekt der Editorimplementierung. Die Anforderungen von nicht-trivialen strukturellen Kopplungen dieser Art können nur schwer vorausgesehen werden. Das vorgestellte Kopplungsmodell bietet aber genügend Flexibilität für solche Fälle, ohne den Spezifikationsaufwand für einfach gelagerte Fälle zu vergrößern.

3.4.6 Kommunikationsmuster in Streets

Bis hierhin wurden schon mehrere Beispiele betrachtet, bei denen die editierbare Struktur durch Verwendung mehrerer Repräsentanten eines semantischen Objekts Redundanz enthielt. Die Redundanz kann sinnvoll sein um Zusammenhänge deutlicher zu machen oder das Layout zu verbessern. In den obigen Fällen lag die Einführung von Redundanz im Ermessen des Sprachanwenders. Nachfolgend wird ein Fall beschrieben, bei dem Redundanz ein essenzielles Ausdrucksmittel der grafischen Sprache ist.

Die visuelle Sprache *Streets* wurde bereits in Abschnitt 2.2.4 vorgestellt. Ein wichtiger Bestandteil von *Streets* ist eine Teilsprache zur Spezifikation so genannter *Topologien*. Topologien beschreiben die Kommunikationsstruktur zwischen einer Gruppe gleichartiger Prozesse. In *Streets* entscheidet sich der Sprachanwender für eine Grundstruktur - z.B. Baum-, Gitter oder Ring - und spezifiziert dann basierend auf dieser Grundstruktur die Verbindungen zwischen den Ein- und Ausgabeports der Prozesse. Abbildung 3.20 zeigt eine derartige Spezifikation. Als Grundstruktur wurde eine Baum gewählt. Die Knoten des Graphen repräsentieren die Prozesse, deren Kommunikationsstruk-

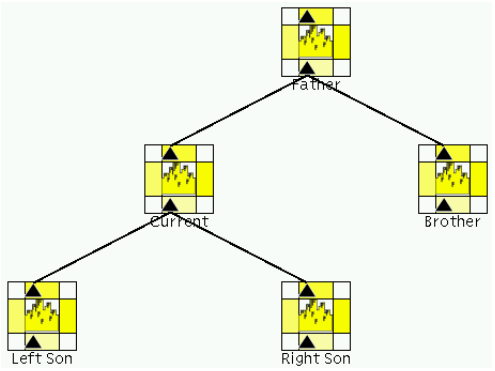


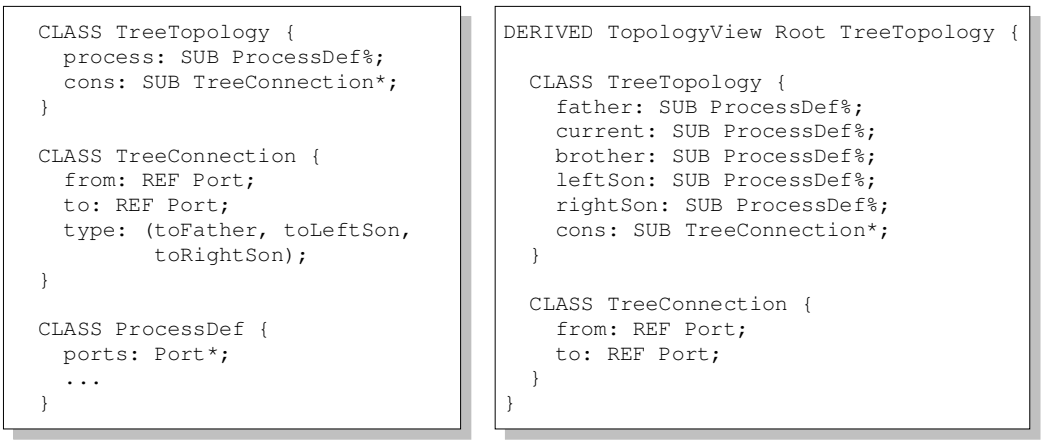
Abbildung 3.20: Topologie-Definition in *Streets*

tur spezifiziert wird. Da die Prozesse vom gleichen Typ sind gibt es nur eine Prozessdefinition, die in diesem Fall einen ausgehenden und einen eingehenden Port besitzt, wobei der eingehende Port mit den ausgehenden Ports der Unterknoten verbunden ist. In der Abbildung sind die Ports durch schwarze Dreiecke symbolisiert und werden im Folgenden A (ausgehender Port) und B (eingehender Port) genannt. Die abgebildete Topologiedefinition legt also fest, dass der Port A eines Prozesses mit dem Port B seines Vaters verbunden werden soll.

Die Spezifikation legt *nicht* fest, dass ein Prozesssystem aus fünf Prozessen besteht, sondern sie spezifiziert lediglich das Schema der Verbindungen. Die Anzahl der Prozesse ist im Allgemeinen erst zur Laufzeit bekannt. Die Zahl und Anordnung der Prozess-Repräsentanten in der visuellen Darstellung dient nur der intuitiven Darstellung. Sie ist für eine bestimmte Topologie fest vorgegeben und kann nicht vom Sprachbenutzer geändert werden. Dieser kann lediglich Verbindungslinien zwischen Ports konstruieren.

Alle Verbindungslinien repräsentieren die gleiche Information: Verbinde den Port A mit dem Port B des Vater-Prozesses. Löscht der Sprachanwender eine der Verbindungen, so werden damit automatisch alle Verbindungen gelöscht. Wird eine der Verbindungen neu erstellt, sind danach auch alle anderen Repräsentanten wieder vorhanden.

Abbildung 3.21 zeigt das semantische und editierbare Strukturmodell, wobei sich der Teil für die abgeleitete Struktur allerdings auf die benötigten Klassen für die Baum-Topologie beschränkt. Ähnliche Klassen würden auch für die anderen in *Streets* erlaubten Topologien benötigt. Im semantischen Modell ist erkennbar, dass Topologiespezifikationen zu einer Prozessdefinition gehören und im Wesentlichen aus einer Menge von Verbindungsdefinitionen



```
CLASS TreeTopology {
  process: SUB ProcessDef%;
  cons: SUB TreeConnection*;
}

CLASS TreeConnection {
  from: REF Port;
  to: REF Port;
  type: (toFather, toLeftSon,
         toRightSon);
}

CLASS ProcessDef {
  ports: Port*;
  ...
}
```

(a) Semantische Syntax

```
DERIVED TopologyView Root TreeTopology {

  CLASS TreeTopology {
    father: SUB ProcessDef%;
    current: SUB ProcessDef%;
    brother: SUB ProcessDef%;
    leftSon: SUB ProcessDef%;
    rightSon: SUB ProcessDef%;
    cons: SUB TreeConnection*;
  }

  CLASS TreeConnection {
    from: REF Port;
    to: REF Port;
  }
}
```

(b) Spezifikation der abgeleiteten Struktur

Abbildung 3.21: Definition einer Editor-Syntax für Baum-Topologien in *Streets*

bestehen. Im Fall der Baum-Struktur gibt es drei Typen von Verbindungen, nämlich `toFather`, `toLeftSon` und `toRightSon`. Jede Verbindungsspezifikation bezieht sich auf bestimmte Ports der Prozessdefinition.

Die editierbare Struktur unterscheidet sich stark von der semantischen. In der editierbaren Struktur gibt es fünf Repräsentanten der Prozesse, die verschiedene Rollen spielen. `Connection`-Objekten sind in der editierbaren Struktur keine Verbindungstypen zugeordnet. Da sie Port-Repräsentanten verbinden und da ein Port-Repräsentant zu einem bestimmten Prozess-Repräsentanten gehört, ergibt sich der Typ einer Verbindung implizit aus der Rolle des Prozess-Repräsentanten.

Die Synchronisation beider Strukturen muss weitgehend manuell implementiert werden. Wird ein `TreeTopology`-Knoten in der editierbaren Struktur erzeugt, muss sein Gegenstück in der semantischen Struktur erzeugt werden. Zusätzlich muss die RR-Relation der fünf Prozess-Repräsentanten mit dem Prozessdefinitions-Knoten der semantischen Struktur hergestellt werden. Die weitere Kopplung der Repräsentanten erfolgt dann automatisch, d.h. es wird z.B. zu jedem Port der Prozessdefinition genau ein Port-Repräsentant im Kontext jedes Prozess-Repräsentanten erstellt. Anhand der Repräsentanten lassen sich z.B. Ports löschen oder neue Ports einfügen.

Manuell zu implementieren ist des Weiteren der Umgang mit den Verbindungsrepräsentanten. Wird die semantische Struktur geändert, so müssen die Verbindungsrepräsentanten der editierbaren Struktur neu berechnet werden. In Rückrichtung muss die Änderungsoperation zur Erstellung neuer Verbin-

```
RULE BinOpr: Expr ::= Expr Opr Expr END;
```

(a) Kontextfreie Grammatik

```
RULE BinOpr: Expr ::= Expr Opr Expr
COMPUTE
  Expr[1].code = PTGBinOpr(Expr[2].code, Opr.code, Expr[3].code);
END;
```

(b) Attributberechnungen

Abbildung 3.22: Beispiel für gekoppelte semantische Strukturen

dungen überschrieben werden. Wird eine neue Verbindung erstellt, so muss aus den Rollen des Start- und End-Ports der Verbindungstyp ermittelt und ein entsprechendes `TreeConnection`-Objekt in der semantischen Struktur angelegt werden. Die Löschoperation braucht nicht überschrieben zu werden, denn das Löschen wird schon durch die Standardimplementierung in gewünschter Weise auf die semantische Struktur abgebildet.

3.4.7 Zuordnung von Attributberechnungen zu Produktionen

Bis hierher wurden Anwendungen des Kopplungskonzepts vorgestellt, die Teile der editierbaren und semantischen Struktur koppeln. Charakteristisch dafür ist, dass die abgeleitete Struktur keine zusätzlichen semantischen Informationen enthält. Das Kopplungskonzept lässt sich aber auch zur Kopplung von Teilstrukturen verwenden, bei denen die abgeleitete Struktur durchaus semantisch relevant ist. Das ist immer dann nützlich, wenn bestimmte Grundstrukturen als Rahmen für hierauf aufbauende Spezifikationen verwendet werden sollen.

Ein Beispiel für diese Situation ist in Abbildung 3.22 zu sehen. Abbildung 3.22a zeigt eine kontextfreie Grammatik und Abbildung 3.22b Attributberechnungen, die sich auf diese Grammatik beziehen. Die hier verwendete Notation stammt aus dem Eli-System [31]. Strukturell betrachtet werden die Grammatik-Regeln in Abbildung 3.22a definiert und in Abbildung 3.22b durch Attributberechnungen ergänzt. Eli erlaubt durchaus mehrere Teilspezifikationen wie in Abbildung 3.22b, die sich auf die gleiche Regel beziehen. Hierdurch lassen sich verschiedene Spezifikationsaspekte wie Namensanalyse, Typanalyse oder Codegenerierung separieren.

Prinzipiell ist die erste Zeile in Abbildung 3.22b redundant und damit entbehrlich. Sie verbessert aber die Lesbarkeit der Spezifikation, denn sie zeigt den Kontext der Attributberechnungen. In visuellen Spezifikationen ist eine explizite Repräsentation des Kontexts sogar manchmal unverzichtbar. Sollen die Grammatik-Symbole in den Attributberechnungen nicht textuell sondern z.B. mit Linienverbindungen spezifiziert werden, werden Stellvertreter-Objekte für die Grammatik-Produktion benötigt, an denen die Linien enden können.

Durch das oben vorgestellte Kopplungskonzept lässt sich die Konsistenz zwischen allen Auftreten einer Produktion automatisch sicherstellen. Die eigentliche Definition der Produktion spielt dabei die Rolle der Basis-Struktur und alle Auftreten der Produktion im Stil von 3.22b sind im Sinne der Kopplung abgeleitete Strukturen. Die Syntax der abgeleiteten Struktur muss bei diesem Anwendungsbeispiel um Attribute erweitert werden, die die Attributberechnungen speichern.

In diesem einfachen Beispiel könnte man die Kopplung der Strukturen auch „von Hand“ implementieren. Je komplexer die Strukturen sind, die als Rahmen für eine Spezifikation dienen, desto mehr lohnt sich aber der Einsatz des Kopplungskonzepts, denn der Kopplungsaufwand hängt dann nicht mehr von der Komplexität der Basisstruktur ab.

3.5 Schlussbemerkungen zum Kopplungsmodell

Nachfolgend möchte ich das Einsatzspektrum und die Grenzen des Kopplungsmodells abschließend einordnen und auf sinnvolle Erweiterungsmöglichkeiten eingehen.

3.5.1 Einsatzspektrum und Erweiterungen

Das Kopplungskonzept wurde mit dem Ziel entworfen, den Spezifikationsaufwand zur Kopplung ähnlicher Strukturen möglichst klein zu halten. Besonders zur Kopplung von editierbarer und semantischer Struktur ist dies sinnvoll, da sie vielen Fällen sehr ähnlich sind. Falls die Unterscheidung zwischen editierbarer und semantischer Struktur für eine Sprachimplementierung nicht notwendig ist, braucht dieser Mechanismus überhaupt nicht zur

Kenntnis genommen werden. Stellt sich nachträglich heraus, dass doch zwischen editierbarer und semantischer Struktur unterschieden werden muss, erlaubt das Spezifikationskonzept einen fließenden Übergang zur Trennung beider Strukturen.

Wie durch die Mehrzahl der oben diskutierten Beispiele gezeigt werden konnte, lassen sich viele in realistischen visuellen Sprachen vorkommende Phänomene auf sehr einfache Weise umsetzen. Häufig brauchen lediglich zusätzliche Attribute eingeführt oder existierende teilweise entkoppelt werden, so dass sehr wenig Zusatzaufwand entsteht.

Die Beispiele aus den Abschnitten 3.4.5 (Assoziationen) und 3.4.6 (Topologie-Definitionen in *Streets*) zeigen, dass die Kopplung nicht immer geradlinig ist. In diesen Fällen müssen bei der Aktualisierung relativ komplizierte strukturelle Entscheidungen getroffen werden. Es ist klar, dass solche Implementierungen recht aufwändig sein können. Würde die editierbare und semantische Struktur aber nicht unterschieden, würde sich dieser Aufwand lediglich auf die Implementierung der grafischen Darstellung verlagern. In diesem Fall würden die Teilprobleme vermischt und wären noch schwerer lösbar. Visuelle Muster ließen sich zudem nur noch schwer anwenden. Durch die Trennung lassen sich die strukturellen Eigenarten der Sprache vollständig von Details des Layouts oder konkreten Interaktionsmechanismen abkoppeln. Die eigentliche visuelle Darstellung lässt sich so vollständig auf vordefinierte visuelle Muster zurückführen.

Zur Umsetzung spezieller Anpassungsoperationen muss der Sprachentwickler handimplementierten Code hinzufügen. Dies ist der flexibelste Ansatz, denn auf diese Weise hat der Sprachentwickler alle Freiheiten. Es ist aber vorstellbar, hierfür Spezifikationsmittel höheren Niveaus bereitzustellen. Benutzerdefinierte Anpassungsfunktionen ließen sich z.B. evtl. durch Graph-Transformationsregeln spezifizieren. Um mit anderen Nutzern über bestimmte Kopplungsaufgaben zu diskutieren, wurden teilweise bereits entsprechende Diagramme gezeichnet. Die derzeitige Umsetzung betrachte ich lediglich als Prototyp, mit dem bereits positive Erfahrungen gesammelt werden konnten.

3.5.2 Grenzen

Das Kopplungsmodell wurde entworfen, um kleine Unterschiede zwischen semantischer und editierbarer Struktur zu überbrücken. Wie oben beschrie-

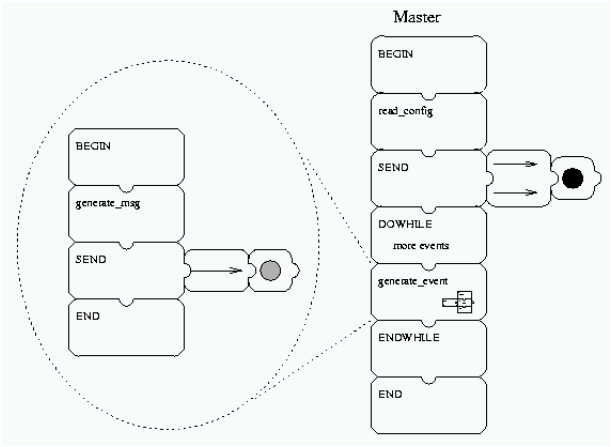


Abbildung 3.23: Die Blox-Methodik anhand des Systems *VPEcons*

ben, können auch größere Unterschiede durch handimplementierte Funktionen überbrückt werden, solange sich jede Änderung der abgeleiteten Struktur direkt auf die semantische Struktur übertragen lässt. Ungeeignet ist das Kopplungsmodell dagegen, wenn das Niveau der editierbaren Struktur wesentlich niedriger als das der semantischen ist. In diesem Fall lassen sich Änderungen an der editierbaren Struktur nicht mehr direkt auf die semantische übertragen.

Ein Beispiel hierfür sind Sprachen, die auf der *Blox* Methodik basieren. In dieser von Glinert [17] eingeführten Sprachklasse werden die Grundelemente einer Sprache durch Puzzle-Teile symbolisiert. Ein Programm wird konstruiert, indem die Puzzle-Teile zu größeren Einheiten zusammengesetzt werden. Abbildung 3.23 zeigt ein visuelles Programm des VPEcons-Systems [62], das die Blox-Methodik verwendet, um die Implementierung paralleler Programme zu vereinfachen.

Da entsprechend der beabsichtigten Editiereigenschaften ein Puzzle-Teil als elementares Sprachobjekt zu betrachten ist, hat die editierbare Struktur ein sehr niedriges Niveau. Abbildung 3.23 enthält z.B. einen *do while* und einen *end while* Block, die – wie aus anderen imperativen Programmiersprachen bekannt – immer paarweise auftreten müssen. Durch das niedrige Niveau der editierbaren Struktur kann diese Eigenschaft weder erzwungen noch leicht erkannt werden.

Aufgrund dieser Eigenschaft ist die editierbare Struktur nicht sehr gut zur Weiterverarbeitung geeignet. Es bietet sich an, durch Grammatiken und Parsing-Techniken eine Struktur höheren Niveaus abzuleiten, die als seman-

tische Struktur dient. Das Kopplungsmodell von DEViL lässt sich allerdings nicht gut hierfür verwenden, denn es setzt voraus, dass jede gültige editierbare Struktur auf eine semantische Struktur abbildbar ist.

Dies kann als charakteristische Eigenschaft von Strukturediteren an sich betrachtet werden. Wenn Zwischenzustände der editierbaren Struktur erlaubt wären, die keine Entsprechung in der semantischen Struktur hätten, wäre es problematisch, das Editiermodell von Strukturediteren aufrechtzuerhalten. Wenn während eines inkonsistenten Zwischenzustands die semantische Struktur anhand einer zweiten Sicht geändert würde, wäre nicht klar, wie die editierbare Struktur der ersten Sicht anzupassen wäre. Strukturediteren für textuelle Sprachen, die zusätzlich freie Texteingabe unterstützen, lösen dieses Problem teilweise dadurch, dass der entsprechende Editiermodus nur verlassen werden darf, wenn ein gültiger Programmzustand erreicht wurde.

Eine semantische Struktur hohen Niveaus und eine editierbare Struktur niedrigen Niveaus entspricht prinzipiell dem Ansatz des so genannten freien Editierens visueller Programme. Einen Kopplungsmechanismus zwischen Strukturen solch unterschiedlichen Niveaus zu entwickeln hieße ein System wie DiaGen II (siehe Abschnitt 2.5.6) zu implementieren. Prinzipiell wäre eine Erweiterung von DEViL in diese Richtung nicht ausgeschlossen. Für die in dieser Arbeit betrachtete Zielsetzung ist so eine Erweiterung aber entbehrlich.

3.6 Verwandte Arbeiten

Im Grundlagen-Kapitel wurden bereits einige Systeme vorgestellt, die teilweise mit den hier vorgestellten Konzepten verwandt sind. Nachfolgend wird der Bezug zu diesen Ansätzen hergestellt.

Syntax-Definition Zur Definition der abstrakten Syntax von Sprachen gibt es eine Vielzahl unterschiedlicher Kalküle. Schon früh wurden Baum-Grammatiken verwendet, um die abstrakte Syntax textueller Sprachen zu beschreiben. Um Editieroperationen auf höherem Niveau zu erlauben, wurden Baum-Grammatiken z.B. in VL-Eli (Abschnitt 2.4) oder PSG (Abschnitt 2.5.1) um zusätzliche Konstrukte zur Listen- und Klassenbildung erweitert.

Auch das Konzept zur Modellierung von Strukturen in UML (siehe Abschnitt 2.2.1) ist mit Baum-Grammatiken verwandt. Hier gibt es außer Teil-Ganzes-Beziehungen jedoch auch andere Beziehungen, die in UML Assozia-

tionen genannt werden. Weitere verwandte Sprachen zur Strukturbeschreibung sind XML-DTDs und XML-Schema-Definitionen. In all diesen Sprachen spielt die Baumstruktur eine wichtige Rolle. Bei Baum-Grammatiken und XML-basierten Definitionssprachen modelliert die Syntax prinzipiell eine Baumstruktur (evtl. mit Querkanten), wohingegen Instanzen von UML-Modellen im Allgemeinen keine zusammenhängenden Bäume sind.

Beim Entwurf der oben beschriebenen Sprache DSSL habe ich versucht, eine möglichst schlanke Sprache zu entwerfen, die das Beste aus all diesen Kalkülen vereint. Bereits in Abschnitt 3.1.2 habe ich sie mit einigen der oben genannten Sprachen verglichen. Grob zusammengefasst gibt es in DSSL gerade genug Modellierungskonzepte, um Strukturen komfortabel und bedarfsgerecht spezifizieren zu können. Die Sprache ist andererseits schlank genug, um die Anzahl der Design-Alternativen zu beschränken und so einen einfachen Umgang mit DSSL-Strukturen zu ermöglichen.

DSSL ist auch eng mit Strukturmodellierungskonzepten in Meta-CASE Werkzeugen verwandt. Häufig basieren diese auf Modellierungskalkülen, die UML-Klassendiagrammen ähneln und besitzen damit eine vergleichbare Ausdrucksstärke. Die automatische Generierung eines Editors aus der nackten Syntax und die Einschränkbarkeit der Strukturen durch Konsistenzbedingungen ist auch in solchen Systemen zu finden. Tendenziell sind die Modellierungskalküle in Meta-CASE Werkzeugen aber stärker spezialisiert und daher nicht so flexibel einsetzbar. Ein gutes Beispiel dafür ist MetaEdit+, das auf dem GOPRR-Modell basiert (siehe Abschnitt 2.5.4). Die Spezialisierung hat den Vorteil, dass sich aus der nackten Syntax bereits Editoren generieren lassen, die typischen Softwaremodellierungssprachen wie UML bereits sehr nahe kommen. Der Nachteil ist jedoch, dass sich anders strukturierte Sprachen mit diesem Ansatz nicht gut umsetzen lassen. Selbst wenn das verwendete Syntax-Kalkül allgemeiner ist, schränken Meta-CASE Werkzeuge häufig die umsetzbaren grafischen Repräsentationen zugunsten einer einfacheren Spezifizierbarkeit ein.

Auch das *Eclipse Modeling Framework* (EMF) [8] hat auf der Struktur-Modellierungsebene Ähnlichkeiten mit DEViL. Auch EMF besitzt eine spezielle Spezifikationssprache für Strukturen und kann daraus automatisch Standard-Editoren erzeugen. Allerdings handelt es sich bei EMF nicht um einen vollständig generierenden Ansatz, da Konsistenzprüfungen, die Weiterverarbeitung sowie die grafische Darstellung von Hand implementiert werden müssen. Die Umsetzung der grafischen Darstellung wird allerdings durch das zugehörige *Graphical Editing Framework* vereinfacht.

Die bis jetzt beschriebenen Modellierungskalküle gehören zu einer Klasse, die manchmal als *modellbasiert* bezeichnet wird. Das wesentliche Merkmal modellbasierter Kalküle ist, dass die Gültigkeit von Modell-Instanzen durch lokale Konsistenzprüfungen verifizierbar ist. Prinzipiell muss für jeden Knoten der Modell-Instanz geprüft werden, ob dessen Eigenschaften und Relationen mit den Attributen und Assoziationen der entsprechenden Klasse im Modell vereinbar sind. Demgegenüber gibt es eine andere Klasse von Kalkülen zur Syntax-Definition, die hier als *ableitungsbasiert* bezeichnet werden soll. Hier ist die Gültigkeit einer Syntax-Instanz dadurch definiert, indem sie durch iterative Anwendung von Transformationsregeln aus einem Startsymbol abgeleitet werden kann. Diese Art der Syntax-Definition wird z.B. im SRG-ASG-Ansatz (siehe Abschnitt 2.5.5) und in DiaGen II (siehe Abschnitt 2.5.6) verwendet. In Strukturediteren werden die Transformationsregeln der Syntax als Editieroperationen verwendet. Auf diese Weise ist sichergestellt, dass nur syntaktisch korrekte Programme konstruiert werden können.

Im Gegensatz zur modellbasierten Syntax-Spezifikation kann die Syntax in ableitungsbasierten Kalkülen wesentlich schärfer formuliert werden. Zusätzlich eignet sich der Prozess der Gültigkeitsprüfung – das so genannte *Parsieren* – auch dazu, die Struktur auf ein höheres Niveau zu transformieren. Dieser Mechanismus wird auch bei der Verarbeitung textueller Sprachen eingesetzt: Eine konkrete Grammatik (ein ableitungsbasiertes Kalkül) beschreibt die Syntax von Symbolfolgen. Während des Parsierens einer Symbolfolge lässt sich eine abstrakte Struktur höheren Niveaus aufbauen, deren Syntax auf einer Baum-Grammatik (einem modellbasierten Kalkül) basiert.

Editieroperationen und Cut-and-Paste Die oben beschriebene Methode, bereits aus der „nackten“ Syntax benutzbare Editoren zu generieren, erinnert an Generatoren für textuelle Strukturediteren. Auch z.B. PSG (siehe Abschnitt 2.5.1) kommt mit sehr wenig Zusatzinformation aus. Das VL-Eli System (siehe Abschnitt 2.4) erlaubt auch die Generierung von Editoren aus der nackten Syntax, wobei in der entsprechenden Sicht allerdings keine Querrelationen editiert werden können. Die in DSSL enthaltenen spezifischeren Konstrukte für Querrelationen sind hierfür eine wichtige Voraussetzung.

Vor allem im Kontext von Systemen, die auf einem ableitungsbasierten Kalkül basieren, wird häufig die Benutzerfreundlichkeit von Editieroperationen diskutiert. Dies liegt daran, dass sich die Editieroperationen dort aus den komplexen Transformationsregeln der Syntax ergeben. Häufig wird das Problem

gelöst, indem zugunsten einfacherer Transformationsregeln auf die syntaktische Korrektheit von Diagrammen verzichtet wird (siehe Abschnitt 2.5.5).

Pfadausdrücke Die oben eingeführte Notation, um strukturelle Zusammenhänge in DSSL-Strukturen auszudrücken, findet sich in ähnlicher Form auch in anderen Systemen. In DiaGen II (siehe Abschnitt 2.5.6) werden vergleichbare Pfadausdrücke benutzt, um Anwendbarkeitsbedingungen von Produktionen zu beschreiben und um zwischen den korrespondierenden Modellen zu navigieren. Dort ist ein Pfadausdruck eine Folge von Hyperkanten. Um die Durchlaufrichtung der Hyperkanten zu beschreiben, wird jeweils ein Eintritts- und ein Austrittstentakel angegeben. Auch in DiaGen II dürfen Pfadausdrücke die aus regulären Ausdrücken bekannten Konstrukte wie Wiederholungen enthalten. Die bekannteste Sprache für Pfadausdrücke ist XPath [68]. Sie bezieht sich auf XML-Datenstrukturen und besitzt wesentlich mehr Sprachkonstrukte.

Strukturelle Kopplung Das Konzept der strukturellen Kopplung wird im Kontext von Sprachimplementierungen häufig verwendet. Im SRG-ASG-Ansatz (siehe Abschnitt 2.5.5) werden zwei Strukturen, der *spatial relations graph* und der *abstract structure graph* durch Graphgrammatiken miteinander gekoppelt. Beim Ansatz von Akehurst [3] werden zur Kopplung von Strukturen OCL-Constraints verwendet. Wenn die so formulierten Bedingungen verletzt sind, müssen die Strukturen in diesem Ansatz durch handgeschriebene Funktionen repariert werden. In beiden Ansätzen ist die Kopplung symmetrisch. Wie oben beschrieben ist diese Art der Kopplung für die hier gezeigte Anwendung ungeeignet, weil hier eine der beiden Strukturen evtl. Redundanz enthält. Die „Hin-Richtung“ der Kopplungsmethode in DEViL ist allerdings eng mit dem Konzept von Akehurst verwandt. Die Constraints werden in DEViL allerdings nicht durch OCL-Constraints, sondern durch prädikatenlogische Formeln bzw. Funktionen und Pfadausdrücke spezifiziert. Im Ansatz von Akehurst sind die Strukturen so unterschiedlich, dass es nicht sinnvoll ist, Mechanismen zur automatischen Kopplung ähnlicher Strukturen anzubieten.

Die „Rück-Richtung“ ist eng mit dem Konzept verwandt, nach dem in VL-Eli und DEViL die Vektorgrafik mit der editierbaren Struktur gekoppelt ist. Wird eine Änderung an der Vektorgrafik vorgenommen, werden die Änderungsoperationen an die editierbare Struktur „weitergeleitet“. Dazu ist jedem Vektorgrafik-Primitiv die Information zugeordnet, welchen Knoten der edi-

tierbaren Struktur es repräsentiert. Ähnliche Ansätze werden auch in anderen funktional implementierten Editoren wie z.B. in *Proxima* [56] verwendet.

Neue Beiträge Der wichtigste Beitrag dieses Abschnitts ist die Methode zur Kopplung von Strukturen. Eine Besonderheit ist, dass die strukturelle Gleichheit als Normalfall und die strukturelle Abweichung als Ausnahme betrachtet wird. Hierauf basiert sowohl die Definition der abgeleiteten Syntax als auch die Kopplung der Strukturen. Ein wichtiges Ergebnis ist, dass in beiden Fällen die Kopplung feingranular abgeschwächt oder überschrieben werden kann.

Ein weiterer Beitrag ist die Zusammenstellung praktisch relevanter Kopplungsphänomene in visuellen Sprachen, die auch zur Evaluation anderer Ansätze dienen können.

4 Spezifikation visueller Sichten

Inhalt

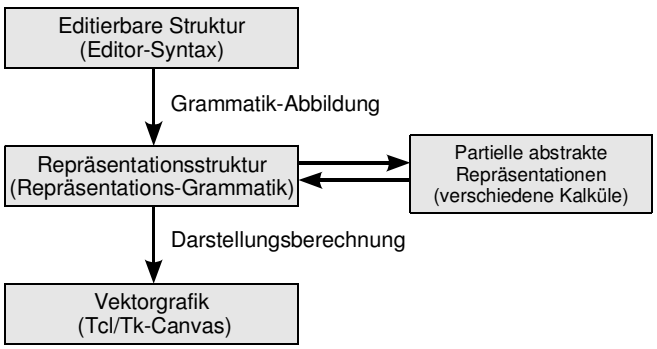


Abbildung 4.1: Modell zur Spezifikation visueller Sichten in DEViL

Im dritten Kapitel ging es um das Verhältnis zwischen editierbarer und semantischer Struktur. Dort wurde deutlich, dass diese Unterscheidung vor allem dazu dient, mit der editierbaren Struktur eine maßgeschneiderte Grundlage zur Implementierung visueller Sichten bereitzustellen. In diesem Kapitel wird beschrieben, wie auf Basis der Editor-Syntax eine visuelle Sicht spezifiziert werden kann. Hauptsächlich muss dazu „nur noch“ eine Funktion definiert werden, die aus der editierbaren Struktur eine konkrete Repräsentation berechnet.

Zur Lösung dieser Aufgabe wurde die recht universelle Transformationskette vorgestellt in Abbildung 2.8 auf Seite 38 vorgestellt. Abbildung 4.1 zeigt, wie dieses Schema in DEViL umgesetzt wurde. Die Kästchen enthalten jeweils die Bezeichnung der entsprechenden Struktur auf Instanzebene und in Klammern dahinter die Bezeichnung der entsprechenden Strukturdefinition.

Die Syntax für die editierbare Struktur heißt abkürzend Editor-Syntax. Hierfür wird die in Kapitel 3 eingeführte Sprache DSSL benutzt. Eine attributierte Grammatik, die so genannte Repräsentations-Grammatik, bildet die Grundlage der Repräsentationsstruktur. Diese berechnet keine abstrakte Repräsentation wie in Abbildung 2.8, sondern direkt die visuelle Darstellung als Vektorgrafik. Stattdessen werden während der Berechnung partielle abstrakte Repräsentationsbeschreibungen generiert, die dann an spezialisierte Systeme zur Lösung bestimmter Layoutaufgaben weitergeleitet werden. Konkret wurden in DEViL ein spezialisierter Algorithmus zum Lösen von Nicht-Überlappungs-Constraints sowie ein Graph-Layout-System verwendet. Das Graph-Layout-System wird allerdings momentan nur zum Layout von Bäumen verwendet.

Der Grund für die Abweichung vom Transformationsschema aus Abbildung 2.8 ist, dass es derzeit kein universelles Kalkül auf der Ebene abstrakter Reprä-

sentationen gibt, das alle Layoutaufgaben angemessen und effizient löst. Die beste Näherung stellen Constraintsolver dar, mit denen jedoch Linienrouting, Graphlayout und Umbruchentscheidungen nur schwer zu realisieren sind.

Das Spezifikationskonzept basiert wie bereits erwähnt auf attributierten Grammatiken. Abschnitt eins dieses Kapitels begründet die Wahl dieser Methode und beschreibt, wie die editierbare Struktur auf eine angemessene Grammatik abgebildet wird. Ferner wird umrissen, wie durch Attributberechnungen grafische Darstellungen spezifiziert werden können.

Eine Besonderheit dieser Arbeit ist das Konzept der visuellen Muster. In Abschnitt zwei wird diese Methode weiter entwickelt und gezeigt, wie Darstellungskonzepte basierend auf visuellen Mustern wiederverwendbar gekapselt werden können. Insbesondere wird hier auf die Kombinierbarkeit visueller Muster-Varianten und die dazu benötigten musterübergreifenden Layoutstrategien eingegangen.

Zur Anwendung von Muster-Varianten ist es manchmal sinnvoll, die Repräsentations-Grammatik den jeweiligen Erfordernissen anpassen zu können. In Abschnitt drei wird eine Spezifikationsmethode beschrieben, die dies ermöglicht. Ferner wird deren Einsatzspektrum anhand von Beispielen demonstriert.

Für bestimmte Repräsentationsformen lässt sich die Spezifikation noch komfortabler gestalten. Hierzu werden in den Abschnitten vier und fünf zwei Spezialsprachen für unterschiedliche Einsatzgebiete vorgestellt. Durch die so genannten Generischen Zeichnungen können Darstellungsdetails bestimmter Sprachkonstrukte visuell spezifiziert werden. Die Sprache SLTR (*specification language for textual representations*) macht es besonders einfach, textuelle Teilrepräsentationen zu spezifizieren.

Abschnitt sechs führt schließlich eine zweite Art von Sichten, die so genannten Dialogsichten ein. Hierunter sind Sichten zu verstehen, die bestimmte aus grafischen Benutzungsschnittstellen bekannte Elemente wie Eingabefelder, Ankreuzfelder oder Auswahllisten enthalten. Es wird gezeigt, dass auch solche Sichten für Strukturediteren wichtig sind und diese mit den gleichen Mitteln wie visuelle Sichten spezifiziert werden können.

4.1 Attributberechnungen zur Spezifikation visueller Sichten

4.1.1 Wahl von attributierten Grammatiken

Zur Berechnung der grafischen Darstellung muss die editierbare Struktur auf eine Vektorgrafik abgebildet werden. Zur Implementierung entsprechender Abbildungsvorschriften gibt es eine Vielzahl von Techniken, u.a. attributierte Grammatiken, Attributberechnungen auf Graphen, gekoppelte Graphgrammatiken, funktionale Programmierung sowie Struktur-Transformationssprachen wie XQuery oder XSLT. Attributberechnungen haben hier den Vorteil, dass sie deklarativ sind und man sie gut modularisieren kann [33]. Bestimmte Berechnungsaspekte können so in einer Bibliothek gekapselt werden, während andere individuell hinzugefügt werden können. Die Modularisierbarkeit ist unerlässlich, wenn höhere Darstellungskonzepte wie visuelle Muster wiederverwendbar gekapselt werden sollen.

Während zur Übersetzung textueller Sprachen vorwiegend attributierte Grammatiken verwendet werden, werden im Kontext visueller Sprachen gerne Attributauswerter auf Graphen eingesetzt. Das liegt daran, dass bei visuellen Sprachen Querbeziehungen häufig strukturell repräsentiert werden, während sich diese bei der Verarbeitung von textuellen Sprachen oft erst durch semantische Analyse der Bezeichner ergeben. Graph-Attributauswerter haben den Vorteil, dass Attribute nicht nur über Baumkanten, sondern auch über Querbeziehungen hinweg zugegriffen werden können. Ein Nachteil ist jedoch, dass die Implementierungen solcher Attributauswerter prinzipbedingt ineffizienter als ihre baumbasierten Gegenstücke sind. Ferner können bei Graph-Attributierungen bestimmte Konsistenzbedingungen wie z.B. Zyklenfreiheit nicht statisch geprüft werden.

In attributierten Grammatiken machen zwar Querbeziehungen vergleichsweise mehr Mühe, sie sind aber dennoch eine sinnvolle Wahl zur Berechnung visueller Repräsentationen. Das liegt daran, dass auch visuelle Repräsentationen zu einem Großteil baumstrukturiert sind und der baumstrukturierte Anteil wesentlich das Layout bestimmt. Beispiele für baumstrukturierte grafische Relationen sind z.B. Schachtelungen, Beschriftungen und lineare Sequenzen von Elementen. Das Layout lässt sich in diesen Fällen sehr gut durch Attributberechnungen in Bäumen spezifizieren. Für Layoutberechnungen, in denen Querrelationen berücksichtigt werden müssen, werden häufig kom-

plexere Techniken wie Constraintsolver oder Graph-Layoutsysteme benötigt. Der Zusatzaufwand für den Umgang mit Querbeziehungen ist in diesem Fall vergleichsweise klein.

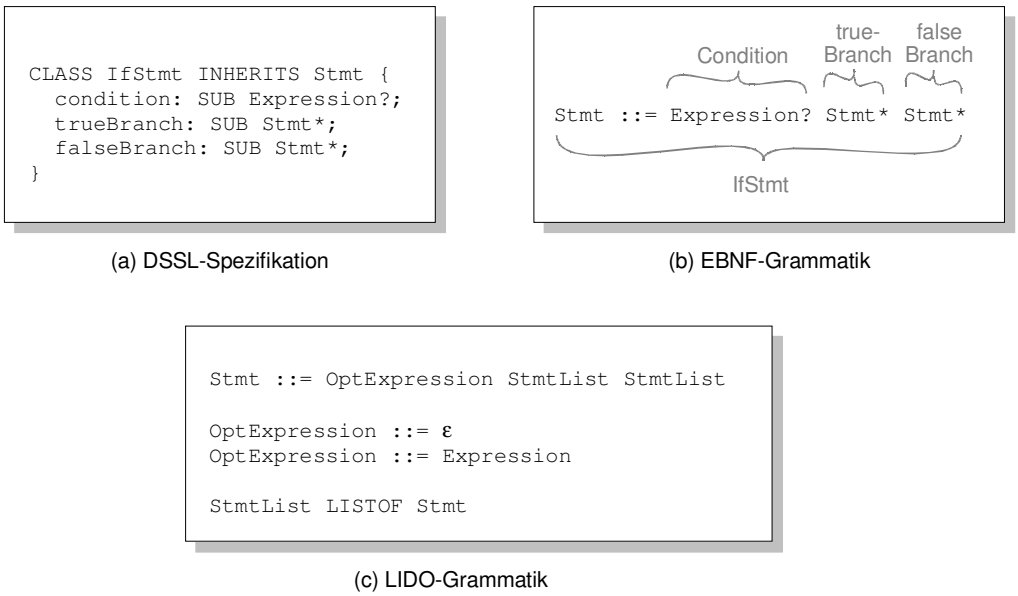
In meinem Ansatz hätten sowohl baum- als auch graphbasierte Attributauswerter benutzt werden können. Eine wichtige technische Forderung war allerdings, dass der zu verwendende Attributauswerter-Generator Abstraktionsmechanismen bereitstellt, um Berechnungen wiederverwendbar zu kapseln. Die Spezifikationssprache LIDO besitzt solche Abstraktionsmechanismen und es wurden bei der Entwicklung des VL-Eli Systems bereits Erfahrungen damit gesammelt. Daher war es konsequent, auch in DEViL die Spezifikationssprache LIDO zu verwenden. Die Vor- und Nachteile der verschiedenen Varianten ließen sich natürlich im Detail diskutieren. Hierzu möchte ich aber auf die Arbeit von Jung [28, S. 94 ff.] verweisen.

4.1.2 Abbildung der editierbaren Struktur auf die Repräsentations-Struktur

Um attributierte Grammatiken als Kalkül für die Berechnung visueller Repräsentationen verwenden zu können, muss die Editor-Syntax auf eine kontextfreie Grammatik abgebildet werden. Die in Kapitel 3 eingeführte Sprache DSSL wurde so entworfen, dass dieser Abbildungsschritt relativ einfach ist. Abbildung 4.2a zeigt eine DSSL-Klasse und Abbildung 4.2b eine dazu äquivalente Regel in EBNF-Form. Die Klasse `IfStmt` ist eine Unterklasse von `Stmt` und besitzt die Attribute `condition`, `trueBranch` und `falseBranch`. Das Attribut `condition` ist optional[1], `trueBranch` und `falseBranch` sind jeweils Listen. Aus Abbildung 4.2b ist ersichtlich, dass die abstrakte Oberklasse die linke Seite der Produktion bestimmt, während die Attribute der Klasse die rechte Seite der Produktion bestimmen. Die Kardinalitäten der Attribute lassen sich direkt durch EBNF-Operatoren ausdrücken. Man beachte aber, dass bei der Abbildung die Namen einiger Rollen verloren gehen. Dies ist zum einen die Rolle der Produktion, zum anderen sind dies die Rollen der Symbole. Zur Illustration sind diese Rollen der Abbildung 4.2b hinzugefügt.

Die Lösung in Abbildung 4.2b ist noch nicht als Grundlage für attributierte Grammatiken geeignet, da dafür üblicherweise eine strikte BNF-Notation erforderlich ist. Als einzige Ausnahme enthält LIDO ein Spezialkonstrukt zur

[1] In diesem Beispiel sind Bedingungen optional, um bestimmte Phänomene bei der Abbildung auf Grammatiken zu verdeutlichen.



(a) DSSL-Spezifikation

(b) EBNF-Grammatik

(c) LIDO-Grammatik

Abbildung 4.2: Einfache Abbildung auf kontextfreie Grammatiken

Listenbildung. Abbildung 4.2c zeigt eine transformierte attributierte Grammatik, die für LIDO geeignet ist.

Unterscheidung von Rollen und Kontexten Die einfache Umsetzung aus Abbildung 4.2c ist aus verschiedenen Gründen nicht direkt geeignet, um sie als Grundlage zur Darstellungsberechnung zu verwenden. Zunächst stört, dass die Rollen der `StmtList`-Symbole nicht mehr zu erkennen sind. Welches Symbol steht für den `trueBranch` und welches für den `falseBranch`? Außerdem kommt es häufig vor, dass der Produktion „`StmtList LISTOF Stmt`“ je nachdem, ob `StmtList` einen `trueBranch` oder `falseBranch` repräsentiert, verschiedene Berechnungen zugeordnet werden sollen. Die gezeigte Modellierung erlaubt im Gegensatz dazu aber nicht einmal die Feststellung, ob `StmtList` überhaupt zu einem `IfStmt` gehört.

All diese Probleme rühren daher, dass die Rollen der Grammatiksymbole nicht unterschieden werden können. Abbildung 4.3a zeigt, was in diesem Fall unter der Rolle eines Symbols zu verstehen ist. Die Rollen sind dort in eckigen Klammern hinter dem Symbolnamen notiert.

Ideal wäre es, wenn die in 4.3a angegebenen Regeln direkt attributiert werden könnten. Da LIDO jedoch eine derartige Notation nicht vorsieht, müssen die Rollen explizit in Symbolnamen kodiert werden. Prinzipiell muss dazu ledig-

```
Stmt[IfStmt] ::= OptExpression[IfStmt_cond]
                 StmtList [IfStmt_trueBranch]
                 StmtList [IfStmt_falseBranch]

OptExpression[IfStmt_cond] ::= ε[IfStmt_cond_ε]
OptExpression[IfStmt_cond] ::= Expression[IfStmt_cond_element]

StmtList[IfStmt_trueBranch]  LISTOF Stmt[IfStmt_trueBranch_element]
StmtList[IfStmt_falseBranch] LISTOF Stmt[IfStmt_falseBranch_element]
```

(a) Grammatikregeln mit Rollen an den Symbolen

```
IfStmt ::= IfStmt_cond fStmt_trueBranch IfStmt_falseBranch

IfStmt_cond ::= ε| UnaryExpression | BinaryExpression | ...
IfStmt_trueBranch  LISTOF IfStmt | WhileStmt | ...
IfStmt_falseBranch LISTOF IfStmt | WhileStmt | ...
```

(b) Grammatikregeln mit Symbolnamen, die ihren Rollen entsprechen

Abbildung 4.3: Reale Abbildung auf kontextfreie Grammatiken

lich der eigentliche Symbolname durch den Namen der Rolle ersetzt werden. Das Ergebnis ist in Abbildung 4.3b dargestellt.

Das allgemeine Verfahren zur Abbildung der Editor-Syntax auf die Repräsentations-Grammatik lautet also wie folgt: Jede nicht-abstrakte DSSL-Klasse wird auf eine Grammatikregel abgebildet. Das Symbol auf der linken Seite ergibt sich aus dem Namen der Klasse. Die Symbole auf der rechten Seite ergeben sich aus der Attribut-Hülle der Klasse. Die Attribut-Hülle ist die Menge der direkt definierten Attribute, vereinigt mit der Menge der direkt oder indirekt geerbten Attribute. Jedes Attribut wird zu einem Symbol auf der rechten Seite, wobei ihm als Präfix der Name der Klasse vorangestellt ist. Zusätzlich gibt es zu jedem Attribut eine oder mehrere Produktionen, die dessen „Inhalt“ bestimmen. SUB*-Attribute werden zu einer LISTOF-Produktion, wobei als Listenelemente alle nicht-abstrakten Unterklassen des Attributtyps erlaubt sind. SUB? und SUB!-Attribute werden zu Kettenproduktionen, wobei auf der rechten Seite alle nicht-abstrakten Unterklassen des Attributtyps erlaubt sind. VAL- und REF-Attribute werden zu Produktionen mit leerer rechter Seite.

Zu dem Abbildungsschema ist anzumerken, dass es eigentlich nicht der Intention von kontextfreien Grammatiken entspricht. Eigentlich werden die Rollen der Symbole durch Regelattributierungen unterschieden. Die hier ge-

wählte Abbildung resultiert hauptsächlich aus der besonderen Art, wie LIDO Wiederverwendung unterstützt. Eine maßgeschneiderte Attributierungssprache könnte sicherlich die Rollen der Symbole, wie sie in Abbildung 4.3a angegeben sind, durch besondere Sprachkonstrukte unterstützen, so dass eine konventionellere Grammatik-Abbildung gewählt werden könnte.

Aus Sicht des Sprachentwicklers hat das Abbildungsschema aber keinen nennenswerten Nachteil. Regelattributierung wird zwar erschwert, was aber kein Problem darstellt, denn dank der genauen Symbolnamen können alle erforderlichen Berechnungen durch Symbolattributierung ausgedrückt werden. Falls bestimmte Symbole einheitlich behandelt werden sollen, kann durch Symbolvererbung eine gemeinsame Oberklasse definiert werden. Der Generator, der die Repräsentations-Grammatik generiert, erzeugt hierzu basierend aus der DSSL-Vererbungshirarchie automatisch Klassensymbole. Im obigen Beispiel erben alle `Stmt`-Unterklassen z.B. automatisch von der LIDO-Symbolklasse `Stmt`.

Aus Sicht des Generators haben die genauen Symbolnamen den Nachteil, dass im Gegensatz zur einfachen Modellierung in Abbildung 4.2c mehr Produktionen benötigt werden. Mit der Anzahl der Produktionen steigt auch die Größe des generierten Attributauswerters. Mit der Anzahl der Kettenproduktionen sinkt die Geschwindigkeit des Attributauswerters. Das ist allerdings praktisch nicht besonders tragisch, da es sich hier nur um einen kleinen konstanten Faktor handelt und Attributauswerter ansonsten sehr effizient sind.

Da der Attributauswerter auf Bäumen basiert, können Attribute natürlich nicht direkt über Querkanten hinweg zugegriffen werden. Stattdessen ist jedem Sprachkonstrukt-Knoten ein Schlüssel zugeordnet, unter dem Eigenschaften des jeweiligen Programmobjekts gespeichert und wieder gelesen werden können. Querreferenzen werden durch solche Schlüssel repräsentiert. Auf diese Weise können z.B. in UML-Klassendiagrammen die Endpunkte von Assoziationen wie folgt berechnet werden: Im Kontext der Symbole, die Klassen repräsentieren, wird unter dem eigenen Schlüssel deren grafische Position abgespeichert. Über die Referenzen `source` und `target` der Assoziation kann diese Position dann ausgelesen und verwendet werden. Die hierbei einzuhaltende Reihenfolge der Operationen lässt sich über Abhängigkeitsattribute spezifizieren. Weitere Einzelheiten zur Modellierung von Querbeziehungen in attributierten Grammatiken sind in [28, S. 102 ff.] zu finden.

Zum hier vorgestellten Ansatz ist abschließend zu bemerken, dass den Symbolen an den Schnittstellen von DSSL-Klassen eigentlich zwei Rollen zugeordnet werden müssten, z.B. `IfStmt_trueBranch_element` aus dem

oberen Kontext und `WhileStmt` aus dem unteren (vgl. Abbildung 4.3a und 4.3b). Wollte man dies durch Symbolnamen kodieren, würde sich die Anzahl der Grammatikproduktionen vervielfachen. Alternativ könnte man auch Kettenproduktionen der Form „`IfStmt_trueBranch_element ::= WhileStmt`“ einführen. Diese würden aber den Einsatz von Berechnungsrollen erschweren und unintuitiver machen. In DEViL wird daher die Rolle des oberen Kontexts nicht explizit als Symbolname kodiert.

4.1.3 Spezifikation visueller Repräsentationen

Um zu demonstrieren, wie auf Basis einer DSSL-Spezifikation und der daraus abgeleiteten Repräsentations-Grammatik eine visuelle Sicht spezifiziert werden kann, benutze ich im Folgenden Nassi-Shneiderman Diagramme als Beispiel. Abbildung 4.4 enthält eine vollständige DEViL-Spezifikation eines Editors für Nassi-Shneiderman Diagramme, der allerdings nur die Konstrukte `IfStmt` und `Command` kennt und in dem Ausdrücke lediglich als einfache Zeichenketten modelliert sind.

Ein Bildschirmfoto des generierten Editors ist in Abbildung 4.5 zu sehen. Der Editor gestattet es, mittels der Sprachkonstrukt-Knöpfe auf der linken Seite Instanzen der Sprachkonstrukt-Klassen `IfStmt`, `SimpleStmt` und `Expression` einzufügen und vorhandene Instanzen dieser Klassen zu verschieben oder zu löschen. Das Layout der Darstellung wird automatisch berechnet.

Da die Grundlagen der verwendeten Spezifikationstechnik sowie die Grundlagen zu visuellen Mustern bereits aus dem Abschnitt 2.4 über VL-Eli bekannt sind, habe ich im Beispiel der Einfachheit halber bereits visuelle Muster verwendet. An dieser Stelle ist jedoch nur das Grundverständnis der Spezifikation wichtig. Details zu visuellen Mustern werden im Abschnitt 4.2 erläutert.

Teil (a) der Spezifikation definiert die Syntax der editierbaren Struktur in DSSL-Notation. Ein Diagramm hat einen Namen und eine Liste von Statements. Statements haben entweder die Klasse `IfStmt` oder `Command`. Das `IfStmt`-Konstrukt hat neben zwei Listen von Unter-Statements auch eine optionale Bedingung der Klasse `Expression`. Der Einfachheit halber haben `Command` und `Expression` keine Unterstruktur, sondern deren Inhalt wird als Zeichenkette modelliert.

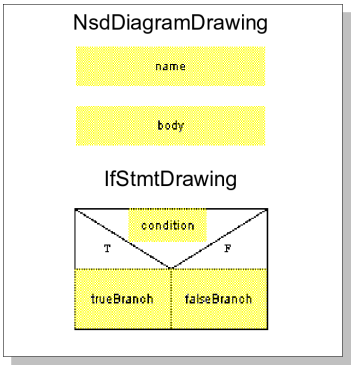
Teil (b) deklariert den Sichttyp `nsdView`. Sichten dieses Typs zeigen die visuelle Repräsentation von Bäumen mit Wurzel `Root` und haben die

```
CLASS Root {
  name: VAL VLString;
  stmts: SUB Stmt*;
}

ABSTRACT CLASS Stmt {}

CLASS IfStmt INHERITS Stmt {
  condition: SUB Expression?;
  trueBranch: SUB Stmt*;
  falseBranch: SUB Stmt*;
}

CLASS Command INHERITS Stmt {
  value: VAL VLString;
}

CLASS Expression {
  value: VAL VLString;
}
```

(a) Syntax der editierbaren Struktur

```
VIEW nsdView ROOT Root {
  BUTTON "If Stmt"
     INSERTS IfStmt;
  BUTTON "Command"
      INSERTS Command;
  BUTTON "Expression"
      INSERTS Expression;
}
```

(b) Deklaration des Sichttyps „nsdView“



(c) Generische Zeichnungen

```
SYMBOL Root
  INHERITS VPRootElement, VPForm
  COMPUTE SYNT.drawing = ADDROF(NsdDiagramDrawing);
END;

SYMBOL Root_name
  INHERITS VPFormElement, VPIdTextPrimitive
  COMPUTE SYNT.formElementName = "name";
          SYNT.font = "_headline";
END;

SYMBOL Root_stmts
  INHERITS VPFormElement, VPSimpleList
  COMPUTE SYNT.formElementName = "body";
END;

SYMBOL Command
  INHERITS VPSimpleListElement, VPSingletonForm
  COMPUTE SYNT.text = THIS.pers_value;
END;

SYMBOL Command_value
  INHERITS VPSingletonFormElement, VPValTextPrimitive
END;

SYMBOL IfStmt
  INHERITS VPSimpleListElement, VPForm
  COMPUTE SYNT.drawing = ADDROF(IfStmtDrawing);
END;

SYMBOL IfStmt_condition
  INHERITS VPFormElement, VPContainer
  COMPUTE SYNT.formElementName = "condition";
END;

SYMBOL IfStmt_trueBranch
  INHERITS VPFormElement, VPSimpleList
  COMPUTE SYNT.formElementName = "trueBranch";
END;

SYMBOL IfStmt_falseBranch
  INHERITS VPFormElement, VPSimpleList
  COMPUTE SYNT.formElementName = "falseBranch";
END;

SYMBOL Expression
  INHERITS VPContainerElement, VPTextPrimitive
  COMPUTE SYNT.text = THIS.pers_value;
END;
```

(d) Attributberechnungen für den Sichttyp „nsdView“

Abbildung 4.4: DEViL-Spezifikation eines Nassi-Shneiderman Editors

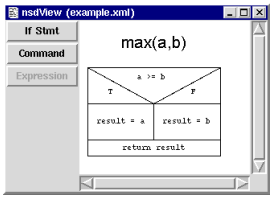


Abbildung 4.5: Bildschirmfoto des generierten Editors für Nassi-Shneiderman Diagramme

Sprachkonstrukt-Knöpfe „If Stmt“, „Command“ und „Expression“, durch die die entsprechenden Sprachelemente eingefügt werden können. Die Sicht-Deklaration bewirkt, dass aus der Editor-Syntax entsprechend des obigen Verfahrens eine Repräsentations-Grammatik für diesen Sichttyp generiert wird. Hierbei werden natürlich nur Produktionen generiert, die auch im entsprechenden Teilbaum angewendet werden können.

In bestimmten Fällen ist es nützlich, die generierte Repräsentations-Grammatik an spezielle Erfordernisse der Sicht anpassen zu können. Dazu gibt es im Rahmen der Sicht-Deklaration spezielle Ausdrucksmittel, auf die in Abschnitt 4.3 näher eingegangen wird.

Teil (c) zeigt zwei so genannte Generische Zeichnungen. Sie werden in Abschnitt 4.4 genauer behandelt. An dieser Stelle habe ich sie nur aufgeführt, um eine vollständige Spezifikation präsentieren zu können. Da die Spezifikationsmethode sehr intuitiv ist, sollte ihr Beitrag grundsätzlich schon jetzt verständlich sein. Prinzipiell sind Generische Zeichnungen Vorlagen grafischer Repräsentationen, die Platzhalter für grafische Unterobjekte enthalten. Die Generische Zeichnung `IfStmtDrawing` hat z.B. die Platzhalter `condition`, `trueBranch` und `falseBranch`, in die bei der Instanziierung konkrete Ausdrücke bzw. Anweisungssequenzen eingesetzt werden.

Teil (d) ist eine attributierte Grammatik, die konkrete Repräsentationen des Sichttyps `nsdView` berechnet. Durch Zuordnung von Berechnungsrollen wie `VPRootElement`, `VPForm` oder `VPTextPrimitive` an bestimmte Grammatiksymbole werden ihnen Attributberechnungen hinzugefügt, die zusammengenommen die Darstellungsberechnung beschreiben. Die geerbten Attributberechnungen können überschrieben oder durch handgeschriebene Attributberechnungen ergänzt werden, um die Details der grafischen Darstellung zu beeinflussen. Beispielsweise wird im Kontext `Root_name` die von der Berechnungsrolle `VPIdTextPrimitive` geerbte Berechnung des `font`-Attributs überschrieben, um die Größe des Diagrammtitels anzupassen.

Die Bedeutung der Symbolrollen ist recht einfach zu verstehen. `Root` erbt von `VPRootElement`, weil dieses Symbol die Wurzel der Darstellung repräsentiert. Des Weiteren wird von `VPForm` geerbt, da die Repräsentation dem Formular-Muster entspricht: Es gibt zwei Unterelemente (nämlich `name` und `stmts`), die visuell eine feste relative Anordnung besitzen. Das Aussehen dieser Formular-Instanz wird durch die Generische Zeichnung `NsdDiagramDrawing` festgelegt. Das Unterelement `Root_name` erbt die Rolle `VPFormElement` um zu kennzeichnen, dass dies ein Teil des Inhalts

des obigen Formulars ist. Durch das Attribut `formElementName` wird festgelegt, in welchen Container diese Teilrepräsentation gehört. Die Repräsentation selbst wird durch die Rolle `IdTextPrimitive` definiert. In diesem Fall wird festgelegt, dass lediglich der Name des Sprachelements als Text angezeigt wird.

Hinter den Kulissen verbergen sich eine Reihe von Mechanismen, damit aus der Spezifikation ein funktionsfähiger Editor wird. Während der Ausführung der geerbten und hinzugefügten Attributberechnungen werden in erster Linie grafische Primitive auf die Zeichenfläche gezeichnet, so dass sich die gewünschte grafische Repräsentation ergibt. Zusätzlich werden jedoch auch unsichtbare Kontextinformation in die grafischen Repräsentation eingefügt, damit der Sprachbenutzer das Programm anhand dieser editieren kann. Beispielsweise werden in die Darstellung unsichtbare Einfügestellen für neue Anweisungen integriert. Betätigt der Sprachbenutzer den Sprachkonstrukt-Knopf für `IfStmt` oder `Command`, werden ihm automatisch die dem Mauszeiger nächstgelegenen passenden Einfügestellen angezeigt. Durch das Einfügen selbst wird die editierbare Struktur geändert, was nach dem Model-View-Paradigma eine Aktualisierung der grafischen Darstellung auslöst, so dass der Attributauswerter die Darstellung neu berechnet.

4.1.4 Konkrete Interaktionsmechanismen

Auf welche Weise der Benutzer mit der grafischen Darstellung interagieren kann, hängt von der Implementierung der visuellen Muster ab. Einige in diesem Sinne interessante Muster-Varianten werden in Abschnitt 4.2.6 vorgestellt. Die dort beschriebenen Editiermechanismen basieren allerdings auf einer allgemein verwendbaren Basis-Bibliothek, die interaktive Grundmechanismen zur Verfügung stellt. Sie macht die Implementierung der Muster-Varianten einfacher und trägt zur Konsistenz der generierten Editoren bei. Nachfolgend sollen die Basismechanismen für Interaktionen und deren Einsatzmöglichkeiten vorgestellt werden.

Die Basis-Bibliothek kapselt die folgenden grundlegenden grafischen Interaktionskonzepte.

- Selektierbarkeit von visuellen Objekten
- Einfügbarkeit von visuellen Objekten durch Selektion von Einfügestellen

- Änderbarkeit von Werten durch Ziehpunkte
- Änderbarkeit von Referenzen durch Ziehpunkte und Objektselektion

Um visuelle Objekte mit der Maus selektierbar zu machen, müssen den grafischen Primitiven bestimmte Zusatzinformationen hinzugefügt werden. Zunächst wird zu jedem Primitiv die Information benötigt, welchen Sprachkonstrukt-Knoten es repräsentiert. Es können durchaus mehrere Primitive zum gleichen Sprachkonstrukt-Knoten gehören. Zusätzlich muss spezifiziert sein, wie eine Selektion grafisch dargestellt werden soll. Standardmäßig werden Objekte mit einem Selektionsrahmen versehen. Wird in Abbildung 4.5 beispielsweise auf eine der schrägen Linien der Fallunterscheidung geklickt, wird in einem ersten Schritt festgestellt, dass der entsprechende Fallunterscheidungs-Knoten der editierbaren Struktur zu selektieren ist. In einem zweiten Schritt wird dann die gesamte Fallunterscheidung durch einen Selektionsrahmen hervorgehoben. Es lassen sich auch abweichende Darstellungen selektierter Objekte spezifizieren. Linien werden z.B. dicker und in einer anderen Farbe gezeichnet, um deren Selektion zu kennzeichnen. Ausgehend von einer Selektion können weitere Operationen durchgeführt werden, beispielsweise kann das selektierte Objekt durch Drücken der Entfernen-Taste oder durch Selektion des entsprechenden Menüpunktes im Kontextmenü gelöscht werden.

Zur Umsetzung des zweiten Spiegelpunkts, der Einfügbarkeit von visuellen Objekten durch Selektion von Einfügestellen, wird die grafische Darstellung um Einfügemarkierungen ergänzt, die im Normalzustand für den Sprachbenutzer unsichtbar sind. Erst, wenn ein neues Sprachkonstrukt eingefügt werden soll, werden die Einfügemarkierungen als visuelle Rückkopplung sichtbar gemacht. Abbildung 4.6 zeigt ein Bildschirmfoto, bei dem der Sprachbenutzer gerade den Knopf „While Loop“ gedrückt hat. Als Reaktion ist der Editor in den Einfügemodus gewechselt. Der Sprachbenutzer muss nun eine Einfügestelle für das neu zu erstellende Sprachkonstrukt auswählen. Als grafische Rückkopplung wird die dem Mauszeiger nächstgelegene gültige Einfügestelle hervorgehoben. Selektiert der Benutzer die gerade hervorgehobene Einfügestelle, würde die Schleife im *true*-Zweig der zweiten Fallunterscheidung vor der dort bereits existierenden Anweisung eingefügt.

Die Einfügemarkierungen werden nicht nur zum Einfügen neuer Sprachkonstrukte, sondern auch beim Verschieben bereits vorhandener verwendet. Wird z.B. in Abbildung 4.6 eine Anweisung selektiert und per *Drag-and-Drop* verschoben, so wird die Zielposition auf genau die gleiche Weise selektiert. Wenn

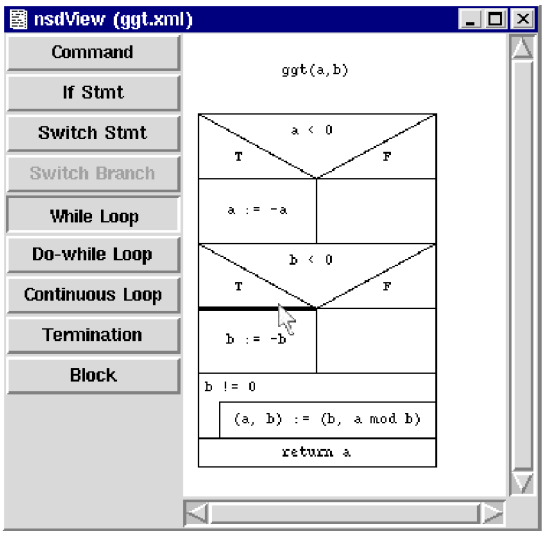


Abbildung 4.6: Editor-Unterstützung zum Einfügen von Listenelementen

einfügbare Objekte wie beim Mengen-Muster frei platzierbar sind, werden während des Einfügens auch die gewählten Zielkoordinaten in der abstrakten Struktur gespeichert.

Ein Beispiel für die Änderbarkeit von Werten durch Ziehpunkte sind die Größen visueller Objekte. Falls der Benutzer die Größe eines visuellen Objekts wie im Fall des Mengen-Musters selbst festlegen darf, besitzt dieses Ziehpunkte, die mit der Maus bewegt werden können. Solche Ziehpunkte sind ähnlich wie in Vektorgrafik-Programmen normalerweise nur im selektierten Zustand sichbar. Verschiebt der Benutzer solche Ziehpunkte, ändert er damit bestimmte persistente Attribute, die z.B. die Größe des Objekts speichern.

Ein Beispiel für den letzten Spiegelpunkt, die Änderbarkeit von Referenzen durch Ziehpunkte und Objektselektion, sind Linienverbindungen. Die Enden einer bereits existierenden Linienverbindung besitzen Ziehpunkte, um das Linienende mit einem anderen Objekt verbinden zu können. Strukturell repräsentieren solche Ziehpunkte REF-Knoten. Bewegt der Benutzer den Ziehpunkt, wird als visuelle Rückmeldung das der Mausposition nächstgelegene visuelle Objekt hervorgehoben, das als neuer Wert des REF-Knotens in Frage kommt.

All diesen Benutzerinteraktionen ist gemein, dass sie konkrete Benutzeraktionen auf abstrakte Änderungen der editierbaren Struktur abbilden. Die Änderungen lassen sich durch die in Abbildung 3.7 auf Seite 86 gezeigten Funktionen ausdrücken. Durch die Änderung der editierbaren Struktur wird ge-

mäß dem Model-View-Paradigma eine Aktualisierung der Sichten auslöst, woraufhin der Attributauswerter die grafische Darstellung neu berechnet.

Die Themen Model-View, Sichten, Darstellungsberechnung und Editieroperationen werden tiefergehend in der Dissertation von Jung [28, S. 125 ff.] behandelt. Da dieses Konzept aus dem VL-Eli Ansatz mehr oder weniger unverändert übernommen wurde, möchte ich an dieser Stelle auf eine ausführlichere Darstellung verzichten.

4.2 Implementierung und Anwendung visueller Muster

Visuelle Muster wurden bereits in Abschnitt 2.4 im Kontext des VL-Eli Systems eingeführt. Dort wurde zwischen (abstrakten) visuellen Mustern und (konkreten) Implementierungsvarianten unterschieden. Abstrakte visuelle Muster sind ein Konzept, das vor allem bei der Analyse und beim Entwurf von Sprachen nützlich ist. Da in diesem Kapitel die Implementierung visueller Sprachen behandelt wird, werden nachfolgend also konkrete Implementierungsvarianten diskutiert.

Muster-Implementierungen in DEViL bestehen aus einer Menge von Berechnungsrollen. Sie werden angewendet, indem die Rollen den Symbolen der Repräsentations-Grammatik zugeordnet werden. Die Berechnungsrollen eines Musters kooperieren miteinander, um die spezifische Darstellung und dessen Layout zu berechnen.

Das Konzept der grafischen Rollen struktureller Symbole ist zugleich intuitiv und mächtig. Ein vergleichbarer Mechanismus wird auch im Zusammenhang mit HTML- und XML-Dateien verwendet und heißt *Cascading Stylesheets* [65]. Hier lassen sich den XML-Symbolen Darstellungsrollen wie Absätze, Überschriften, Aufzählungen, Tabellen usw. zuordnen und deren Formatierung durch Attribute anpassen.

Die Muster-Varianten von DEViL stellen vergleichbare Rollen zur Verfügung. *Cascading Stylesheets* und Muster-Varianten in DEViL unterscheiden sich jedoch in wichtigen Punkten:

- Die Muster-Varianten in DEViL beschränken sich nicht auf die Darstellungsmittel von HTML, sondern ermöglichen auch andere Darstellungen wie Linienverbindungen, Bäume oder Graphen.

- Die Muster-Varianten in DEViL beschreiben nicht nur die Repräsentation und das Layout, sondern kapseln auch geeignete Mechanismen, um die Darstellung zu editieren.
- Die Implementierung der Muster-Varianten in DEViL ist im Gegensatz zur Implementierung von *Cascading Stylesheets* in Bibliotheken auf Spezifikationsniveau gekapselt. Da die Implementierung und Anwendung der Muster-Varianten auf dem gleichen Kalkül basiert, können die Muster-Varianten sehr flexibel an die Wünsche des Sprachentwicklers angepasst werden.

Zusammenfassend können die Muster-Varianten in DEViL also als Verallgemeinerung des *Cascading Stylesheet* Konzepts betrachtet werden. Sie sind genauso intuitiv, aber wesentlich mächtiger und flexibler. Nachfolgend wird die Methode der visuellen Muster genauer ausgeführt und durch Beispiele untermauert.

4.2.1 Rollendiagramme

Zur Implementierung von Muster-Varianten müssen „nur“ entsprechende Rollen definiert und diesen Attributberechnungen zugeordnet werden. Bei der Anwendung der Muster-Varianten muss allerdings sichergestellt werden, dass die aus der Muster-Überdeckung resultierende Attributierung konsistent und vollständig ist. Die Anwendungsbedingungen und Kombinationsmöglichkeiten müssen so einfach beschrieben sein, dass sie auch von unbedarften Sprachentwicklern verstanden und umgesetzt werden können. Um die passende visuelle Muster-Variante zu finden, sollten Sprachentwickler die wesentlichen Merkmale einer Muster-Variante „auf einen Blick“ erfassen können.

Bereits in unserer Diplomarbeit [54] haben wir eine grafische Sprache zur Beschreibung visueller Muster-Varianten entwickelt. Sie basiert auf den Konzepten von UML-Klassendiagrammen, benutzt jedoch eine spezialisierte Symbolik, um die wichtigsten Informationen noch prägnanter darstellen zu können. Die damals entwickelte Sprache hatte allerdings keinen allgemeinen Mechanismus, um die Kombinierbarkeit der Muster-Varianten auszudrücken. Aus diesem Grund stelle ich nachfolgend eine verbesserte Variante vor, mit der die Anwendbarkeit und Kombinierbarkeit von Muster-Varianten mit sehr wenigen und gut erfassbaren Sprachkonstrukten beschrieben werden kann.

Abbildung 4.7: Typische Rollendiagramme für Muster-Varianten

In Abbildung 4.7 sind einige so genannte *Rollendiagramme* zu sehen. Jedes beschreibt die Struktur und die Schnittstellen einer Muster-Variante. Die Diagramme werden in genau dieser Form auch im Muster-Handbuch von DEViL [51] verwendet, um dem Sprachentwickler einen Überblick über die Struktur einer Muster-Variante zu geben. Auch diese Diagramme wurden mit einem von DEViL generierten Editor erstellt.

Die Rollen eines Musters sind als Kästchen, die Randbedingungen zur Anordnung im Strukturbaum durch Linienverbindungen gekennzeichnet. Die Linien repräsentieren nach UML-Terminologie Kompositionen, d.h. existenzabhängige Teil-Ganzes Beziehungen. Das Aussehen einer Linie bestimmt die Kardinalität der untergeordneten Struktur. Auseinanderlaufende Linien wie in Abbildung 4.7a bedeuten, dass der `VPSimpleList`-Knoten mit beliebig vielen `VPSimpleListElement` Knoten in Beziehung stehen darf

(Kardinalität „0..*“). Eine gestrichelte Linie wie in Abbildung 4.7b bedeutet Kardinalität „0..1“ und eine durchgezogene Linie wie in Abbildung 4.7e bedeutet Kardinalität „1“. Die Beziehung der Objekte ergibt sich dabei durch die Anordnung im Baum. Ein `VPSimpleList`-Knoten steht mit einem `VPSimpleListElement`-Knoten genau dann in Beziehung, wenn sich der letztere in einem Unterbaum des ersteren befindet und es dazwischen keinen weiteren `VPSimpleList`-Knoten gibt. Die angegebenen Kardinalitäten müssen für alle aus der Repräsentations-Grammatik ableitbaren Bäume erfüllt sein.

Die Teildiagramme a, b, c und d zeigen Muster-Varianten, die auch im Beispiel in Abbildung 4.4 auf Seite 136 vorkommen. Anhand der Diagramme kann also die Gültigkeit der Spezifikation geprüft werden. Nach Rollendiagramm 4.7b darf z.B. einem Knoten mit der Rolle `VPContainer` nur höchstens ein `VPContainerElement`-Knoten zugeordnet sein. Die Rolle `VPContainer` wird im Beispiel von `IfStmt_condition` geerbt, die Rolle `VPContainerElement` von `Expression`. Entsprechend der Editor-Grammatik hat das `condition`-Attribut von `IfStmt` die Kardinalität „?“. Dies überträgt sich auf die Repräsentations-Grammatik, so dass tatsächlich einem `IfStmt_condition`-Knoten nie mehr als ein `Expression`-Knoten zugeordnet ist. Anders herum wird `Expression` an keiner anderen Stelle der Editor-Syntax verwendet, so dass einem `VPContainerElement`-Knoten tatsächlich immer ein `VPContainer`-Knoten übergeordnet ist.

Visuelle Muster-Varianten können kombiniert werden, indem ein Symbol der Repräsentations-Grammatik mehrere Muster-Rollen erbt. In Abbildung 4.4 erbt `Expression` z.B. von `VPContainerElement` und `VPTextPrimitive`. Welche Kombinationen erlaubt sind, wird durch die so genannten Rollen-Schnittstellen bestimmt. Die Rollen-Schnittstellen sind im mittleren und unteren Teilbereich eines Rollen-Knotens aufgeführt. Der mittlere Bereich ist rot gefärbt und enthält die *geforderten* Schnittstellen, der untere Bereich ist grün gefärbt und enthält die *bereitgestellten* Schnittstellen. Auch bei einem Schwarzweiß-Ausdruck lässt sich die Anordnung der Teilbereiche leicht merken, denn die Ordnung der Farben - oben rot, unten grün - entspricht der einer Ampel.

Damit die Anwendung der Muster-Rollen an einem Knoten konsistent und vollständig ist, müssen zwei Bedingungen erfüllt sein: (1) Alle geforderten Schnittstellen müssen durch eine andere vom gleichen Symbol geerbte Rolle bereitgestellt sein und (2) keine zwei Rollen dürfen die gleiche Schnittstelle bereitstellen. Betrachtet man die Rollen `VPContainerElement` und

`VPTextPrimitive` in Abbildung 4.7b bzw. 4.7d stellt man fest, dass sie sich bzgl. ihrer Rollen-Schnittstellen perfekt ergänzen, sie zusammengenommen also die Berechnungen an einem Knoten der Repräsentationsstruktur konsistent und vollständig spezifizieren.

Aus der obigen Konsistenzbedingung wird der Sinn der Diagramme 4.7c und 4.7d klar. Solche Einzelrollen werden im Muster-Handbuch [51] „Endstücke“ genannt. Diese werden benötigt, da man mit Muster-Varianten wie in 4.7a oder 4.7b, die aus mehreren Rollen bestehen, alleine keine gültige Überdeckung aufbauen kann. Mit den Endstücken müssen die Muster-Überdeckungen nach oben und nach unten abgeschlossen werden. Grafisch repräsentiert das obere Endstück die gesamte Sicht und die unteren Endstücke repräsentieren visuelle Objekte ohne Unterstruktur.

Abbildung 4.7e zeigt, dass das Rollendiagramm eines Musters durchaus komplexer sein kann. In dem Diagramm sind die Berechnungsrollen für Linienverbindungen mit orthogonalem Linienverlauf dargestellt. `VPConnectionArea` repräsentiert den grafischen Bereich, in dem sich die Linien befinden. `VPConnectionEndpoint` repräsentiert die grafischen Objekte, die potenziell mit Linien verbunden werden können. Die Attributberechnungen dieser Rolle legen z.B. fest, in welcher Form Objekt und Linie miteinander verbunden sind. Die Rolle `VPConnectionList` repräsentiert das Attribut der editierbaren Struktur, das die Linien-Konstrukte speichert. Dieser Kontext wird benötigt, um neue Linienverbindungen erzeugen zu können. Die Rolle `VPOrthoConnection` repräsentiert die eigentliche Linienverbindung. Hier wird u.a. der Verlauf der Linie berechnet. Die Rollen `VPConnectionFrom` und `VPConnectionTo` repräsentieren den Start- bzw. Endpunkt der Linie. Zum einen wird dadurch der Bezug zu den REF-Attributen der editierbaren Struktur hergestellt, die den Start- und Endpunkt der Linie speichern. Zum anderen ist dieser Berechnungskontext wichtig, um den Endpunkten weitere Eigenschaften wie z.B. Beschriftungen zuordnen zu können.

4.2.2 Parametrisierung von Musteranwendungen

Zur vollständigen Beschreibung eines Musters gehört neben dem Rollendiagramm auch eine Auflistung der so genannten *Kontrollattribute* der Musterrollen. Kontrollattribute ermöglichen es, die Repräsentations- und Layoutdetails der Muster-Varianten flexibel zu konfigurieren. Sie modellieren also

parametrisierbare Eigenschaften der Berechnungsrollen. Zwei wichtige Kategorien von Kontrollattributen sind *Layoutattribute* (z.B. der minimale Abstand zwischen Listenelementen) und *Darstellungsattribute* (z.B. das Aussehen eines umschließenden Rahmens). Technisch sind Kontrollattribute „normale“ Attribute meist einfacher Datentypen, denen häufig bereits ein konstanter, sinnvoller Standardwert zugeordnet ist. Durch Überschreiben des Standardwertes kann das Muster den jeweiligen Bedürfnissen angepasst werden. Da die Effekte der Kontrollattribute normalerweise unabhängig voneinander sind, ist die Konfiguration von Muster-Varianten im Allgemeinen sehr einfach. Aufgrund der Standardbelegungen braucht häufig nur ein Bruchteil der bereitgestellten Kontrollattribute überschrieben zu werden.

Die Menge aller Konfigurationsattribute bildet das so genannte *Konfigurationsmodell* einer Muster-Variante. Da jedes Muster individuelle Layout- und Darstellungsmerkmale hat, besitzt im Allgemeinen auch jede Muster-Variante ihr eigenes Konfigurationsmodell. Es kommt aber häufig vor, dass die Konfigurationsmodelle verschiedener Muster-Varianten zumindest gleiche Teile besitzen oder auf den gleichen Konzepten basieren.

Abbildung 4.8 illustriert das Konfigurationsmodell der Muster-Variante `SimpleList`. Layoutattribute sind in nicht-kursiver, Darstellungsattribute in kursiver Schrift gesetzt. Das Attribut `direction` bestimmt die Layout-Richtung der Liste. Gültige Werte sind `North`, `East`, `South` und `West` bzw. `Horizontal` für `East` und `Vertical` für `South`. Die Attribute `alignX` und `alignY` bestimmen die horizontale bzw. vertikale Ausrichtung der Listenelemente. Gültige Werte sind `Left`, `Right`, `Center` und `Scale` für die horizontale Ausrichtung und `Top`, `Bottom`, `Center` und `Scale` für die vertikale. Durch die Attribute `pad`, `elementDistance` und `minWidth` können Feinheiten des Layouts abgestimmt werden. Die Darstellungseigenschaften können durch die Attribute `bgFillColor`, `bgOutlineColor` sowie `separatorStyle` angepasst werden. `bgFillColor` und `bgOutlineColor` bestimmen die Farben von Hintergrund und Rahmen, durch `separatorStyle` wird ggf. die Farbe, Linienstärke und Strichelung einer Separator-Linie zwischen den Listenelementen gewählt. All diese Konfigurationsattribute sind mit sinnvollen Werten vorbelegt, so dass im Allgemeinen bei jeder Musteranwendung nur wenige überschrieben werden müssen.

Die meisten Muster-Varianten haben vergleichbare Konfigurationsmodelle. Ausnahmen sind die Muster-Varianten `Form` und `FormList`, die mittels einer speziellen visuellen Sprache, den so genannten Generischen Zeichnungen

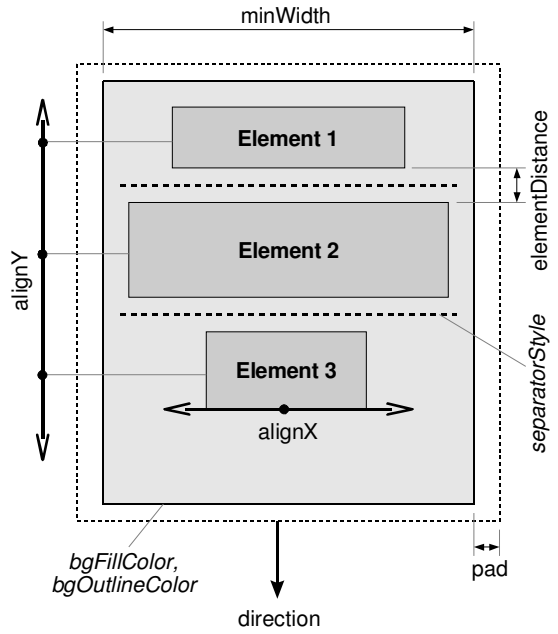


Abbildung 4.8: Parametrisierung der Muster-Variante `SimpleList`

(siehe Abschnitt 4.4), konfiguriert werden. Bei anderen Muster-Varianten wie z.B. `VPContainer` gibt es weniger Kontrollattribute, da es dort nicht so viele Variationsmöglichkeiten gibt.

4.2.3 Musterübergreifende Layoutstrategien

Im Abschnitt wurden 4.2.1 wurden die Rollendiagramme eingeführt, mit denen sich die Kombinationsmöglichkeiten der Muster-Varianten beschreiben lassen. Der Anwender der Muster-Varianten muss sich im Allgemeinen keine Gedanken darüber machen, auf welchen Konzepten die Kombinierbarkeit der Muster basiert. Als Entwickler einer oder mehrerer Muster-Varianten ist es jedoch wichtig sicherzustellen, dass eine Muster-Variante möglichst flexibel mit anderen kombiniert werden kann und dass alle laut Rollendiagramm erlaubten Kombinationen auch zu sinnvollen Ergebnissen führen. Der Schlüssel hierfür sind die Rollen-Schnittstellen in Kombination mit musterübergreifenden Layoutstrategien.

Musterübergreifende Layoutstrategien sind Abhängigkeitsschemata zwischen Einzelentscheidungen beim Layoutprozess. Solche Abhängigkeitsschemata schränken die wechselseitigen Abhängigkeiten der Layoutentscheidungen ein und ermöglichen es, die Layoutberechnung effizient durchzuführen.

Die einzelnen Layoutstrategien sind meist nicht universell, sondern auf eine bestimmte Klasse visueller Repräsentationen zugeschnitten.

Zur Implementierung der Muster-Varianten in DEViL wurden die folgenden vier musterübergreifenden Layoutstrategien verwendet.

- GrowingBox-Layout
- ParBox-Layout
- Flow-Layout
- Layer-Layout

GrowingBox-Layout ist eine Strategie zum Layout geschachtelter Objekte. In einer ersten Phase wird von innen nach außen der Größenbedarf der geschachtelten Objekte propagiert. Der Größenbedarf eines Objekts kann dabei vom Größenbedarf eingeschachtelter Objekte abhängen. In einer zweiten Phase wird der verfügbare Bildbereich für die Objekte von außen nach innen propagiert. Der verfügbare Bereich kann dabei ggf. größer sein als der benötigte. In diesem Fall hat ein Objekt die Möglichkeit, diesen zusätzlichen Platz zu nutzen oder ihn an seine Unterobjekte weiterzugeben. Diese Strategie wird häufig zur Berechnung eines größenoptimierten Layouts geschachtelter Strukturen wie z.B. bei Nassi-Shneiderman Diagrammen eingesetzt. Der Name „GrowingBox-Layout“ ist frei erfunden.

ParBox-Layout ist ebenfalls eine Strategie zum Layout geschachtelter Objekte. In einer ersten Phase wird von außen nach innen die maximale Breite der Objekte propagiert. In einer zweiten Phase wird von innen nach außen die erforderliche Höhe und Breite der Objekte propagiert. In einer dritten Phase werden von außen nach innen die endgültigen Bildregionen für die Objekte verteilt. Diese Strategie findet man z.B. beim Layout von geschachtelten HTML-Tabellen in Webbrowsern. Die Maximalbreite wird hier u.a. durch die Größe des Browser-Fensters bestimmt. Die Bezeichnung „ParBox-Layout“ stammt aus dem Textsatzsystem TeX [34] das eine ähnliche Layoutstrategie verfolgt.

Flow-Layout ist eine Strategie zum Layout von Sequenzen und sequenzialisierbaren Strukturen. Die Einzelelemente werden so lange horizontal nebeneinander angeordnet, bis aus bestimmten Gründen ein Zeilenumbruch stattfindet. In der hier verwendeten Variante sind die Zeilen linksbündig ausgerichtet, wobei allerdings eine Einrückungstiefe mit berücksichtigt wird, die bei

geschachtelten Strukturen zum Tragen kommt. Diese Layoutstrategie wird z.B. bei der Formatierung textueller Programme verwendet. Die Bezeichnung „Flow-Layout“ ist durch den gleichnamigen Layoutmanager in Java inspiriert, der eine ähnliche Strategie verfolgt und häufig zum Layout der Knopfzeile in Dialogen verwendet wird.

Layer-Layout ist eine Strategie zum Layout von heterogenen Objektmengen. Nach dieser Strategie werden die Objekte basierend auf ihren Klassen in Ebenen eingeteilt. Die Ebenen werden dann schrittweise bearbeitet. In jedem Schritt wird das Layout einer Ebene berechnet, wobei Layoutinformationen der bereits bearbeiteten Ebenen verwendet aber nicht mehr geändert werden können. Diese Strategie wird z.B. für das Routing von Linien verwendet. Die erste Ebene sind hier die Objekte, die durch Linien zu verbinden sind und die zweite Ebene sind die Linien selbst. Bei dieser Strategie haben die Linien also keinen Einfluss auf die Positionierung der verbundenen Objekte. Der Begriff „Layer-Layout“ ist von Grafik-Editoren inspiriert, bei denen die Einzelelemente ebenfalls in Ebenen angeordnet werden können, um die Bearbeitung zu vereinfachen.

Man beachte, dass diese musterübergreifenden Layoutstrategien nicht mit den Layoutberechnungen innerhalb eines Musters verwechselt werden dürfen. Zum Layout innerhalb eines Musters werden teilweise wesentlich komplexere Strategien verwendet. Im Gegensatz zum *Layer-Layout* können z.B. innerhalb eines Graph- oder Baum-Musters die Kanten sehr wohl Einfluss auf die Positionierung der Knoten haben.

Bedeutung der Rollen-Schnittstellen Die Rollen-Schnittstellen bieten die Möglichkeit, die oben beschriebenen musterübergreifenden Abhängigkeitsschemata zu modellieren und umzusetzen. Technisch repräsentiert jede Rollen-Schnittstelle einer Menge von Attributen. Bei bereitgestellten Rollen-Schnittstellen enthält die Rolle Berechnungen dieser Attribute, bei geforderten Rollen-Schnittstellen werden die Attribute in anderen Berechnungen verwendet. Durch die Attributabhängigkeiten im Inneren der Muster-Varianten ergeben sich beim Kombinieren der Muster-Varianten die oben beschriebenen musterübergreifenden Strategien. Die Strategien werden jedoch nirgends explizit spezifiziert, sondern werden vom Attributauswerter-Generator aus der Abhängigkeitsstruktur abgeleitet.

Am Beispiel der Muster-Variante `SimpleList` (siehe Abbildung 4.7a) lässt sich die Bedeutung der Rollen-Schnittstellen verdeutli-

chen. Zunächst wird durch Lesen der geforderten Rollen-Schnittstelle `VPContainerSizeInterface` von `VPSimpleListElement` die bevorzugte Größe der Listenelemente bestimmt. Anschließend wird hieraus die bevorzugte Größe der Gesamtliste berechnet und über die bereitgestellte Schnittstelle `VPContainerSizeInterface` dem `VPSimpleList`-Knotens verfügbar gemacht. Basierend auf dieser Information wird von einem anderen Muster der zu benutzende Darstellungsbereich innerhalb der Gesamtdarstellung berechnet und über die Schnittstelle `VPContainerPositionInterface` von `VPSimpleList` verfügbar gemacht. Dieser Darstellungsbereich kann dann mittels der bereitgestellten Schnittstelle `VPContainerPositionInterface` von `VPSimpleListElement` auf die Listenelemente verteilt werden. Stellt man sich mehrere ineinander geschachtelte Listen vor, ergibt sich das oben beschriebene *GrowingBox-Layout*.

Dem aufmerksamen Leser wird nicht entgangen sein, dass die Schnittstellen `VPAggregateInterface` und `VPCompactObjectInterface` von `VPSimpleList` für diesen Layoutprozess keine Rolle spielen. Die geforderte Schnittstelle `VPAggregateInterface` stellt die Funktionalität zur Verfügung, weitere Listenelemente in den Baum einfügen zu können. Sie wird zur Implementierung der interaktiven Eigenschaften der Liste benötigt. Die Schnittstelle wird automatisch von allen Baumsymbolen bereitgestellt, die SUB-Knoten repräsentieren. Die bereitgestellte Schnittstelle `VPCompactObjectInterface` ist eine Serviceleistung für weitere Rollen, die mit dieser Rolle kombiniert werden können. Dadurch ist es u.a. möglich, die Liste imsgesamt als Endpunkt einer Linienverbindung darzustellen. Im Sinne der oben beschriebenen Layoutstrategien ist dies also eine Schnittstelle für das *Layer-Layout*.

Für die Praxis ist es wichtig, dass nicht nur eine der oben beschriebenen „reinen“ globalen Strategien, sondern auch Kombinationen daraus verwendet werden können. Beispielsweise soll das Layout eines komplexen Objekts nach dem *GrowingBox-Layout* erfolgen, wobei die ebenfalls enthaltenen textuellen Anteile jedoch nach dem *Flow-Layout* formatiert werden und gleichzeitig noch Linienverbindungen basierend auf dem *Layer-Layout* gezeichnet werden sollen. Auch dies lässt sich mit Hilfe der Schnittstellen auf natürliche Weise ausdrücken. Wie bereits oben erwähnt, lässt sich durch zusätzlich bereitgestellte Schnittstellen die Kombinierbarkeit mit dem *Layer-Layout* erreichen. Um die Kombinierbarkeit der anderen Layout-Strategien zu erreichen, können so genannte Adapter eingeführt werden.

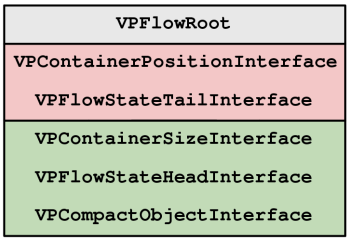


Abbildung 4.9: Adapter zur Kombination von *GrowingBox* und *Flow-Layout*

Abbildung 4.9 zeigt z.B. einen Adapter, der eine Teilrepräsentation mit *Flow-Layout* in eine übergeordnete Repräsentation mit *GrowingBox-Layout* integriert. Über die geforderte Schnittstelle `VPFlowStateTailInterface` wird dazu zunächst die bevorzugte Größe des *Flow-Layouts* beschafft und durch die bereitgestellte Rollen-Schnittstelle `VPContainerSizeInterface` an das umgebende *GrowingBox-Layout* weitergegeben. Dieses legt dann über die Schnittstelle `VPContainerSizeInterface` den zu verwendenden Bildschirmbereich fest, der dann mittels der bereitgestellte Schnittstelle `VPFlowStateHeadInterface` an das *Flow-Layout* weitergegeben wird. Adapter dienen also zur „Übersetzung“ der Layoutinformationen an den Grenzen zwischen verschiedenen Strategien. In DEViL gibt es neben dem Adapter GrowingBox → Flow auch die Adapter Flow → GrowingBox und GrowingBox → ParBox.

4.2.4 Was ist mit constraint-basiertem Layout?

Zur Layoutberechnung in visuellen Editoren werden häufig Constraint-Solver eingesetzt. Der Leser mag sich wundern, warum diese im vorherigen Kapitel keine Rolle spielten. Tatsächlich ist in DEViL derzeit kein allgemeiner Constraintsolver integriert. Das ist aber nicht prinzipbedingt, sondern lediglich darin begründet, dass diese Technik in DEViL „nicht benötigt“ wird.

Um diese Aussage zu verstehen, muss man sich zunächst die Rolle der Constraintsolver-Methode im Kontext visueller Editoren verdeutlichen. Constraint-Netzwerke sind in erster Linie eine Spezifikationsmethode. Sie wird von vielen Systemen eingesetzt, da Layout-Abhängigkeiten auf diese Weise vergleichsweise einfach spezifiziert werden können. Ein besonderer Vorteil ist, dass Constraint-Netzwerke deklarativ sind und keine vorgeplanten Lösungsstrategien im Sinne des vorherigen Abschnitts erfordern. Da auch

die Spezifikationsmethode basierend auf visuellen Mustern deklarativ ist und die Lösungsstrategien ebenfalls für den Anwender transparent sind, wird die Constraintsolver-Methode auf dieser Ebene nicht benötigt.

Vergleicht man die Mächtigkeit beider Spezifikationsmethoden, so schneiden visuelle Muster wesentlich besser ab, denn bei der Constraintsolver-Methode gibt es prinzipbedingte Einschränkungen der Ausdrucksstärke. Das liegt daran, dass in der Regel die gültigen Formeln so eingeschränkt werden, dass die Formelsysteme noch effizient lösbar sind. Einige Constraintsolver beschränken sich z.B. auf lineare Gleichungen und Ungleichungen. Da sich mit linearen Gleichungssystemen aber nicht alle Layoutaufgaben adäquat beschreiben lassen, haben manche Constraintsolver spezielle Erweiterungen. Der Constraintsolver Parcon [20] hat z.B. ein spezielles Konstrukt zur Modellierung von Kreisen sowie die Möglichkeit, Constraints disjunktiv zu verknüpfen. Diskrete Layoutentscheidungen wie Umbrüche oder automatisches Graphlayout liegen natürlich trotzdem weiterhin jenseits der Möglichkeiten eines Constraintsolvers.

Ein weiterer Nachteil der Constraintsolver-Methode ist ihre „Unberechenbarkeit“. Bei der Spezifikation des Constraintnetzwerks braucht der Lösungsweg nicht betrachtet zu werden, aber genau aus diesem Grund ist es im Allgemeinen unklar, ob die sich ergebenden Constraintnetzwerke in allen Fällen lösbar sind und wie lange es dauert, eine Lösung zu finden. Außerdem ist die Reaktion des Constraintsolvers auf bestimmte Änderungen manchmal anders, als der Benutzer dies erwartet. Ein Entwurfsprinzip von Constraintsolvern ist zwar, die Änderungen möglichst klein zu halten, um den Benutzer nicht übermäßig zu verwirren. Aus eigener Erfahrung weiß ich aber, dass es schwierig sein kann Constraintnetzwerke so zu entwerfen, dass sie mit den Erwartungen des Benutzers übereinstimmen. Schließlich sind Constraintsolver prinzipbedingt nicht so effizient wie Algorithmen, bei denen der Lösungsweg schon vorgeplant ist. Aus diesem Grund betrachten einige Ansätze die Constraintsolver-Methode lediglich als „prototypischen Spezifikationsmechanismus“ [37, S. 135].

Constraint-Netzwerke lassen sich aber auch als Implementierungsmethode verstehen und sind in dieser Rolle auch für das DEViL-System interessant. Besonders hervorzuheben sind hier die ungerichteten Abhängigkeiten zwischen Layoutvariablen, die mit anderen Mechanismen nur schwer zu erreichen sind. Weiterhin ist das Hinzufügen von Constraints zu einem bestehenden Constraint-Netzwerk sehr einfach, so dass die Anpassbarkeit von Muster-Varianten dadurch verbessert werden könnte. Aus diesem

Grund sind Constraint-Netzwerke als Implementierungsmethode bestimmter Muster-Varianten durchaus sinnvoll. Dies hätte auch den Vorteil, dass die entsprechenden Muster die effiziente Lösbarkeit des generierten Constraint-Netzwerkes sicherstellen könnten. Außerdem könnten hier Erfahrungen einfließen, wie Constraint-Netzwerke formuliert werden müssen, damit die Reaktionen den Erwartungen des Benutzers entsprechen.

Die Integration eines constraint-basierten Layouts läßt sich einfach bewerkstelligen, indem eine weitere „musterübergreifenden Layoutstrategie“ im Sinne von Abschnitt 4.2.3 eingeführt wird. Nach der „Constraint-Layout“-Strategie würden die Muster-Rollen zunächst ihren Nachbarn die von ihnen verwendeten Constraint-Variablen bekannt geben. Im Anschluss daran würden die Beiträge der Muster-Anwendungen zum Constraint-Netzwerk in beliebiger Reihenfolge eingesammelt. In einem letzten Schritt würde das Constraint-Netzwerk dem Solver übergeben und die Ergebnisse verteilt. Im VL-Eli System wurde dieser Ansatz bereits erfolgreich umgesetzt [28, S. 131 ff.]. Da sich in DEViL alle Layoutberechnungen auch ohne Constraintsolver umsetzen ließen, sprach der Wartungsaufwand zunächst gegen die Integration eines Constraintsolvers. Im Rahmen der Weiterentwicklung orthogonaler Muster-Eigenschaften wäre die Integration der „Constraint-Layout“-Strategie aber eine lohnende Aufgabe.

4.2.5 Übersicht über die implementierten Muster-Varianten

Die Tabelle 4.1 gibt einen Überblick über die Muster-Varianten, die derzeit in DEViL implementiert sind. Diese Liste soll das Einsatzspektrum visueller Muster verdeutlichen. Natürlich erhebt die Muster-Bibliothek in DEViL keinen Anspruch auf Vollständigkeit, denn eine solche Bibliothek kann prinzipiell niemals die Wünsche und Vorstellungen aller Sprachentwickler hundertprozentig abdecken. Es hat sich aber gezeigt, dass die hier aufgeführte Menge von Muster-Varianten bereits eine sehr große Klasse visueller Sprachen abdeckt. Dies würde selbst dann noch gelten, wenn man die Muster-Bibliothek erheblich verkleinern würde, denn wie in Kapitel 5 (Abschnitt 5.2.7) gezeigt wird, ist die Anwendungshäufigkeit der verschiedenen Muster-Varianten sehr unterschiedlich. In allen umgesetzten Beispielen wurde die Muster-Variante `Form` z.B. ca. 50 mal angewendet, während es insgesamt nur 2 Anwendungen der Muster-Variante `Table` gibt.

Um dem Leser eine Vorstellung vom Funktionsumfang der Muster-Varianten zu geben, sind in Abbildung 4.10 schematische Darstellungen aufgeführt. Je-

Tabelle 4.1: Implementierte Muster-Varianten

Muster-Variante	Layout-Konzept	Benutzerdefinierte Layout-Vorgaben	Kontroll-attribute
SingletonForm	GrowingBox	-	4
SimpleForm	GrowingBox	-	10
Form	GrowingBox	-	6
SimpleList	GrowingBox	Element-Reihenfolge	10
FormList	GrowingBox	-	4
Container	GrowingBox	-	6
Set	GrowingBox	Element-Positionen, Gesamtgröße	6
Table	GrowingBox	Zeilen-Reihenfolge	11
Matrix	GrowingBox	Zeilen/Spalten-Reihenfolge	6
Tree	GrowingBox	Unterbaum-Reihenfolge	9
ExplorerTree	GrowingBox	Unterbaum-Reihenfolge	9
FlowForm	Flow	-	1
FlowList	Flow	Element-Reihenfolge, Zeilenumbrüche	2
FlowContainer	Flow	-	1
ParboxList	Parbox	Element-Reihenfolge	10
Connection	Layer	Position der Stützpunkte	6
OrthoConnection	Layer	Position der Zwischensegmente	6
RefConnection	Layer	Position der Stützpunkte	6
RefPolyConnection	Layer	Position der Zwischensegmente	6
LabelAtLine	Layer	Position relativ zur Linie	0

de Muster-Variante basiert auf einem zugrundeliegenden abstrakten Muster im Sinne von Abschnitt 2.4.3, das am Namen der Variante abzulesen ist.

Die Variante `SingletonForm` ist ein Spezialfall des Formular-Musters, bei dem es genau ein darzustellendes Unterelement gibt. `SimpleForm` ist ebenfalls ein Spezialfall des Formular-Musters, bei dem die Unterelemente entweder horizontal oder vertikal zueinander ausgerichtet sind. `Form` ist die allgemeinste Variante des Formular-Musters, bei der die Darstellung im Gegensatz zu den vorher genannten Varianten nicht durch Kontrollattribute, sondern durch eine Generische Zeichnung (siehe Abschnitt 4.4) spezifiziert wird.

`SimpleList` ist eine Variante des Listen-Musters, bei der die Listenelemente nur horizontal oder vertikal ausgerichtet werden können. `FormList` ist eine Kombination des Listen- und Formularmusters, bei dem das Layout flexibler spezifiziert werden kann und so z.B. auch diagonal verlaufende Listen möglich sind. `Container` ist eine Variante des Container-Musters, wobei ein Container entweder genau ein Unterelement aufnehmen oder leer sein kann. `Set` ist eine Variante des Mengen-Musters, bei dem Unterelemente beliebig innerhalb einer vorgegebenen Region platziert werden können.

`Table` ist eine Variante des Tabellen-Musters, das eine Liste von Tupeln gleicher Struktur tabellenartig visualisiert. `Matrix` ist eine Variante des Matrix-Musters. Im Gegensatz zu einer Tabelle sind hier alle Zellen gleichberechtigt und es können sowohl neue Zeilen als auch neue Spalten eingefügt werden. `Tree` und `ExplorerTree` sind zwei Varianten des Baum-Musters. Bei der `Tree`-Variante werden Bäume in der unter Informatikern gebräuchlichen visuellen Notation repräsentiert. Die Repräsentation der `ExplorerTree`-Variante erinnert an die Darstellung in Dateimanagern. `FlowForm`, `FlowList` und `FlowContainer` sind Varianten der `Form`-, `List`- bzw. `Container`-Muster, die auf das *Flow-Layout* spezialisiert sind. `ParBoxList` ist eine weitere Listen-Variante mit *ParBox-Layout*. Die vier Muster-Varianten `Connection`, `OrthoConnection`, `RefConnection` und `RefOrthoConnection` sind Varianten des Linien-Musters. `Connection` und `OrthoConnection` unterscheiden sich dabei in der Art des Linienverlaufs. Während `Connection` beliebige Linienverläufe erlaubt, erzwingt `OrthoConnection` Verbindungen, die nur aus horizontalen und vertikalen Segmenten bestehen. Die Varianten mit Präfix „Ref“ sind auf eine andere zugrundeliegende Struktur zugeschnitten. Hier gehört die Linie zu ihrem Startknoten und ist fest mit diesem verbunden. `LabelAtLine` implementiert eine Linienbeschriftung, die eine beliebige grafische Repräsentation besitzen kann.

Wie schon in den Beschreibungen deutlich wurde, ist die Zugehörigkeit

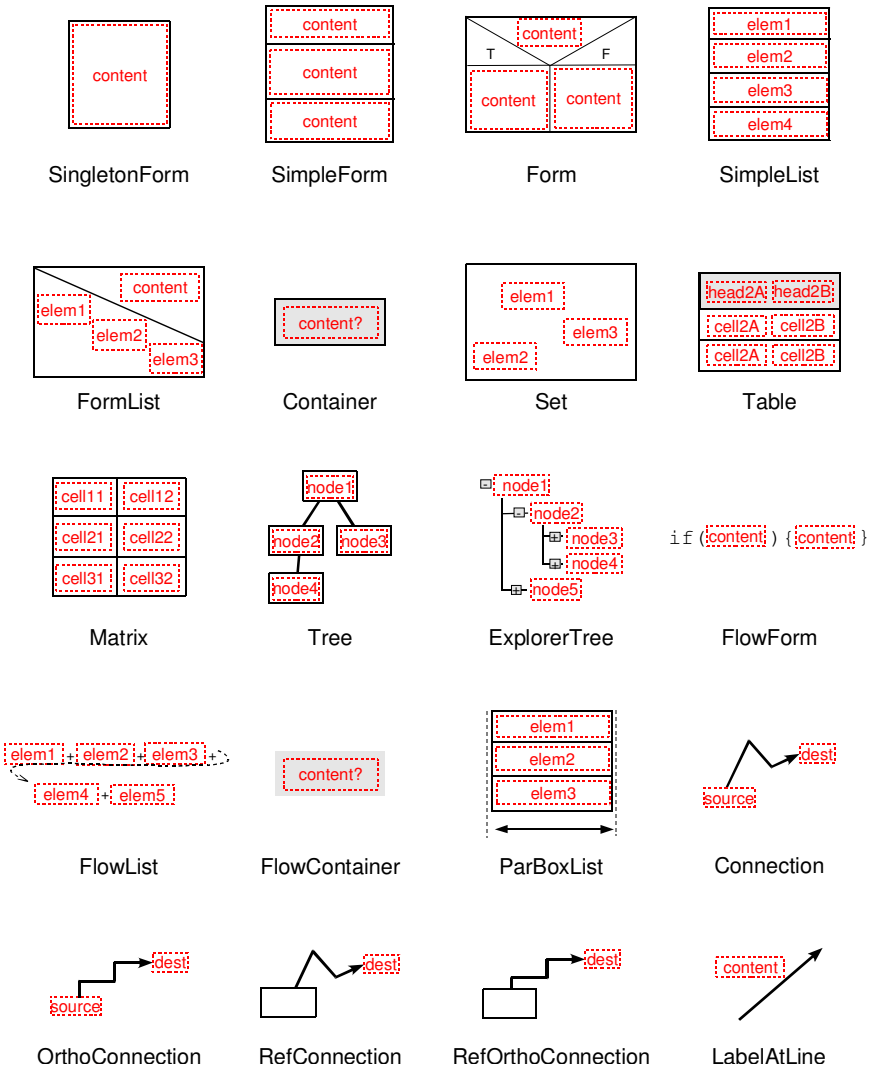


Abbildung 4.10: Schematische Darstellung der implementierten Muster-Varianten

zum zugrundeliegenden (abstrakten) visuellen Muster für das Verständnis einer Muster-Variante äußerst nützlich. Die Muster-Varianten `SimpleList`, `FlowList` und `ParboxList` sind z.B. Variationen des abstrakten Listen-Musters, nach dem Objekte visuell in einer linearen Sequenz angeordnet werden. Die Varianten unterscheiden sich im Layout. Bei der Variante `SimpleList` ist die Liste ein kompaktes Objekt, dessen Größe durch die Listenelemente bestimmt wird. Bei der Variante `ParboxList` wird die erlaubte Gesamtbreite von außen vorgegeben, so dass die Größe der Listenelemente hierdurch beeinflusst werden kann. Bei der Variante `FlowList` wird die Liste in einen größeren „Fluss“ von Objekten integriert, wobei Zeilenumbrüche auftreten können und die Liste nicht mehr notwendigerweise kompakt ist.

Bei den Muster-Varianten `Connection` und `OrthoConnection` sind nicht nur die Darstellung und das Layout, sondern auch die Interaktionsmechanismen unterschiedlich. Während die `Connection`-Variante Linienverbindungen mit beliebig definierbaren Zwischenpunkten realisiert, verlaufen die Liniensegmente bei der `OrthoConnection`-Variante stets orthogonal. Das bedeutet, dass zum Editieren des Linienverlaufs nicht wie bei der `Connection`-Variante Punkte, sondern ganze Liniensegmente verschoben werden müssen.

Wie man erkennt gibt es im Fall der `Connection`-Variante zwei orthogonale Unterdimensionen. Die eine legt die Art der Linienführung fest, die andere bestimmt die Struktur, auf die die Variante angewendet werden kann. Natürlich wird in der Implementierung hierfür eine gemeinsame Basis benutzt. Lediglich zur Präsentation werden die Varianten als Einzelmuster behandelt, um sie möglichst einfach vermitteln zu können. Auch Varianten verschiedener Muster können eine gemeinsame Basis haben. Ein Beispiel sind die Varianten `SimpleForm` und `SimpleList`, die zwar vollkommen andere Strukturen repräsentieren, deren Kontrollattribute und Layoutberechnungen aber „zufällig“ sehr ähnlich sind.

Tabelle 4.1 enthält einige weitere interessante Angaben zu den Muster-Varianten. Das musterübergreifende Layoutkonzept wurde bereits bei der Beschreibung der Varianten erwähnt. Die oben erwähnten drei Varianten des Listen-Musters unterscheiden sich z.B. im musterübergreifenden Layoutkonzept, während das Layout der `Connection`-Varianten lediglich lokal unterschiedlich ist, also nicht die Schnittstelle beeinflusst. Die Klassifikation nach Layout-Konzept kann sehr hilfreich sein um zu entscheiden, welche Muster-Variante in einem bestimmten Kontext anzuwenden ist.

Ein anderes wichtiges Merkmal einer Muster-Implementierung ist der Grad des automatischen Layouts. Je automatischer das Layout ist, desto einfacher

und schneller können die entsprechenden Repräsentationen geändert werden, da der Zeitbedarf für aufwändige Layoutkorrekturen entfällt. Andererseits sind gewisse Layoutfreiheiten des Benutzers sinnvoll, um damit zusätzliche informelle Informationen, so genannte *sekundäre Annotationen* [19] in die grafische Darstellung integrieren zu können. Die Tabelle 4.1 listet die Layout-Angaben auf, die der Sprachbenutzer selbst festlegen kann, um so das Layout zu beeinflussen. Bei den Layoutvorgaben kann prinzipiell zwischen diskreten Vorgaben (z.B. Reihenfolgen, Zeilenumbrüche) und kontinuierlichen Vorgaben (Positionen und Größen) unterschieden werden. All diese benutzerdefinierten Layout-Vorgaben können Semantik tragen, brauchen es aber nicht. Bei Listen hat z.B. die Reihenfolge der Listenelemente häufig eine Semantik, in Sequenzdiagrammen dient die lineare Anordnung der Lebenslinien jedoch lediglich dem Layout. Anders herum ist die genaue Position von Elementen häufig bedeutungslos, in Generischen Zeichnungen (siehe Abschnitt 4.4) sind die Koordinaten demgegenüber wichtig für die Übersetzung.

Schließlich sind in der Tabelle die Anzahl der Kontrollattribute des Konfigurationsmodells angegeben. Hieran lässt sich abschätzen, wie flexibel die jeweilige Muster-Variante parametrisiert werden kann. Natürlich charakterisieren die Zahlen nicht nur die grundsätzliche Flexibilität einer Muster-Variante, sondern zum Teil auch ihre „Ausbaustufe“. Wichtige, häufig verwendete Muster-Varianten wie z.B. `SimpleList` sind im Hinblick auf ihre Parametrisierbarkeit besser ausgebaut als unwichtigere Varianten.

4.2.6 Kapselung musterspezifischer Eigenschaften

Ein großer Vorteil visueller Muster ist die Möglichkeit, musterspezifische Darstellungs-, Layout- und Interaktionsmechanismen zu kapseln. In Handimplementierungen visueller Sprachen gibt es teilweise speziell entworfene Layout- und Interaktionsmechanismen für bestimmte Strukturen, die einen besonders einfachen Umgang mit diesen gestatten. Durch visuelle Muster kann dieses Expertenwissen wiederverwendbar und einfach anwendbar gemacht werden, was nachfolgend an drei ausgewählten Beispielen demonstriert werden soll.

Die Muster-Variante *Set* Diese Ausprägung des Mengen-Musters visualisiert eine Objektmenge, deren Objekte innerhalb einer gegebenen grafischen

Region frei positioniert werden können. Die vom Benutzer gewählten Positionen der Mengenelemente und die gewählte Größe der Menge werden persistent in der editierbaren Struktur gespeichert.

Neue Mengenelemente können per *Drag-and-Drop* eingefügt werden. Während der Einfügeoperation ist ein Positionierungsanzeiger zu sehen, der zeigt, an welchen Koordinaten das Element bei aktueller Mausposition eingefügt würde. Zusätzlich wird die Menge selbst hervorgehoben, um den strukturellen Einfügekontext sichtbar zu machen. Vorhandene Mengenelemente können mit dem gleichen Mechanismus verschoben werden. Technisch repräsentiert die gesamte Fläche der Menge eine Einfügestelle im Sinne von Abschnitt 4.1.4.

Die Größe der Menge kann frei durch den Benutzer bestimmt werden. Wenn er die Menge selektiert, erscheint hierzu ein Ziehpunkt in der unteren rechten Ecke. Durch Ziehen mit der Maus wird das persistente Attribut, das die Größe der Darstellungsregion speichert, geändert. Die Muster-Implementierung stellt automatisch sicher, dass alle Mengenelemente vollständig innerhalb der Mengenregion liegen. Notfalls wird die Mengenregion vergrößert, so dass alle Elemente hineinpassen.

Um die Benutzungseigenschaften weiter zu verbessern, besitzt die Muster-Variante zwei spezielle Mechanismen, die je nach Bedarf aktiviert oder deaktiviert werden können, nämlich ein Positionierungsgitter und eine automatische Nicht-Überlappungs-Korrektur.

Das Positionierungsgitter ist in Abbildung 4.11 durch die Gitterpunkte zu erkennen. Hierdurch wird die exakte Positionierung der Einzelelemente erleichtert. Bei eingeschaltetem Positionierungsgitter werden die Positionen der Objekte beim Einfügen und Verschieben an den Gitterpunkten ausgerichtet. Zur Interpretation von Abbildung 4.11 ist zu beachten, dass nur jeder zweite Gitterpunkt markiert ist.

Durch die Nicht-Überlappungs-Korrektur wird verhindert, dass sich zwei Einzelelemente überlappen. Diese Situation kann z.B. durch Verschieben oder durch Größenänderung eines der Elemente entstehen. Für den Fall, dass sich Elemente überlappen, werden die Koordinaten automatisch angepasst. Die Koordinatenänderungen werden so gering wie möglich gehalten, um den Benutzer nicht übermäßig zu verwirren.

Eine Nicht-Überlappungs-Korrektur ist auch im VL-Eli System (siehe Abschnitt 2.4) vorhanden. In VL-Eli wird dazu ein allgemeiner Constraintsolver

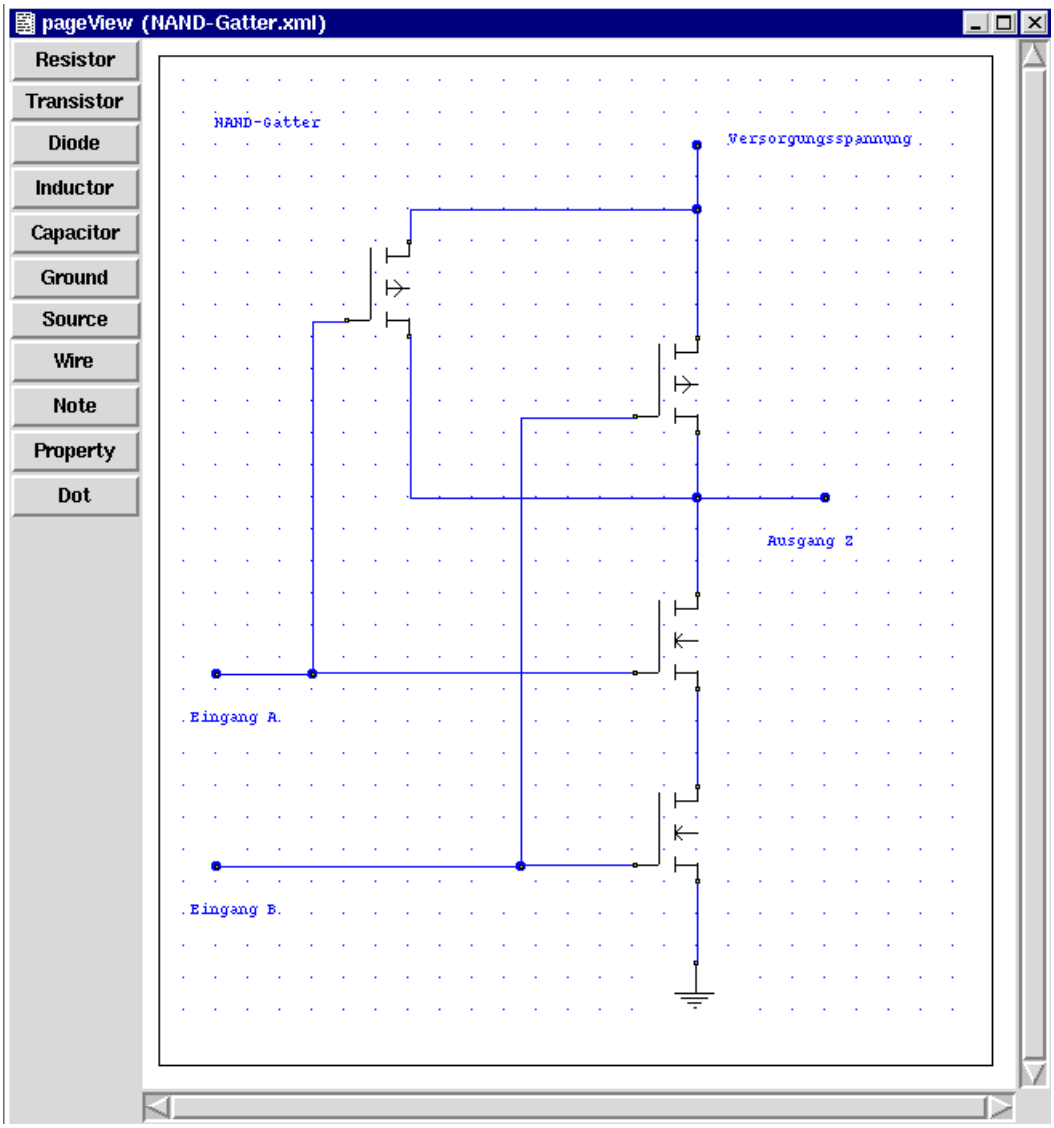


Abbildung 4.11: Beispiel für das Positionierungsgitter der Muster-Variante `Set`

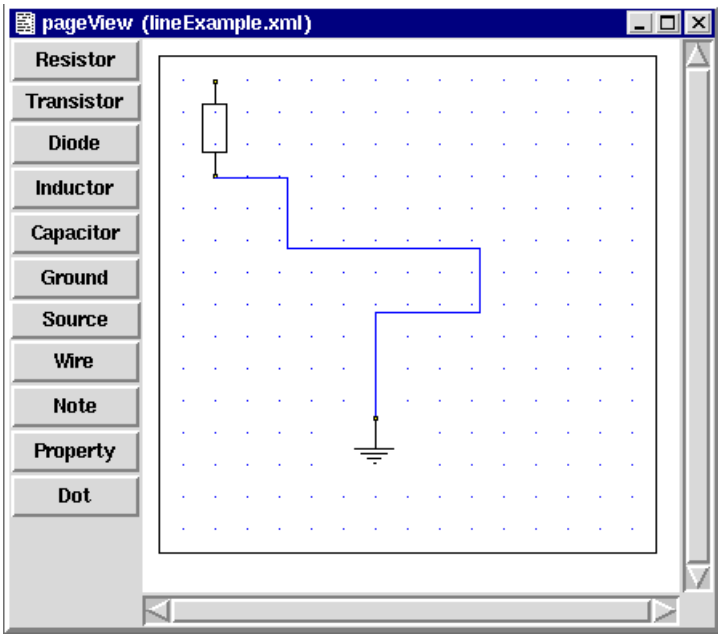


Abbildung 4.12: Orthogonale Linienführung mit benutzerspezifizierbarem Linienverlauf

verwendet. Problematisch ist, dass Constraintsolver für diese Problemklasse nicht besonders gut geeignet sind, da zur Beschreibung der Randbedingungen Disjunktionen verwendet werden müssen. Der in VL-Eli eingesetzte Constraintsolver Parcon [20] besitzt für diese Problemklasse ein exponentielles Laufzeitverhalten. Der Grund dafür ist, dass für eine optimale Lösung (im Sinne des Constraintsolvers) ein exponentiell großer Lösungsbaum durchsucht werden muss. In DEViL wurde daher ein handimplementierter, heuristischer Algorithmus verwendet. Im Vergleich zum Constraintsolver ist die Implementierung von Hand natürlich aufwändiger und liefert auch nicht unbedingt eine optimale Lösung. Wie in Kapitel 5 (Abschnitt 5.3.10) gezeigt wird, wird aber erst dadurch die Nicht-Überlappungs-Korrektur auch für größere Objektmengen praktikabel.

Die Muster-Variante *OrthoConnection* Diese Ausprägung des Linien-Musters zeichnet sich durch eine Linienführung mit orthogonal verlaufenden Liniensegmenten aus. Ein Beispiel für eine entsprechende Linienverbindung ist in Abbildung 4.12 dargestellt.

Eine Gemeinsamkeit mit anderen Varianten des Linien-Musters ist der mausbasierte Interaktionsmechanismus zur Erzeugung und Verbindung von Lini-

en. Noch unverbundene Linien lassen sich durch Selektion des entsprechenden Sprachkonstrukt-Knopfes und Auswahl einer Zielposition an einer beliebigen Stelle der Linienregion erstellen. Hierdurch wird ein noch unverbundenes Linien-Objekt angelegt. Um die Linienenden mit anderen Objekten zu verbinden, besitzen die Linien im selektierten Zustand Ziehpunkte an den Enden. Durch Ziehen mit der Maus wird automatisch der dem Mauszeiger nächstgelegene selektierbare Endpunkt hervorgehoben (vgl. auch Abschnitt 4.1.4). Des Weiteren können Sprachkonstrukt-Knöpfe können auch so definiert werden, dass sie die direkte Konstruktion verbundener Linen gestatten. Dann kann nach dem Drücken des Knopfes die Linienverbindung per *Drag-and-Drop* konstruiert werden.

Eine Besonderheit der hier vorgestellten Muster-Variante ist der Mechanismus, wie der Benutzer den Verlauf der Linie beeinflussen kann. Wenn die Linie selektiert ist, existiert dazu in der Mitte jedes Liniensegments ein Ziehpunkt, durch den der Verlauf geändert werden kann. Der Ziehpunkt ermöglicht es, das Liniensegment senkrecht zu dessen Verlauf zu verschieben, wodurch evtl. neue Linienabschnitte entstehen oder bestehende verschwinden können. Abbildung 4.12 zeigt einen Linienverlauf, der bereits auf diese Weise durch den Sprachbenutzer beeinflusst wurde.

Der Editor garantiert in jedem Fall, dass die Liniensegmente zusammenhängend und orthogonal bleiben. Auch wenn sich die Quell- und Zielpunkte verschieben, wird die Linienverbindung automatisch aktualisiert.

Zur Entwicklung dieser Muster-Variante wurden die Interaktionsmechanismen kommerzieller Editoren für elektronische Schaltpläne untersucht. Basierend auf dieser Untersuchung wurden die dort verwendeten Interaktions- und Layoutkonzepte als Variante des Linien-Musters gekapselt [47]. Unter Anwendung dieser Muster-Variante kann ein Editor für elektronische Schaltpläne spezifiziert werden, der dem Interaktionsmechanismus seines Vorbildes sehr nahe kommt.

Die Muster-Variante *Matrix* Diese Implementierung des Matrix-Musters zeigt, wie auch inhärent mehrdimensionale visuelle Darstellungsformen implementiert und durch maßgeschneiderte Layout- und Interaktionsmechanismen wirkungsvoll unterstützt werden können. Abbildung 4.13 zeigt eine Ausprägung der Muster-Variante. Die beiden dort dargestellten mathematischen Matrizen bestehen aus drei Zeilen und drei Spalten. In jeder Matrix-Zelle befindet sich ein mathematischer Ausdruck. Zur besseren Illustration

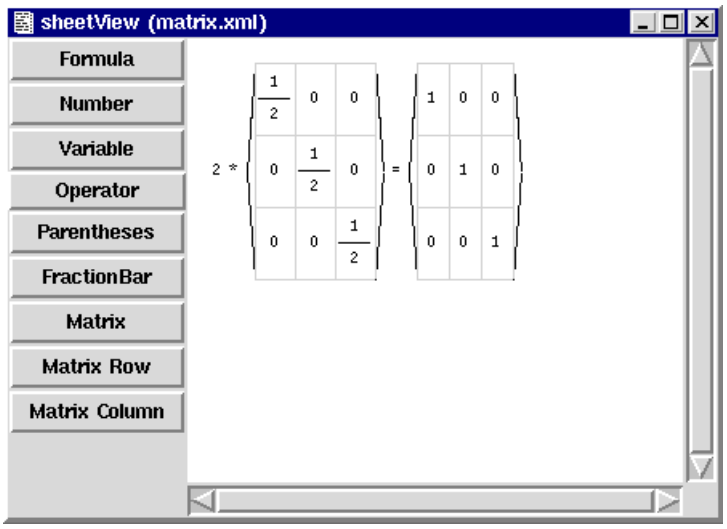


Abbildung 4.13: Beispiel für das Matrix-Muster

besitzt die Matrix Gitterlinien, die natürlich in mathematischen Formeln normalerweise nicht enthalten sind.

Das Layout von Matrizen wird basierend auf dem Platzbedarf der Zellen automatisch berechnet. Durch die Sprachkonstrukt-Knöpfe „Matrix Row“ und „Matrix Column“ können neue Zeilen bzw. Spalten eingefügt werden, wodurch automatisch eine entsprechende Anzahl neuer Zellen entsteht. Zum Einfügen muss lediglich eine gültige Einfügestelle ähnlich wie in Abbildung 4.6 auf Seite 140 gewählt werden. Zeilen und Spalten können auch mit der Maus selektiert und dann an eine andere Stelle verschoben oder gelöscht werden.

Interessant ist in diesem Fall die Struktur der zugrundeliegenden Syntax. Abbildung 4.14 zeigt die Modellierung der Matrix in DSSL-Notation. Wie zu erkennen ist, besteht eine Matrix aus einer Folge von Zeilen. Jede Zeile besteht wiederum aus einer Folge von Zellen. Aus den Reihenfolgen der Zeilen und Zellen ergibt sich die grafische Positionierung der Zellen.

Da auch eine Spalte ein visuelles Objekt ist, besitzt die Syntax auch eine Klasse für die Spalten, die jedoch keine Attribute hat. Jedoch besitzt jede Zelle eine Referenz auf die Spalte, zu der sie gehört. Durch diese Modellierung lassen sich Konsistenzüberprüfungen und automatische Struktur-Korrekturen mit den in Kapitel 3 eingeführten Pfadausdrücken einfach spezifizieren. So bleibt die Matrix-Struktur auch dann gültig, wenn sie nicht anhand der Matrix-Repräsentation, sondern anhand einer anderen Sicht geändert wird. Wird beispielsweise ein `MatrixColumn`-Objekt gelöscht, verschwinden damit auto-

```
CLASS Matrix INHERITS Symbol {
  rows: SUB MatrixRow*;
  columns: SUB MatrixColumn*;
}

CLASS MatrixRow {
  cells: SUB MatrixCell*;
}

CLASS MatrixColumn {
}

CLASS MatrixCell {
  columnRef: REF MatrixColumn;
  symbols: SUB Symbol*;
}
```

Abbildung 4.14: Editor-Syntax für Matrizen

matisch auch alle `MatrixCell`-Objekte, die zu dieser `MatrixColumn` gehören.

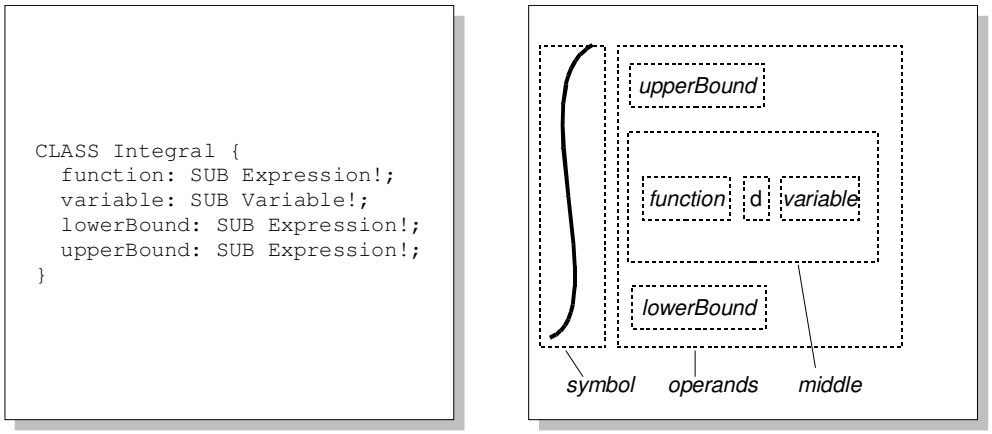
4.3 Anpassung der Grammatik-Abbildung

Zur Anwendung einer Muster-Variante ist es entscheidend, eine „passend“ strukturierte Repräsentations-Grammatik zu haben. Bereits in Kapitel 3 wurde dargelegt, dass man dazu evtl. eine von der semantischen Struktur abweichende editierbare Struktur definieren muss. Auch kann es manchmal notwendig sein, die Abbildung der editierbaren Struktur auf die Repräsentationsstruktur beeinflussen zu können.

Nachfolgend wird ein einfacher Spezifikationsmechanismus beschrieben, mit dem die Repräsentations-Grammatik den Erfordernissen der Sicht angepasst werden kann. Der Mechanismus ist im Vergleich zur Kopplung divergierender semantischer und editierbarer Strukturen sehr einfach und intuitiv. Wie in Kapitel 5 (Abschnitt 5.2.6) gezeigt wird, ist so eine Anpassung nur bei ca. fünf Prozent aller Sprachkonstrukt-Klassen erforderlich. In diesen Fällen ist sie aber unerlässlich.

4.3.1 Konzept der Grammatik-Abbildung

Das Konzept der Grammatik-Abbildung ist eng mit dem entsprechenden Konzept des GIGAS-Systems verwandt (siehe Abschnitt 2.5.2). Daher soll es



(a) Editor-Syntax (b) Schema der grafischen Darstellung

Abbildung 4.15: Struktur und Repräsentation eines Integral-Ausdrucks

nachfolgend anhand des gleichen Beispiels erklärt werden. Abbildung 4.15 zeigt eine Sprachkonstrukt-Klasse für ein mathematisches Integral und ein Schema der zu spezifizierenden grafischen Darstellung. Das Layout der Darstellung lässt sich durch geschachtelte Rechtecke ausdrücken. Die vollständige Darstellung `Integral` besteht aus zwei Teildarstellungen `symbol` und `operands`, die horizontal angeordnet und zentriert ausgerichtet werden sollen. `Operands` besteht aus den Unterblöcken `upperBound`, `middle` und `lowerBound`, die vertikal angeordnet und linksbündig ausgerichtet werden sollen. `middle` besteht wiederum aus den drei Blöcken `function`, `d` und `variable`.

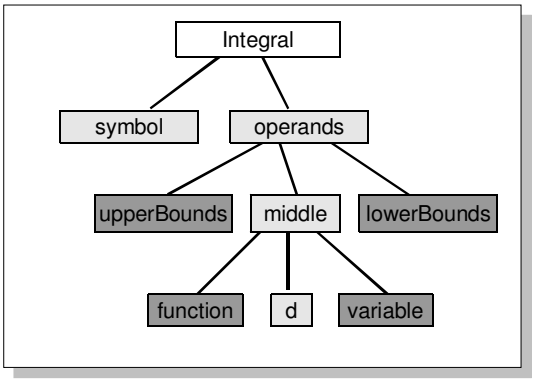
Um dieses Layout durch Attributberechnungen auszudrücken ist es zweckmäßig, eine Baumstruktur zugrunde zu legen, die der Schachtelungsstruktur der Rechtecke entspricht. Durch die Standard-Grammatikabbildung (siehe Abschnitt 4.1.2) ergibt sich aber eine Baumstruktur, bei der sich die Symbole `function`, `variable`, `lowerBound` und `upperBound` auf gleicher Ebene direkt unterhalb des Symbols für `Integral` befinden.

Diese Standard-Abbildung lässt sich anpassen, indem man die in Abbildung 4.16a gezeigte Spezifikation hinzufügt. Hierdurch ergibt sich die gewünschte Baumstruktur. Abbildung 4.16b ist eine grafische Illustration der in Abbildung 4.16a gezeigten Spezifikation, die den gleichen Informationsgehalt besitzt. Hier ist der Bezug zur Schachtelungsstruktur in Abbildung 4.15b klar erkennbar.

In der Spezifikation aus Abbildung 4.16a bzw. 4.16b lassen sich drei Arten von Knoten unterscheiden, nämlich Sprachkonstrukt-Knoten, Zwischenkno-

```
Integral(
  symbol(),
  operands(
    upperBounds,
    middle(function, d(), variable),
    lowerBounds
  )
)
```

(a) Grammatik-Abbildung



(b) Illustration von (a)

```
Integral ::= Integral_symbol Integral_operands
Integral_symbol ::= ε
Integral_operands ::= Integral_upperBounds Integral_middle Integral_lowerBounds
Integral_middle ::= Integral_function Integral_d Integral_variable
Integral_d ::= ε

Integral_upperBounds ::= Expression
Integral_lowerBounds ::= Expression
Integral_function ::= Expression
Integral_variable ::= Variable
```

(c) Generierte Repräsentations-Grammatik

Abbildung 4.16: Spezifikation der Grammatik-Abbildung für Integral-Ausdrücke

ten und Attribut-Knoten. In der grafischen Illustration sind Sprachkonstrukt-Knoten weiß, Zwischenknoten hellgrau und Attributknoten dunkelgrau gefärbt.

Die Wurzel des Baums ist generell der einzige Sprachkonstrukt-Knoten der Spezifikation. Durch ihn wird festgelegt, auf welche Sprachkonstrukt-Klasse sich die Spezifikation bezieht. Die dunkelgrau gefärbten Attribut-Knoten beziehen sich auf geerbte oder direkt definierte Attribute der Sprachkonstrukt-Klasse. Sie dürfen keine Unterknoten besitzen. Die Zwischenknoten können beliebig benannt werden und beschreiben neu einzufügende Zwischenproduktionen der Repräsentations-Grammatik.

In der textuellen Spezifikation werden Unterknoten in Klammern hinter ihrem Vater-Knoten notiert. Zwischenknoten und Attribut-Knoten lassen sich daran unterscheiden, dass Zwischenknoten ein (evtl. leeres) Klammerpaar folgt.

Formal werden durch die Spezifikation zwei Dinge beeinflusst, nämlich (1) die Repräsentations-Grammatik, in der jetzt evtl. zusätzliche Zwischenproduktionen auftreten und (2) die Abbildung der editierbaren Struktur auf die Repräsentationsstruktur, bei der dies berücksichtigt werden muss. Die aus der Spezifikation 4.16a bzw. 4.16b und der Editor-Grammatik aus 4.15a generierte Repräsentations-Grammatik ist in Abbildung 4.16c dargestellt. Man erkennt, dass die Baumstruktur direkt in Grammatikproduktionen umgesetzt ist. Die Spezifikation bestimmt auch die Reihenfolge der Symbole auf der rechten Seite der Produktionen.

Bezug zum Ansatz von Franchi-Zannettacci Ein ähnliches Konzept wird auch im GIGAS-System (siehe Abschnitt 2.5.2) verwendet. Dort wird die Editor-Syntax „abstrakte Grammatik“ und die Repräsentationsgrammatik „Layout-Grammatik“ genannt. Beide Grammatiken sind dort im Gegensatz zur DEViL-Variante konventionelle kontextfreie Grammatiken. In GIGAS ist die Spezifikation der Grammatik-Abbildung mit der Spezifikation der grafischen Darstellung verwoben (siehe Abbildung 2.16a auf Seite 56), wohingegen in DEViL beide Teile separat spezifiziert werden.

In GIGAS kann die Ausrichtung von Unterblöcken eines Blockes (z.B. horizontal oder vertikal nebeneinander, rechts/linksbündig oder zentriert) festgelegt werden. Zusätzlich können die sich daraus ergebenden Attributberechnungen überschrieben werden, um Details des Layouts zu beeinflussen. In

DEViL-Terminologie entspricht die Ausrichtungsangabe dem Kontrollattribut eines visuellen Musters. Das in GIGAS verwendete Konfigurationsmodell ähnelt der Muster-Variante `SimpleForm`. Der Unterschied zu DEViL ist, dass es in GIGAS nur wenige Muster gibt, die fest eingebaut sind, wohingegen es in DEViL auch Muster für andersartige Repräsentationen gibt und diese in erweiterbaren Bibliotheken gekapselt sind.

Zusammenfassend kann man sagen, dass DEViL den GIGAS-Ansatz verallgemeinert, indem das Kopplungskonzept zwischen editierbarer Struktur und Repräsentations-Struktur beibehalten wird, aber das fest eingebaute Konfigurationsmodell von GIGAS durch eine erweiterbare Bibliothek visueller Muster ersetzt wird.

4.3.2 Anwendungsbeispiele

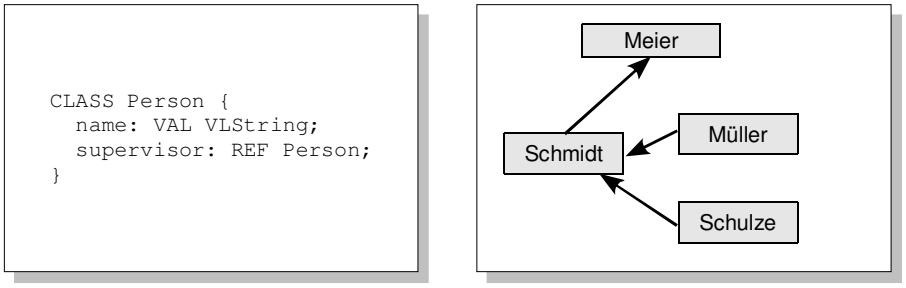
Wie bereits oben erwähnt muss die Grammatik-Abbildung nur in etwa fünf Prozent aller Fälle explizit spezifiziert werden (siehe Abschnitt 5.2.6). Das liegt daran, dass z.B. die oben diskutierte Integral-Repräsentation in DEViL normalerweise nicht durch die Muster-Variante `SimpleForm`, sondern durch die Muster-Variante `Form` spezifiziert wird und diese es gestattet, die Repräsentationen mittels so genannter Generischer Zeichnungen (siehe Abschnitt 4.4) visuell zu spezifizieren, so dass keine Feinstrukturierung der zugrundeliegenden Grammatik benötigt wird. Aber auch in DEViL gibt es Fälle, in denen die Anpassung der Repräsentations-Grammatik unerlässlich ist.

Es lassen sich vier Ziele der Anpassung unterscheiden, von denen das oben diskutierte Beispiel bereits drei beinhaltet.

1. Es sollen Zwischenknoten eingeführt werden, um Attributberechnungen zu erleichtern oder die Anwendung visueller Muster zu ermöglichen. Im Integral-Beispiel wurden z.B. die Zwischenknoten `operands` und `middle` eingeführt.

2. Es sollen neue Blattknoten[2] hinzugefügt werden, die als Anwendungskontexte bestimmter Musterrollen dienen. Im Integral-Beispiel wurde der Blattknoten `d` eingeführt.

[2]Im Sinne der Grammatik-Abbildung sind dies Zwischenknoten, die keine Unterknoten besitzen



(a) Editor-Syntax (b) Beispiel für die grafische Darstellung

Abbildung 4.17: Beispiel, bei dem ein REF-Attribut als Linie visualisiert wird

3. Die Reihenfolge der Symbole auf der rechten Seite einer Produktion soll angepasst werden, da diese Reihenfolge bei bestimmten Attributberechnungen und visuellen Mustern eine Rolle spielt. Im Integral-Beispiel ist die Reihenfolge `function d variable` wichtig für die konkrete Repräsentation.

4. Bestimmte Nichtterminale auf der rechten Seite einer Produktion sollen weggelassen werden, so dass der entsprechende Unterbaum „abgeschnitten“ wird.

Nachfolgend soll für all dieser Punkte eine weiteres Anwendungsbeispiel vorgestellt werden, um das Einsatzspektrum der spezifizierbaren Grammatik-Abbildung deutlich zu machen.

Hinzufügen von Zwischenknoten Die Muster-Variante `RefConnection` visualisiert ein Referenz-Attribut als Linienverbindung. Der Anfangspunkt der Linie ist das Objekt, zu dem diese Referenz gehört, der Endpunkt das Objekt, das referenziert wird. Um das Muster anwenden zu können, werden zwei separate Kontexte für den Linienbesitzer und die Linien selbst benötigt.

Die Anwendung des Musters soll an dem in Abbildung 4.17 gezeigten Beispiel verdeutlicht werden. In dem Beispiel soll das Sprachkonstrukt `Person` als Kästchen repräsentiert werden und das Attribut `supervisor` als Linienverbindung, die von diesem Kästchen ausgeht.

Die Umsetzung ist in Abbildung 4.18 illustriert. Die Abbildung visualisiert zwei Teilspezifikationen, nämlich eine Grammatik-Abbildung im Stil von Abbildung 4.16b auf Seite 166 und die hierauf basierende Zuordnung von

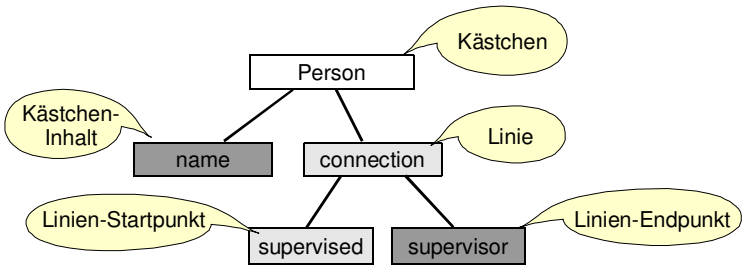


Abbildung 4.18: Umsetzung des Beispiels aus Abbildung 4.17

```
CLASS Person {
  name: VAL VLString;
  birthDate: VAL VLDate;
}
```

(a) Editor-Syntax

Name	Geb.Datum	Alter
Schmidt	01.08.1974	31
Schindler	09.02.1973	32

(b) Beispiel für die grafische Darstellung

Abbildung 4.19: Beispiel für eine Repräsentation mit berechnetem Inhalt

Symbol-Rollen. Die Symbol-Rollen sind lediglich informell durch Sprechblasen beschrieben, da die technischen Details der Musterrollen hier nicht von Bedeutung sind. Wichtig ist, dass der Zwischenknoten `connection` eingeführt wurde, der die Linie an sich repräsentiert. Die Zuordnung der Rollen „Kästchen“, „Linie“ und „Linienendpunkt“ an verschiedene Symbole ist spätestens dann notwendig, wenn einem dieser Kontexte weitere Eigenschaften, wie z.B. eine Beschriftung zugeordnet werden soll.

Hinzufügen von Blattknoten Manchmal werden Kontexte für Informationen benötigt, die nicht direkt Teil der editierbaren Struktur sind, sondern z.B. aus anderen Daten berechnet werden. Im Beispiel in Abbildung 4.19 sollen Personendaten als Tabelle dargestellt werden. Dabei soll sowohl das Geburtsdatum als auch das Alter angezeigt werden, wobei das Alter nicht Teil der editierbaren Struktur ist, da es aus dem Geburtsdatum berechnet werden kann. Um einen Kontext für die entsprechende Tabellenzelle zu definieren, muss der Repräsentations-Grammatik das zusätzliche Blattsymbol `age` hinzugefügt werden. Die zugehörige Grammatik-Abbildung und eine Illustration der Symbolrollen ist in Abbildung 4.20 dargestellt.

Festlegung der Symbol-Reihenfolge Beispielsweise bei der Anwendung der Muster-Variante `FlowForm` ergibt sich die Reihenfolge der visuel-

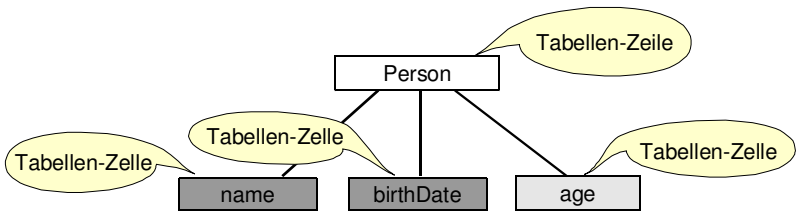


Abbildung 4.20: Umsetzung des Beispiels aus Abbildung 4.19

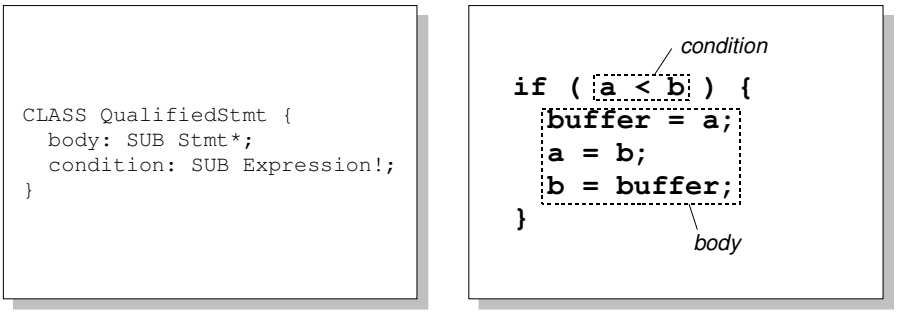


Abbildung 4.21: Beispiel für eine Repräsentation nach dem *Flow-Layout*

len Objekte aus der Reihenfolge der Symbole auf der rechten Seite der Repräsentationsgrammatik-Regeln. Im Beispiel aus Abbildung 4.21 soll eine bedingte Anweisungsfolge textuell dargestellt werden. Da zur Spezifikation dieser Repräsentation mit visuellen Mustern Grammatik-Symbole für Schlüsselwörter und Unterstrukturen in der richtigen Reihenfolge benötigt werden, ergibt sich die in Abbildung 4.22 dargestellte Grammatik-Abbildung.

Auslassen von Symbolen Durch die Grammatik-Abbildung kann auch ausgedrückt werden, dass bestimmte Unterstrukturen eines Symbols weggelassen werden sollen. Soll z.B. ein Strukturobjekt lediglich als beschriftetes Kästchen dargestellt werden, können Symbole auf der rechten Seite der Produktion weggelassen werden, die die Unterstruktur des Sprachkonstrukts modellieren. Dies kommt häufig in Sichten vor, die die Grobstruktur eines

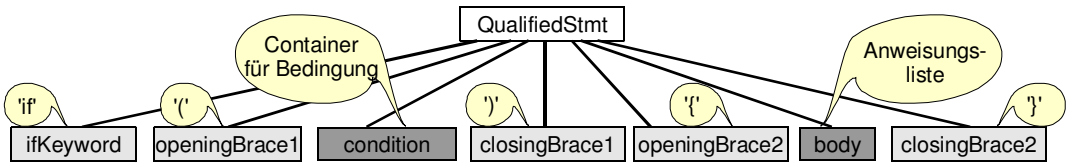


Abbildung 4.22: Umsetzung des Beispiels aus Abbildung 4.21

Programms repräsentieren und die durch Detailsichten ergänzt werden. Um eine entsprechende Repräsentations-Grammatik zu spezifizieren, lässt man einfach den entsprechenden Attribut-Knoten in der Grammatik-Abbildung weg. Dies bewirkt, dass in der Repräsentations-Struktur der gesamte zugehörige Teilbaum fehlt.

Das Ausblenden bestimmter Teilbäume ist nicht nur aus Effizienzgründen sinnvoll. Es kann auch vorkommen, dass in den nicht darzustellenden Unterstrukturen Symbole vorkommen, die in anderen Kontexten in der aktuellen Sicht dargestellt werden sollen und denen deshalb Attributberechnungen oder Muster-Rollen zugeordnet wurden. Würde der entsprechende Teilbaum nicht ausgeblendet, wäre die Attributierung in diesen Fällen formal unvollständig, was sich durch das Ausblenden der unerwünschten Teilbäume elegant umgehen lässt.

4.4 Generische Zeichnungen

Generische Zeichnungen wurden bereits informell durch das Beispiel in Abbildung 4.4 auf Seite 136 eingeführt. Grob gesagt lassen sich durch Generische Zeichnungen die Darstellungsdetails bestimmter visueller Objekte spezifizieren.

Generische Zeichnungen haben bereits einen langen Entwicklungsprozess hinter sich. Wir haben sie ursprünglich im Rahmen unserer Diplomarbeit [54] konzipiert, aber dort nur prototypisch eine textuelle Spezifikationssprache dafür entwickelt. Durch eine Studienarbeit von Schwindt [58] wurde erstmals ein visueller Strukturedítor für Generische Zeichnungen erstellt. Im Laufe der Zeit haben sich Generische Zeichnungen zu einer zentralen Spezifikationsmethode in DEViL entwickelt und es wurden viele Verbesserungen und Erweiterungen der ursprünglichen Spezifikationssprache vorgenommen. Durch die Umsetzung vieler Beispiele konnten wir zeigen, dass Generische Zeichnungen für ein breites Spektrum visueller Sprachen einsetzbar sind. Darüber hinaus werden Generische Zeichnungen von Nutzern als sehr leicht anwendbar klassifiziert (siehe Abschnitt 5.2.6).

Generische Zeichnungen sind zunächst ein abstraktes Konzept. Im abstrakten Sinne besteht eine Generische Zeichnung aus vier Bestandteilen:

- Einer Menge so genannter Container, in die bei der Instanziierung der Zeichnung beliebige Inhalte eingesetzt werden können.

- Einer Spezifikation der grafischen Verzierungen, die neben den Container-Inhalten das Aussehen des spezifizierten Sprachkonstrukts bestimmen.
- Einer Spezifikation des Layoutverhaltens, das bestimmt, wie sich Instanzen der Generischen Zeichnung an ihre Umgebung und die Containerinhalte anpassen.
- Einer Menge formaler Parameter, durch die bestimmte Darstellungs- und Layouteigenschaften individuell bei der Instanziierung der Generischen Zeichnung festgelegt werden können.

Die Container und die formalen Parameter bilden zusammen die Schnittstelle der Generischen Zeichnung. Wenn zwei Generische Zeichnungen in Bezug auf ihre Container und formalen Parameter übereinstimmen, können sie an den Anwendungsstellen transparent ausgetauscht werden, selbst dann, wenn es sich um unterschiedliche Implementierungsvarianten handelt.

In DEViL gibt es zwei Implementierungsvarianten für Generische Zeichnungen, nämlich Generische Verktorgrafik-Zeichnungen und Generische Kachel-Zeichnungen.

4.4.1 Generische Vektorgrafik-Zeichnungen

Abbildung 4.23 zeigt eine Generische Vektorgrafik-Zeichnung, die das Aussehen der Fallunterscheidung in Nassi-Shneiderman Diagrammen beschreibt. Die Abbildung 4.5 auf Seite 136 zeigt eine Instanz dieses Sprachkonstrukts.

Die Container `condition`, `trueBranch` und `falseBranch` definieren die grafischen Positionen, an denen die Bedingung und die beiden Zweige der Fallunterscheidung dargestellt werden. Durch Vektorgrafik-Primitive wie Linien, Rechtecke und Textzeichen werden Verzierungen und damit das Aussehen des Sprachkonstrukts genau beschrieben. Durch so genannte Dehnungsintervalle, die sich am unteren und rechten Rand der Zeichnung befinden, wird die Reaktion der Instanzen auf Layoutänderungen spezifiziert. Vergrößert sich z.B. der Inhalt des Containers `falseBranch`, wird der rechte untere Teil der Zeichnung gedehnt, um dem höheren Platzbedarf gerecht zu werden. Der linke und der obere Teil der Zeichnung bliebe nach der hier gezeigten Spezifikation unberührt.

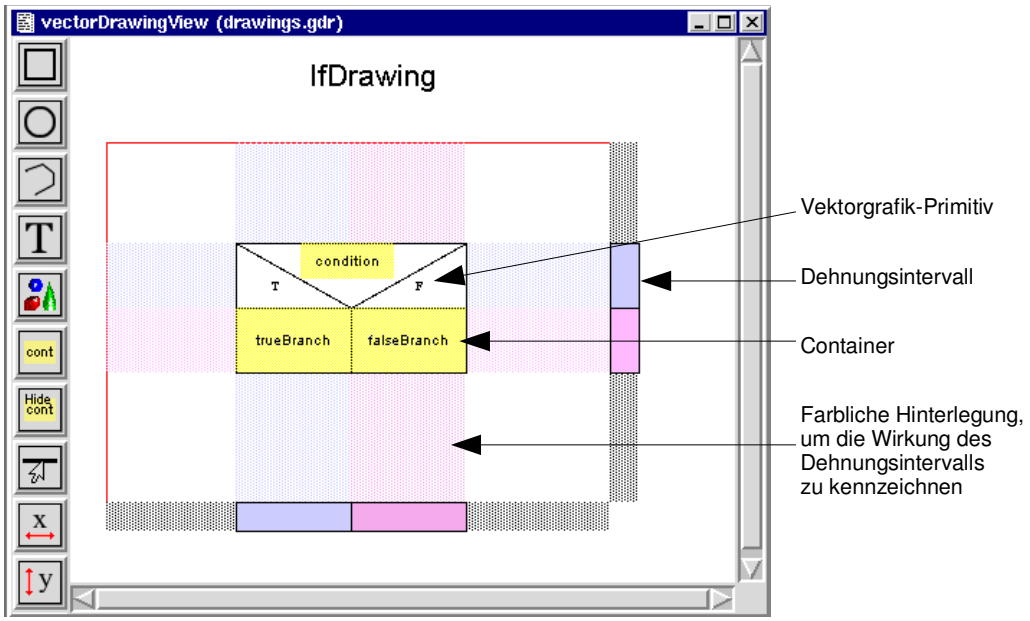


Abbildung 4.23: Beispiel für eine Generische Vektorgrafik-Zeichnung

Abbildung 4.24 zeigt einige weitere Anwendungsbeispiele für Generische Vektorgrafik-Zeichnungen. Die Zeichnung in 4.24a spezifiziert einen PNP-Transistor, der als Bauelement in elektronischen Schaltungen dient. Die Container kennzeichnen die drei Anschlussstellen des Transistors. Abbildung 4.24b spezifiziert einen XOR-Superstate in Zustandsdiagrammen. Die Container nehmen den Namen und den Inhalt des XOR-Superstates auf. Die Abbildungen 4.24c und 4.24d spezifizieren Stellen in Petri-Netzen. Abbildung 4.24c versinnbildlicht die Situation, in der eine Stelle genau drei Marken enthält. Dieses Beispiel zeigt, dass Generische Zeichnungen auch dazu „missbraucht“ werden können, um Sinnbilder ohne Unterstruktur zu spezifizieren. Da eine entsprechende Darstellung nur bei einer kleinen Markenanzahl sinnvoll ist, zeigt die Abbildung 4.24d die Spezifikation einer Stelle mit einem Container, der die Dezimalkodierung der Markenanzahl aufnehmen kann. Schließlich zeigt Abbildung 4.24e das Sinnbild für Benutzerrollen in Use-Case Diagrammen. Der Container nimmt hier den Namen der entsprechenden Rolle auf.

Nach diesem Überblick über das Prinzip und die Anwendungen Generischer Zeichnungen sollen die Sprachelemente jetzt im Detail behandelt werden.

Container Im Fall der Generischen Vektor-Zeichnungen kann die Größe und Position der Container beliebig gewählt werden. Jedem Con-

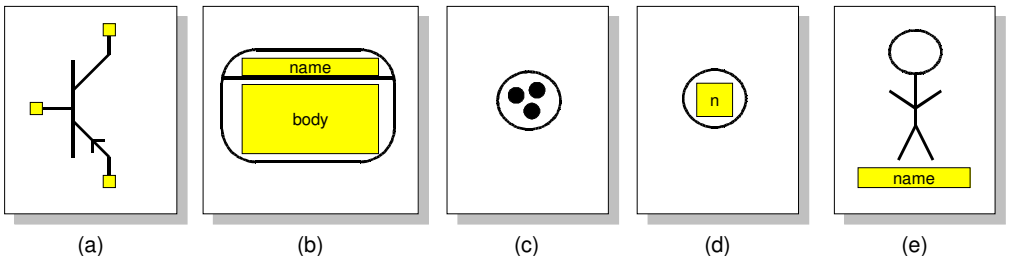


Abbildung 4.24: Anwendungsbeispiele für Generische Vektorgrafik-Zeichnungen

tainer muss ein für die Zeichnung eindeutiger Name zugeordnet werden. Die Container-Namen werden bei der Instanziierung der Generischen Zeichnung referenziert, um die Container mit Inhalt zu füllen. Im Beispiel in Abbildung 4.4 auf Seite 136 wird der Container-Inhalt der Zeichnung `IfStmtDrawing` durch die Knoten `IfStmt_condition`, `IfStmt_trueBranch` und `IfStmt_falseBranch` repräsentiert. Dort wird über das Attribut `formElementName` der Bezug zum Container hergestellt.

Manchmal kommt es vor, dass abhängig von bestimmten Laufzeitbedingungen verschiedene Generische Zeichnungen zur Darstellung des gleichen Konstrukts verwendet werden sollen. Beispielsweise ist es denkbar, dass die Bedingung einer Fallunterscheidung nicht angezeigt werden soll, wenn sie zu lang ist. Dies kann man erreichen, wenn man eine zweite Generische Zeichnung für Fallunterscheidungen anlegt, in der der Container `condition` nicht vorkommt. Um nach außen die gleiche Schnittstelle bereitzustellen, kann man diesen Container als „versteckt“ deklarieren. Die hieran zugewiesenen Inhalte werden einfach ignoriert, nach außen tragen versteckte Container aber zur Schnittstelle bei. Auf diese Weise lassen sich basierend auf Laufzeitbedingungen Repräsentationsdetails ändern oder Informationen ausblenden.

Verzierungen Durch Vektorgrafik-Primitive wie Rechtecke, Kreise, Linien, Zeichenketten oder Pixelgrafik-Bilder werden die Details der Darstellung festgelegt. All diese Elemente lassen sich frei positionieren und besitzen spezifische Darstellungsattribute wie Füll- oder Rahmenfarbe, Liniendicke, Strichelung, Füllmuster oder Schriftart. Diese können in entsprechenden Dialogsichten wie in Abbildung 4.25 gewählt werden. In diesem Punkt haben Generische Vektorzeichnungen große Ähnlichkeit mit Vektorgrafik-Programmen. Genau wie in solchen Programmen lässt sich auch die Darstellungsreihenfol-

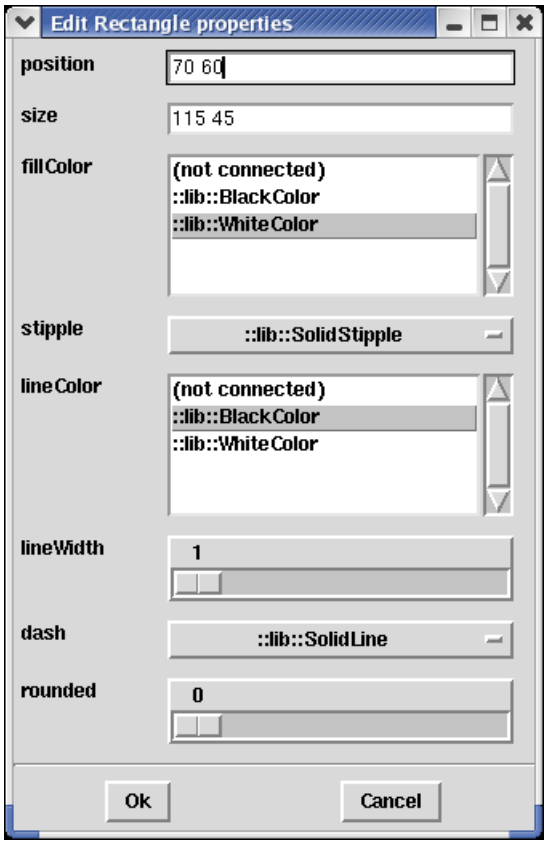


Abbildung 4.25: Dialogsicht zur Festlegung von Darstellungsdetails eines Rechtecks

ge der Vektorgrafik-Primitive festlegen, die bestimmt, welches Element von welchem anderen überdeckt wird.

Layoutverhalten Instanzen einer Generischen Vektorgrafik-Zeichnung müssen sich an den Platzbedarf von Container-Inhalten und an das Platzangebot ihrer Umgebung anpassen können. Solche Anpassungen lassen sich mathematisch als Koordinatentransformation auffassen. Durch die so genannten *Dehnungsintervalle* lässt sich recht intuitiv spezifizieren, welche Transformationen erlaubt sind.

In der grafischen Spezifikation werden Dehnungsintervalle durch flache Rechtecke repräsentiert, die sich auf den beiden gemusterten Streifen am unteren und rechten Rand der Generischen Zeichnung befinden. In Abbildung 4.23 sind je zwei horizontale und zwei vertikale Dehnungsintervalle enthalten, die jeweils den Bereich 0..50% Prozent und 50..100% der entsprechenden Dimension überdecken. Die Wirkung der Dehnungsintervalle auf die

Grafikprimitive wird visualisiert, indem der durch das Dehnungsintervalle überdeckte Teil farblich hinterlegt wird. Im Allgemeinen brauchen die Dehnungsintervalle einer Dimension nicht den vollständigen Koordinatenbereich zu überdecken. Sie dürfen sich allerdings nicht überlappen.

Dehnungsintervalle markieren Teilintervalle der jeweiligen Raumdimension, die zur Anpassung des Layouts linear skaliert werden dürfen. Enthält eine Dimension mehrere Dehnungsintervalle, dürfen deren Skalierungsfaktoren unabhängig voneinander gewählt werden. Bildlich kann man sich die Wirkung von Dehnungsintervallen so vorstellen, als ob die Grafikprimitive und Container auf Gummi gedruckt sind. Durch Ziehen an den Rändern der Dehnungsintervalle kann der Zwischenraum linear gedehnt werden. Gibt es mehrere Dehnungsintervalle in einer Dimension, können diese durch unterschiedlichen Kraftaufwand auch verschieden stark gedehnt werden. Da es in Abbildung 4.23 zwei separate horizontale Dehnungsbereiche gibt, ist es dort z.B. möglich, dass sich nach der Dehnung die Steigung der beiden schrägen Linien unterscheidet. Gäbe es stattdessen nur ein Dehnungsintervall über die gesamte Breite der Zeichnung, könnte sie nur insgesamt linear gedehnt werden. In diesem Fall wäre auch die Steigung der schrägen Linien nach der Dehnung garantiert gleich.

Falls ein bestimmter Ordinatenbereich nicht von einem Dehnungsintervall überdeckt wird, dürfen sich die Abstände darin liegender Punkte nicht ändern. Würde z.B. das linke horizontale Dehnungsintervall in Abbildung 4.23 entfernt, wäre die Breite der linken Hälfte der Fallunterscheidung fest. Allerdings wäre dann unklar, wie auf einen höheren Platzbedarf des Containers `trueBranch` zu reagieren ist. Aus diesem Grund muss in Generischen Verktor-Zeichnungen jeder Container in jeder Dimension von mindestens einem Dehnungsintervall geschnitten werden, damit er bei Bedarf gedehnt werden kann. Diese Bedingung wird automatisch zur Übersetzungszeit der Spezifikation geprüft.

Die für das Layout notwendigen Transformationen werden basierend auf den spezifizierten Dehnungsintervallen automatisch berechnet. Um das Layoutverhalten flexibler steuern zu können, lassen sich den Dehnungsintervallen Prioritäten zuordnen. Gibt es mehrere Möglichkeiten, den erforderlichen Platzbedarf zu erfüllen, werden Dehnungsintervalle mit höherer Priorität bevorzugt. Ferner kann jedem Dehnungsintervall eine Schrittweite zugeordnet werden, so dass die Dehnung nur in ganzzahligen Vielfachen dieser Schrittweite erfolgt.

Schließlich kann auch die Bearbeitungsreihenfolge der Container während

des Layoutprozesses geändert werden, um das Dehnungsverhalten zu beeinflussen. Während sich die oben dargestellten Einflussmöglichkeiten deklarativ beschreiben lassen, ist es hierzu allerdings erforderlich, den Layoutalgorithmus zu verstehen.

Zu Beginn des Layoutprozesses ergibt sich die Startgröße der Zeichnungs-Instanz aus deren Spezifikation. Die Container werden nacheinander in der spezifizierten Reihenfolge bearbeitet, um deren Größe den Erfordernissen anzupassen. Wenn die aktuelle Containergröße für den aufzunehmenden Inhalt nicht ausreicht, wird die Menge der Dehnungsintervalle berechnet, die den Container in der entsprechenden Dimension überlappen. Aus dieser Menge werden nur die Dehnungsintervalle mit maximaler Priorität betrachtet. All diese Dehnungsintervalle werden gleichzeitig so lange gedehnt, bis der Container seine Zielgröße erreicht hat. Am Ende des Durchlaufs haben demnach alle Container eine ausreichende Größe. Bevor die Instanz der Generischen Zeichnung tatsächlich auf dem Bildschirm dargestellt wird, wird deren Größe ggf. noch an den verfügbaren Darstellungsbereich angepasst, falls dieser größer als benötigt ist. Dazu werden alle Dehnungsintervalle mit maximaler Priorität so lange gleichzeitig gedehnt, bis die Instanz der Zeichnung ihre Zielgröße erreicht hat.

Wie zu erkennen ist, kann die Bearbeitungsreihenfolge der Container das Ergebnis des Dehnungsprozesses beeinflussen. Im Beispiel aus Abbildung 4.23 kann z.B. die Bearbeitung von `condition` vor der Bearbeitung von `trueBranch` und `falseBranch` dazu führen, dass das Ergebnis breiter als nötig wird. Müssen nämlich z.B. `condition` und `trueBranch` gedehnt werden, würden bei der Dehnung von `condition` `trueBranch` und `falseBranch` gleichmäßig gedehnt. Würde demgegenüber zuerst `trueBranch` gedehnt, wäre dadurch evtl. auch bereits die Platzanforderung für `condition` erfüllt, so dass `falseBranch` überhaupt nicht gedehnt würde. Eine Faustregel ist also, dass Container, die von nur wenigen Dehnungsbereichen geschnitten werden, früh bearbeitet werden sollten.

Die Größenänderung der Generischen Zeichnung ist nicht das einzige Layoutverhalten, das spezifiziert werden kann. Gleichfalls gibt es mehrere Möglichkeiten, wie der Inhalt eines Containers ausgerichtet werden kann. Die Ausrichtung des Containerinhalts ist relevant, wenn dieser kleiner als der aufnehmende Container ist. Für diesen Fall gibt es äquivalent zur Parametrisierung der Muster-Variante `SimpleList` (siehe Abschnitt 4.2.2) die Ausrichtungsoptionen `Left`, `Right`, `Center` und `Scale` für die horizontale Ausrichtung und `Top`, `Bottom`, `Center` und `Scale` für die vertikale. Auch diese

Festlegungen können in einer Dialogsicht des entsprechenden Containers gewählt werden und werden durch die Ausrichtung der Container-Beschriftung visualisiert.

Formale Parameter Normalerweise sind Farben, Füll- und Linienmuster der Vektorgrafik-Elemente festgelegt, denn sie bestimmen das Erscheinungsbild und damit die Wiedererkennbarkeit des spezifizierten Sprachkonstrukts. In manchen Fällen können diese Darstellungsattribute aber auch bestimmte Eigenschaften des Sprachkonstrukts, wie Attributwerte oder Laufzeitinformationen visualisieren. Für diesen Fall können in Generischen Vektorgrafik-Zeichnungen formale Parameter für diese Darstellungsattribute verwendet werden. Formale Parameter werden im Kopfbereich der Generischen Zeichnung deklariert. In den Dialogsichten der Vektorgrafik-Elemente können dann nicht nur konstante Farben, Füll- oder Linienmuster, sondern auch die formalen Parameter als Werte ausgewählt werden. Im Kontext der Parameter-Deklaration muss ein Standardwert für diesen Parameter gewählt werden, der zur Darstellung der Primitive in der Spezifikation benutzt wird.

Eine alternative Sicht Ein großer Vorteil der vorgestellten Spezifikationssprache ist die Tatsache, dass die Abbildungsdistanz zwischen Spezifikation und gewünschtem Ergebnis sehr klein ist. Anders ausgedrückt: Der Sprachentwickler kann direkt aufzeichnen, wie das Sprachkonstrukt aussehen soll. Diese Eigenschaft einer Spezifikationssprache wird *closeness of mapping* [19] genannt.

Es gibt allerdings Situationen, in denen es sinnvoll ist, die Spezifikation einer Generischen Zeichnung aus einem anderen Blickwinkel zu betrachten. Hierzu implementiert der Editor für Generische Zeichnungen einen zweiten, zur Hauptsicht orthogonalen Sichttyp. Abbildung 4.26 zeigt eine entsprechende Sicht am Beispiel der bedingten Anweisung in Nassi-Shneiderman Diagrammen. Diese Sicht hebt genau die Eigenschaften hervor, die in der Primärsicht in Abbildung 4.23 nur implizit bzw. überhaupt nicht dargestellt werden, nämlich Attributwerte, Koordinaten und Reihenfolgen der Elemente. In der sekundären Sicht ist es z.B. besonders einfach, die Darstellungsreihenfolge der Vektorgrafik-Primitive zu ändern, indem das entsprechende Objekt einfach an eine andere Stelle der Liste geschoben wird. Da beide Sichten die gleiche Struktur visualisieren, wirken sich Änderungen unmittelbar auf die jeweils andere Sicht aus.

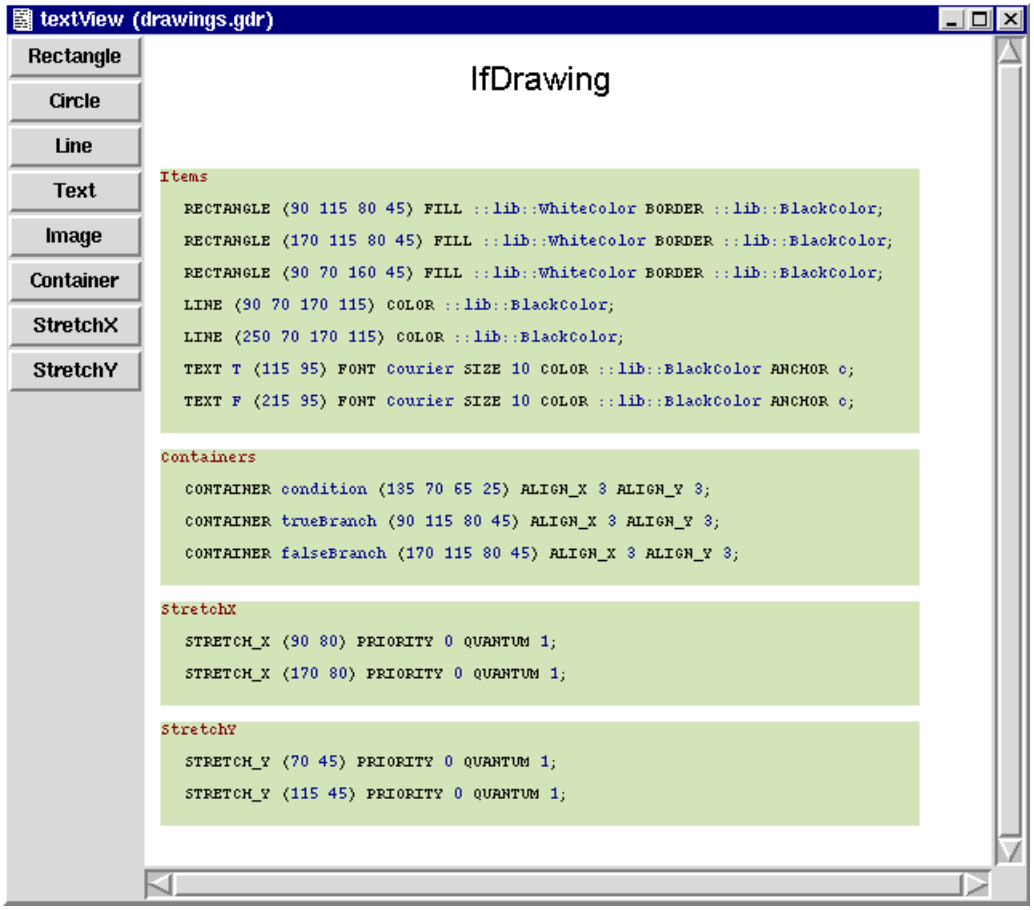


Abbildung 4.26: Textuelle Sicht auf die Generische Zeichnung in Abbildung 4.23

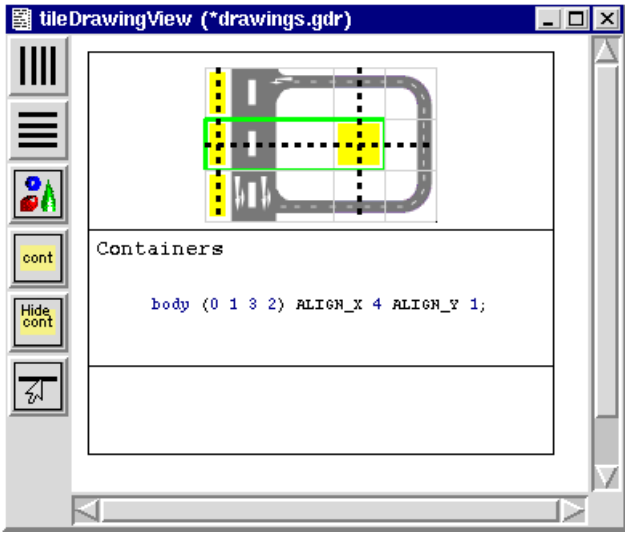


Abbildung 4.27: Beispiel für eine Generische Kachel-Zeichnung

4.4.2 Generische Kachel-Zeichnungen

Generische Kachel-Zeichnungen nutzen einen anderen Spezifikationsmechanismus, um konkrete Darstellungen zu spezifizieren. Hierzu werden die Verzierungen eines Programmkonstrukts mittels Pixelgrafik-Kacheln spezifiziert, die vorher separat mit einem externen Grafikprogramm erstellt wurden. Basierend auf dieser Technik können grafische Repräsentationen spezifiziert werden, die mit Generischen Vektorgrafik-Zeichnungen nicht realisierbar sind.

Abbildung 4.27 zeigt die Spezifikation einer Schleife der visuellen Programmiersprache *Streets* (siehe Abschnitt 2.2.4). Instanzen dieses Konstrukts sind in Abbildung 2.5 auf Seite 24 zu sehen. Die Grundlage dieser Spezifikationstechnik ist eine Matrix von Pixelgrafik-Kacheln. Die Anzahl der Zeilen und Spalten kann beliebig variiert werden. Container überdecken rechteckige Teilbereiche der Matrix und werden durch ein grünes Rechteck dargestellt. Unterhalb der Matrix werden die Container zusätzlich mit Namen und Koordinaten aufgeführt.

Auch Generische Kachel-Zeichnungen können sich vergrößern, um zusätzlichem Platzbedarf gerecht zu werden. Die Transformation erfolgt hier jedoch anders als bei der Vektorgrafik-Variante. Die durch gepunktete Linien dargestellten so genannten Dehnungslinien geben an, dass die entsprechende Zeile bzw. Spalte repliziert werden kann. Auf diese Weise lassen sich die Container, die von den Dehnungslinien geschnitten werden, vergrößern. Der Dehnungs-

algorithmus sowie die Möglichkeiten, das Layout durch Prioritäten, Reihenfolgen und Ausrichtungsangaben zu beeinflussen, entsprechen weitgehend denen der Generischen Vektorgrafik-Zeichnungen.

Durch Generische Kachel-Zeichnungen können auch sehr komplexe grafische Metaphern wie Straßenverläufe sehr effizient umgesetzt werden. Bei Generischen Vektorgrafik-Zeichnungen könnte es hier Effizienzprobleme geben, da sehr viele grafische Primitive benötigt würden. Allerdings ist der Aufwand zur Erstellung Generischer Kachel-Zeichnungen wesentlich größer, da zunächst alle Kacheln erstellt werden müssen. Zudem können bestimmte grafische Inhalte, wie schräg verlaufende Linien nicht umgesetzt werden. Welche Art von Generischen Zeichnungen eingesetzt wird, hängt auch stark vom Charakter der Sprache ab. Da sehr viele Sprachen geometrische Grundelemente besitzen, werden Kachel-Zeichnungen im Vergleich zu Vektor-Zeichnungen relativ selten eingesetzt.

4.5 Spezifikation textueller Teilrepräsentationen

Textuelle Teilrepräsentationen kommen auch in visuellen Sprachen vor. In UML-Zustandsdiagrammen sind z.B. die Beschriftungen von Transitionen rein textuell. Sie beschreiben Vorbedingungen, Auslöser und Aktionen der jeweiligen Transition, haben also durchaus eine recht komplexere Syntax. Ein anderes Beispiel sind die Constraints in UML, die ebenfalls textuell sind und eine komplexe Syntax haben. Häufig werden in visuellen Sprachen vor allem große Strukturen und Zusammenhänge visuell dargestellt, während Details wie mathematische Formeln oder Fragmente mit Anweisungssequenzen textuell dargestellt werden.

Die Rolle textueller Anteile in visuellen Sprachen ist durchaus umstritten. Es gibt die Auffassung, dass der Übergang zu textuellen Repräsentationen einen Bruch im Sprachkonzept darstellt und daher vermieden werden sollte [36, S.186], während es nach einer anderen Meinung eine gute Entscheidung ist, Details textuell darzustellen, damit die visuelle Repräsentation umso deutlicher das Wesentliche herausstellen kann. Ich teile die letztgenannte Auffassung und halte es für wichtig, dass ein Generator zur Implementierung visueller Strukturediteren auch textuelle Teilrepräsentationen adäquat unterstützt. Da textuelle Repräsentationen ein „einfacher Spezialfall“ visueller Repräsentationen sind, sollten sie zudem sehr einfach spezifizierbar sein.

```
CLASS QualifiedStmt {
  body: SUB Stmt*;
  condition: SUB Expression!;
}
```

(a) Editor-Syntax

```
QualifiedStmt: "if"
  "(" condition ")" "{" body "}".
```

(b) Spezifikation der textuellen Repräsentation

Abbildung 4.28: Beispiel für die Spezifikation einer textuellen Repräsentation

In DEViL lassen sich textuelle Teilrepräsentationen durch eine kleine Spezialsprache namens SLTR (*specification language for textual representations*) spezifizieren. Die in SLTR spezifizierten Teile können mit Spezifikationen für visuelle Repräsentationen zu einer vollständigen Sicht kombiniert werden. Eigentlich ist SLTR nur eine abkürzende Schreibweise, da SLTR-Spezifikationen auf sehr einfache Weise auf Grammatik-Abbildungen und die Anwendung visueller Muster abgebildet werden. Einerseits wird so sichergestellt, dass visuelle und textuelle Anteile problemlos kombiniert werden können. Andererseits lassen sich so komplexe Layoutstrategien für textuelle Teilrepräsentationen gut wartbar in visuellen Mustern kapseln, anstatt sie fest im Generator zu implementieren.

Die Sprache SLTR soll anhand des Beispiels in Abbildung 4.21 auf Seite 171 erklärt werden. Dort soll eine bedingte Anweisung in textueller Form repräsentiert werden. Abbildung 4.28b zeigt die entsprechende SLTR-Spezifikation. Um eine vollständige textuelle Repräsentation der editierbaren Struktur zu spezifizieren, müsste lediglich zu jeder Sprachkonstrukt-Klasse eine entsprechende SLTR-Klausel angegeben werden.

Wie zu erkennen, hat eine SLTR-Klausel Ähnlichkeit mit einer Grammatik-Produktion, die zum *Unparsing* verwendet wird. Auf der linken Seite steht der Name der zugehörigen Sprachkonstrukt-Klasse und auf der rechten Seite eine Sequenz von Symbolen, wobei jedes Symbol entweder ein Schlüsselwort oder ein Attributname ist. Unter Schlüsselwort ist hier jedes konstante Grundsymbol zu verstehen, also z.B. auch Operatoren, Klammern oder andere Sonderzeichen. Die Attributnamen beziehen sich auf geerbte oder direkt definierte Attribute der entsprechende Sprachkonstrukt-Klasse. Die Sequenz in Abbildung 4.28b besteht also aus den Schlüsselwörtern „if“, öffnende Klammer, schließende Klammer, öffnende geschweifte Klammer, schließende geschweifte Klammer und den Attributnamen `condition` und `body`.

Die Semantik der Sprache ist einfach: Die Repräsentation eines Sprachkonstrukts ergibt sich, indem die Repräsentationen der Symbole auf der rechten

Tabelle 4.2: Abbildung von Attributtypen auf Berechnungsrollen bei der Übersetzung von SLTR-Spezifikationen

Attributart	Rolle	Rolle der Unterelemente
VAL	VPFlowValPrimitive	-
REF	VPFlowRefPrimitive	-
SUB*	VPFlowList	VPFlowListElement
SUB?	VPFlowContainer	VPFlowContainerElement
SUB!	VPFlowContainer	VPFlowContainerElement

Seite konkateniert werden. Die Repräsentation von Attributen ergibt sich dabei aus dem Typ des Attributs. Bei VAL-Attributen wird der Inhalt des Attributs und bei REF-Attributen der Name des referenzierten Objekts dargestellt. Bei SUB-Attributen ergibt sich die Repräsentation aus anderen SLTR-Klauseln oder Muster-Anwendungen.

Abbildung 4.29 zeigt am Beispiel der Spezifikation aus Abbildung 4.28b, wie SLTR-Klauseln auf Grammatik-Abbildungen und Anwendungen von Muster-Varianten abgebildet werden. Die dort gezeigten Spezifikationen werden automatisch generiert. Aus der Grammatik-Abbildung in 4.29a geht hervor, dass jedes Symbol einer SLTR-Klausel auf ein Symbol der Repräsentations-Grammatik abgebildet wird. In der Teilspezifikation 4.29b werden diesen Symbolen Berechnungsrollen zugeordnet. Das Symbol für die Sprachkonstrukt-Klasse erbt die Rolle `VPFlowForm`. Die Muster-Varianten für die Untersymbole ergeben sich aus deren Typ. In Tabelle 4.2 ist aufgelistet, auf welche Muster-Varianten die einzelnen Attributarten abgebildet werden.

Es gibt einige SLTR-Sprachkonstrukte, um die Details der Repräsentation zu variieren. Um beispielsweise das Trenn-Symbol zwischen Listenelementen zu spezifizieren, kann dieses in eckigen Klammern hinter einem SUB*-Attribut angegeben werden. Im Allgemeinen werden durch solche Zusatzangaben die Kontrollattribute der jeweiligen Berechnungsrollen beeinflusst.

Einsatzspektrum Die Sprache SLTR soll in erster Linie zeigen, wie sich textuelle Teilrepräsentation in das hier vorgestellte Spezifikationskonzept integrieren lassen. Für einfache Textrepräsentationen hat sie sich bereits als sehr nützlich erwiesen, da die Spezifikation im Gegensatz zur Anwendung visueller Muster nochmals erheblich vereinfacht wird (vgl. Abbildung 4.28 und Ab-

```
QualifiedStmt(keyword1(), keyword2(), condition, keyword3(),
  keyword4(), body, keyword5());
```

(a) Grammatik-Abbildung

```
SYMBOL rootView_QualifiedStmt
  INHERITS VPFlowForm
END;

SYMBOL rootView_QualifiedStmt_keyword1
  INHERITS VPFlowFormElement, VPFlowKeywordPrimitive
  COMPUTE SYNT.text="if";
END;

SYMBOL rootView_QualifiedStmt_keyword2
  INHERITS VPFlowFormElement, VPFlowKeywordPrimitive
  COMPUTE SYNT.text="(";
END;

SYMBOL rootView_QualifiedStmt_condition
  INHERITS VPFlowFormElement, VPFlowContainer
END;

SYMBOL rootView_Expression
  INHERITS VPFlowContainerElement
END;

SYMBOL rootView_QualifiedStmt_keyword3
  INHERITS VPFlowFormElement, VPFlowKeywordPrimitive
  COMPUTE SYNT.text=")";
END;

SYMBOL rootView_QualifiedStmt_keyword4
  INHERITS VPFlowFormElement, VPFlowKeywordPrimitive
  COMPUTE SYNT.text="{";
END;

SYMBOL rootView_QualifiedStmt_body
   INHERITS VPFlowFormElement, VPFlowList
  COMPUTE SYNT.separatorList=vlList();
END;

SYMBOL rootView_Stmt
  INHERITS VPFlowListElement
END;

SYMBOL rootView_QualifiedStmt_keyword5
  INHERITS VPFlowFormElement, VPFlowKeywordPrimitive
  COMPUTE SYNT.text="}";
END;
```

(b) Anwendung visueller Muster

Abbildung 4.29: Umsetzung der SLTR-Spezifikation aus Abbildung 4.28

bildung 4.29). Um anspruchsvolle textuelle Repräsentationen wirkungsvoll zu unterstützen müsste allerdings die Funktionalität von SLTR und die Leistung der zugrundeliegenden Muster-Varianten noch weiterentwickelt werden. Wichtige Aspekte, die bis jetzt nur unzureichend berücksichtig wurden sind z.B. Strategien für die Bestimmung des Zeilenumbruchs und Strategien zur kontextbasierten Vereinfachung der Repräsentation. Unter den ersten Aspekt fällt z.B. der Wunsch, Zeilenumbrüche von der Länge der Zeilen abhängig machen zu können. Der Umbruch sollte dann an sinnvollen Stellen, z.B. nach einem Operator auf oberster Ebene stattfinden. Unter den zweiten Aspekt fällt z.B. das Weglassen von Klammern um einen Block, der nur eine Anweisung enthält, oder das Weglassen von Klammern in Ausdrücken, falls diese aufgrund der Präzedenzen und Assoziativitäten nicht notwendig sind. Solche Layoutentscheidungen sind notwendig, da die editierbare Struktur häufig ein Niveau besitzt, das von diesen Darstellungsdetails abstrahiert.

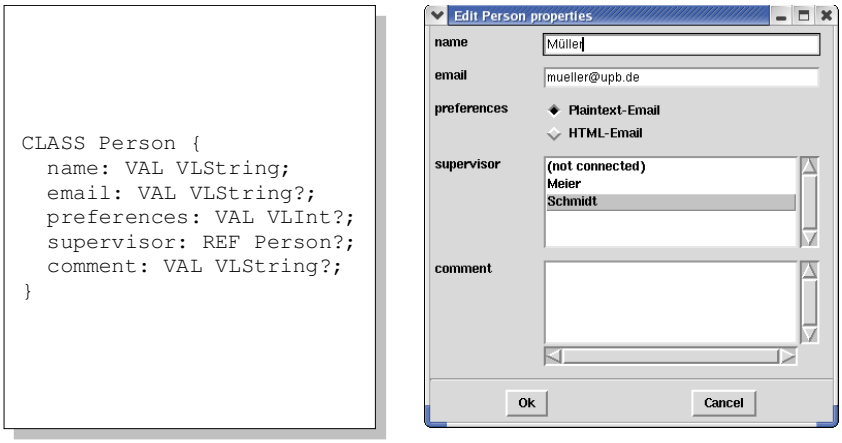
Natürlich müssen entsprechende Methoden nicht neu erfunden werden, denn im Bereich der automatischen Programmformatierung existiert ein großes Spektrum an Arbeiten und Systemen. Interessanterweise basieren viele fortgeschrittene Methoden auf kontextfreien Grammatiken [42]. Solche Lösungen könnten problemlos als Muster-Varianten gekapselt und in DEViL integriert werden.

4.6 Dialogsichten

Bis jetzt wurde die Spezifikation „normaler“, so genannter visueller Sichten betrachtet. In DEViL gibt es eine weitere Art benutzerspezifizierbarer Sichten, die so genannten Dialogsichten.

In Abbildung 4.30b ist eine typische Dialogsicht dargestellt, die zu der in Abbildung 4.30a definierten Sprachkonstrukt-Klasse gehört. Wie zu erkennen ist, sind Dialogsichten im Normalfall recht unspektakulär. Sie werden benötigt, um VAL- und REF-Attribute, in diesem Fall die Attribute `name`, `email`, `preferences`, `supervisor` und `comment` zu editieren. Prinzipiell könnten einige dieser Attribute auch in einer visuellen Sicht editiert werden. Bei Attributen wie `preferences` oder `supervisor` ist dies aber problematisch, denn visuelle Repräsentationen sind normalerweise platzoptimiert, so dass dort eine Auswahlliste nicht gut aufgehoben wäre.

Hieran zeigen sich bereits die grundsätzlichen Unterschiede der beiden Sichttypen. Visuelle Sichten sind darauf ausgelegt, Informationen übersichtlich



(a) Editor-Syntax (b) Beispiel für die grafische Darstellung

Abbildung 4.30: Beispiel für eine Dialog-Sicht

und kompakt zu repräsentieren und deren Zusammenhänge darzustellen. Im Gegensatz dazu sind Dialogsichten maßgeschneidert zum Eingeben und Ändern von Daten. Aus diesem Grund werden in der grafischen Darstellung besonders die Wertebereiche und Typen der Attribute sowie Interaktionselemente hervorgehoben. Wertebereiche werden z.B. durch Auswahllisten, Ankreuzfelder oder Schieberegler visualisiert, Interaktionselemente sind z.B. Knöpfe, Rollbalken oder die Griffe von Schiebereglern. Weiterhin besitzen Dialogsichten normalerweise ein klares Ordnungsschema, das die Tastaturnavigation erleichtert.

Diese Unterschiede begründen, warum visuellen Sichten und Dialogsichten separat betrachtet und unterstützt werden sollten. In DEViL äußert sich diese Unterscheidung z.B. dadurch, dass Dialogsichten im Gegensatz zu visuellen Sichten modal sind, d.h. während der Bearbeitung einer Dialogsicht keine anderen Operationen durchführbar sind. Der Vorteil ist, dass die aus grafischen Oberflächen vertrauten Dialog-Befehle „Ok“ und „Cancel“ beibehalten werden können. Mit „Ok“ werden die Änderungen übernommen, „Cancel“ verwirft sie.

Trotz der Unterschiede zu visuellen Sichten sind auch Dialogsichten *Sichten* im Sinne des Model-View Konzepts, denn auch hier wird der Informationsgehalt der editierbaren Struktur auf spezifische Weise visualisiert und es werden Änderungen daran ermöglicht. Tatsächlich müssen Änderungen, die in Dialogsichten vorgenommen wurden, noch vor Betätigung des „Ok“-Knopfes auf die editierbare Struktur angewendet werden, denn um die Gültigkeit von Eingaben zu überprüfen oder um den Zustand von Dialogfeldern

```
CLASS Person {
  name: VAL VLString EDITWITH "Entry";
  email: VAL VLString? EDITWITH "Entry";
  preferences: VAL VLInt EDITWITH "Radio -values {Plaintext-Email 0 HTML-Email 1}";
  supervisor: REF Person? EDITWITH "Listbox";
  comment: VAL VLString? EDITWITH "Text -width 30 -height 5";
}
```

Abbildung 4.31: Spezifikation der Dialog-Sicht aus Abbildung 4.30b

zu bestimmen, müssen im Allgemeinen komplexe Zusammenhänge berücksichtigt werden, so dass die vollständige editierbare Struktur betrachtet werden muss.

Dialogsichten gab es in einer einfachen Form bereits im VL-Eli System (siehe Abschnitt 2.4). Dort wurden sie allerdings nicht als Sicht betrachtet und ihnen wurde keine große Bedeutung beigemessen. Erst spät habe ich erkannt, dass Dialogsichten auch in visuellen Strukturediteren eine wichtige Rolle spielen und ihnen dementsprechend viel Beachtung geschenkt werden muss.

Im Folgenden gehe ich zunächst auf die so genannten Standard-Dialogsichten ein. Sie entsprechen dem Funktionsumfang, der bereits in VL-Eli vorhanden war und ermöglichen es, Dialogsichten mit sehr geringem Aufwand zu spezifizieren. Allerdings sind der Flexibilität hier Grenzen gesetzt. Als alternative Spezifikationsmethode für „anspruchsvolle“ Dialogsichten habe ich daher die so genannten Spezial-Dialogsichten entwickelt. Als Preis für die größere Flexibilität bezahlt man einen etwas höheren Spezifikationsaufwand.

4.6.1 Standard-Dialogsichten

Dialogsichten wie die in Abbildung 4.30 werden für fast jedes Sprachelement benötigt und müssen daher schnell und einfach spezifizierbar sein. In DEViL werden daher die erforderlichen Angaben direkt in die Editor-Syntax integriert. Abbildung 4.31 zeigt eine entsprechend erweiterte Editor-Syntax, aus der die in Abbildung 4.30b gezeigte Dialogsicht generiert werden kann.

Wie zu erkennen ist, kann jedem VAL- und REF-Attribut durch eine EDITWITH-Klausel eine Eingabekomponente zugeordnet werden, mit der man das entsprechende Attribut editieren kann. Im Normalfall reichen diese Angaben bereits aus, um eine Dialogsicht zu generieren.

DEViL besitzt eine umfangreiche Bibliothek von Eingabekomponenten, z.B. Eingabefelder für Texte, Schieberegler für Zahlenintervalle, Auswahllisten,

Ankreuzfelder, Editoren für Farben und Dateisystempfade usw. Bei Bedarf können aber auch neue Eingabekomponenten implementiert und eingebunden werden. Alle Eingabekomponenten können durch optionale Parameter individuell angepasst werden. Beispielsweise enthält in Abbildung 4.31 die EDITWITH-Klausel für das Attribut `comment` bestimmte Parameter, die die Höhe und Breite der Eingabekomponente festlegen.

Um die Spezifikation noch einfacher und flexibler zu machen, gibt es einige Zusatzmechanismen. Wenn keine EDITWITH-Klausel angegeben wird, wird basierend auf dem Typ des Attributs automatisch eine sinnvolle Eingabekomponente gewählt. Für `VLString` und `VLInt`-Attribute wird z.B. `Entry`, für `VLBoolean` `Option` und für REF-Attribute `Listbox` gewählt. Das hat den Vorteil, dass selbst aus einer „nackten“ Editor-Syntax bereits sinnvolle Dialogsichten generiert werden können. Des Weiteren besteht die Möglichkeit, die Reihenfolge der Attribute in der Dialogsicht explizit festzulegen. Ohne diese Angabe wird die Reihenfolge aus der Reihenfolge der Attribute in der Editor-Syntax abgeleitet. Die explizite Reihenfolgeangabe ist vor allem dann nützlich, wenn Attribute von Oberklassen geerbt werden.

Wie zu erkennen ist, werden im Allgemeinen zur Spezifikation der Dialogsichten nur sehr wenige Zusatzangaben benötigt. Streng genommen wird bei dieser Art der Spezifikation Struktur- und Sichtdefinition vermischt, d.h. eigentlich müssten die Dialogsichten separat von der Struktur spezifiziert werden. Das würde allerdings dazu führen, dass sich der Spezifikationsaufwand vervielfachen würde, denn schließlich muss der Bezug zu den Klassen und Attributen der Editor-Struktur hergestellt werden. Daher wurde in diesem Fall die Vermischung von Struktur- und Sichtdefinition in Kauf genommen.

4.6.2 Spezial-Dialogsichten

Mit den oben vorgestellten Standard-Dialogsichten lässt sich die überwiegende Mehrheit aller erforderlichen Dialogsichten einfach und schnell beschreiben. Ab und zu kommt es aber vor, dass besondere Dialogsichten realisiert werden sollen, die auf diese Weise nicht umgesetzt werden können. Daher habe ich einen alternativen, allgemeineren Spezifikationsmechanismus realisiert.

Anforderungen Es lassen sich folgende, in der Praxis sehr häufig vorkommende Anforderungen an komplexere Dialogsichten identifizieren.

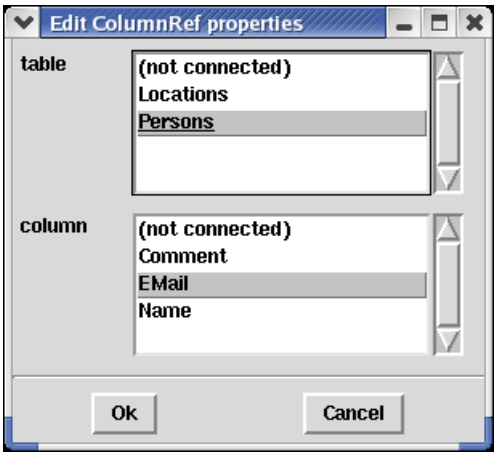


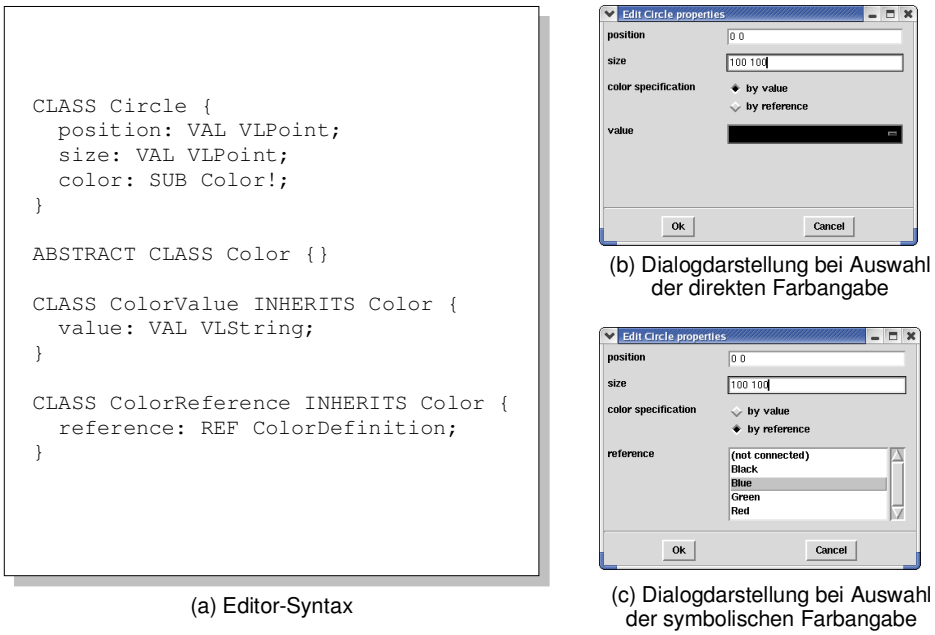
Abbildung 4.32: Mehrstufiger Auswahldialog für Spalten in Datenbanktabellen

- Einzelne Eingabekomponenten sollen basierend auf Laufzeitbedingungen deaktiviert oder ausgeblendet werden können.
- Die Konfiguration einer Eingabekomponente soll zur Laufzeit dynamisch geändert werden können.
- Es sollen nicht nur VAL- und REF-Attribute, sondern auch einfache Unterstrukturen dargestellt werden können.

Das dynamische Deaktivieren oder Ausblenden bestimmter Eingabekomponenten wird oft benötigt, wenn Attribute nicht unabhängig voneinander sind, sondern ein Attribut nur bei bestimmten Werten anderer Attribute sinnvoll ist. Ein Beispiel dafür liefert Abbildung 4.30. Dort ist die Angabe der Email-Adresse optional. Falls keine Email-Adresse angegeben wurde, ist auch das Attribut `preferences` bedeutungslos und sollte als Hinweis darauf im Dialog deaktiviert oder ausgeblendet werden.

Ein Beispiel dafür, dass die Konfiguration von Eingabekomponenten dynamisch änderbar sein muss, ist in Abbildung 4.32 dargestellt. Dort kann eine Spalte einer Datenbanktabelle mehrstufig selektiert werden, indem zuerst in der oberen Liste die gewünschte Tabelle ausgewählt und dann in der unteren Liste die gewünschte Spalte dieser Tabelle selektiert wird. Natürlich muss der Inhalt der unteren Liste dynamisch an die Auswahl in der oberen Liste angepasst werden.

Ein Beispiel für die Darstellung von Unterstrukturen eines Sprachelements findet sich in Abbildung 4.33. Dort geht es um das grafische Primitiv `Circle`,



(a) Editor-Syntax

(b) Dialogdarstellung bei Auswahl der direkten Farbangabe

(c) Dialogdarstellung bei Auswahl der symbolischen Farbangabe

Abbildung 4.33: Beispiel für eine Dialogsicht, die Unterstrukturen enthält

das eine Position, eine Größe und eine Farbe besitzt. Die Farbe kann auf zwei Arten angegeben werden, entweder direkt oder durch Bezug auf eine Farbdefinition an anderer Stelle. So eine Fallunterscheidung modelliert man am besten durch die abstrakte Oberklasse `Color` wie in Abbildung 4.33a gezeigt. Eine angemessene Dialogsicht für diese Struktur besitzt eine Auswahlkomponente, die über die Art der Farbangabe entscheidet. Diese Auswahlkomponente heißt in Abbildungen 4.33b und 4.33c „color specification“. Je nach Auswahl enthält das Attribut `color` ein Objekt der Klasse `ColorValue` oder `ColorReference`. Wie man an den Abbildungen 4.33b und 4.33c erkennen kann, sollte die Eingabekomponente für die Farbe von diesem Zustand abhängen. Im Fall der direkten Farbangabe wird ein Farbeditor und im Fall der indirekten Farbangabe eine Auswahlliste angezeigt.

Spezifikationskonzept Da Dialogsichten wie oben diskutiert prinzipiell mit visuellen Sichten vergleichbar sind, liegt es nahe, dafür auch den gleichen Spezifikationsmechanismus zu nutzen. In DEViL werden Spezial-Dialogsichten demnach durch attributierte Grammatiken und die Anwendung von Darstellungsrollen spezifiziert. Der Unterschied zur Spezifikation visueller Sichten ist lediglich, dass die Darstellungsmuster andersartig und in der Regel einfacher sind. In Dialogsichten gibt es z.B. keine frei positionier-

baren Elemente, keine anwendungsspezifischen grafischen Verzierungen und keine visuelle Repräsentation von Querbeziehungen. Zur Strukturierung von Dialogsichten werden im Allgemeinen höchstens Gruppierungen und Registerkarten verwendet.

Die Eingabekomponenten sind der weitaus komplexeste Teil von Dialogsichten. Glücklicherweise sind solche Komponenten bereits in Standard-Bibliotheken zur Implementierung grafischer Benutzungsschnittstellen enthalten und müssen nicht neu entwickelt werden. Auch sie kapseln weitentwickelte Darstellungs- und Interaktionsmechanismen, die erfreulicherweise bereits sehr einheitlich sind und damit zu einer schnell erlernbaren Benutzungsschnittstelle beitragen. Beispiele hierfür sind die Repräsentationen von Ankreuz- oder Selektionsfeldern oder die Interaktionsmechanismen zum Selektieren oder Löschen von Textteilen durch Tastaturkürzel oder Mausgesten.

In DEViL sind die Berechnungsrollen zur Spezifikation von Dialogsichten noch recht einfach gehalten. Sie genügen aber bereits den oben beschriebenen Anforderungen. Abbildung 4.34 zeigt die Spezifikation der in Abbildung 4.33 gezeigten Dialogsicht. Wie zu erkennen ist, ist diese Spezifikationsform nicht wesentlich komplizierter als im Fall der Standard-Dialogsichten, lediglich die Zuordnung zu den Strukturelementen verursacht Mehraufwand. Die Wahl der Eingabekomponenten und die entsprechende Parametrisierung erfolgt hier durch Attributberechnungen anstatt durch statische Definitionen. Da die Attributberechnungen nach strukturellen Änderungen neu ausgeführt werden, passt sich die Dialogsicht automatisch den jeweiligen Gegebenheiten an. Wie in Abbildung 4.34 zu erkennen ist, können auch Unterstrukturen Berechnungen zugeordnet werden, so dass die Dialogsichten nicht auf die Darstellung von VAL- und REF-Attributen beschränkt sind.

Alle Konzepte, die oben im Kontext der visuellen Sichten eingeführt wurden, lassen sich auf Dialogsichten übertragen. Durch eine benutzerdefinierte Grammatik-Abbildung können Dialogsichten strukturiert und so z.B. Gruppen oder Registerkarten realisiert werden. Durch einen ähnlichen Ansatz wie Generische Zeichnungen könnte das Layout von Dialogen visuell spezifiziert werden, wie es in manchen Systemen zur Entwicklung von Benutzungsschnittstellen bereits üblich ist.

```
SYMBOL dialogView_Circle
  INHERITS VPDialogRoot
END;

SYMBOL dialogView_Circle_position
  INHERITS VPDialogComponent
  COMPUTE SYNT.type = "Entry";
END;

SYMBOL dialogView_Circle_size
  INHERITS VPDialogComponent
  COMPUTE SYNT.type = "Entry";
END;

SYMBOL dialogView_Circle_color
  INHERITS VPDialogComponent
  COMPUTE SYNT.type = "Radio";
          SYNT.label = "color specification";
          SYNT.options = vlList("-values",
             vlList("by value", "ColorValue",
                    "by reference", "ColorReference"));
END;

SYMBOL dialogView_ColorValue_value
  INHERITS VPDialogComponent
  COMPUTE SYNT.type = "Color";
END;

SYMBOL dialogView_ColorReference_reference
  INHERITS VPDialogComponent
  COMPUTE SYNT.type = "Listbox";
END;
```

Abbildung 4.34: Spezifikation der Dialogsicht aus Abbildung 4.33

4.7 Verwandte Arbeiten

Dieses Kapitel kombiniert eine Vielzahl von Konzepten aus verschiedenen Bereichen. Nachfolgend sollen die wichtigsten Konzepte separat betrachtet und zu verwandten Arbeiten in Beziehung gesetzt werden.

Sicht-Spezifikation durch attributierte Grammatiken Attributierte Grammatiken werden bereits seit geraumer Zeit zur Spezifikation grafischer Repräsentationen eingesetzt. Beispiele hierfür sind GIGAS (siehe Abschnitt 2.5.2) und Loggie [4]. In Loggie wurde z.B. auch ein Graphlayout-System integriert. Die Modellierung von Querbeziehungen ist in den Systemen unterschiedlich gelöst. GIGAS beschränkt sich auf baumstrukturierte Sprachen. In Loggie werden spezielle Konstrukte, so genannte *Garlands* verwendet, um Querbeziehungen in Bäumen zu repräsentieren. DEViL sowie dessen Vorgänger VL-Eli benutzt ein externes Property-Modul zum Datentransport über Querbeziehungen hinweg.

In anderen Systemen wie DiaGen II (siehe Abschnitt 2.5.6) werden Attributauswerter auf Graphen verwendet. Ein Vorteil ist dort, dass der Umgang mit Querbeziehungen wesentlich vereinfacht wird. Allerdings basiert der in meiner Arbeit verfolgte Ansatz stark auf den Baumkanten der Struktur und den entsprechenden Zugriffsmechanismen entlang dieser Kanten. Systeme, die auf Graphen basieren, schenken den Baumkanten häufig wenig Bedeutung, so dass hier bestimmte Sprachkonstrukte „nachgerüstet“ werden müssten. Attributauswerter auf Graphen haben den weiteren Nachteil, dass sie prinzipbedingt weniger effizient sind und bestimmte Inkonsistenzen der Spezifikation erst zur Laufzeit erkennen können.

Visuelle Muster Die Idee, strukturellen Symbolen grafische Rollen zuzuordnen, findet sich in vielen Systemen. Im SRG-ASG-Ansatz (siehe Abschnitt 2.5.5) werden den Knoten und Kanten des so genannten *Spatial relationship graphs* geometrische Objekte und räumliche Relationen zugeordnet. In GenGEd [7] lassen sich den Symbolen ebenfalls grafische Objekte und räumliche Relationen zuordnen, wobei die verfügbaren Relationen in einer Bibliothek so genannter *high level constraints* gekapselt sind. Diese werden auf die Eingabesprache eines Constraintsolvers abgebildet. Auch die Internet-Technologie *Cascading Stylesheets* basiert auf der Zuordnung von Darstellungsrollen zu Symbolen der abstrakten Struktur. Hier werden den Symbolen der HTML-

oder XML-Struktur Darstellungsrollen wie Überschriften, Absätze oder Tabellen zugeordnet. All diesen Ansätzen ist gemein, dass die Darstellungsrollen keine Interaktionsmechanismen kapseln. Ferner sind durch die zugrundeliegenden, relativ „einheitlichen“ Kalküle die Darstellungs- und Layoutmöglichkeiten begrenzt. Mit Constraintsolver-basierten Ansätzen lassen sich z.B. kaum diskrete Layoutentscheidungen wie Zeilenumbrüche modellieren. Das Layoutmodell von *Cascading Stylesheets* ist auf Baumstrukturen und bestimmte Repräsentationsarten beschränkt.

Nardi und Zarmer [39] definieren so genannte *visuelle Formalismen*, die sich in Bibliotheken kapseln lassen und als Grundlage zur Implementierung interaktiver visueller Systeme dienen können. Visuelle Formalismen kapseln wie visuelle Muster die grafische Darstellung, das Layout und die Interaktionsmechanismen. Konzeptionell unterscheiden sich die beiden Ansätze dadurch, dass sich die visuellen Formalismen weit weniger flexibel kombinieren lassen. Auch methodisch unterscheiden sich die Ansätze, da Nardi und Zarmer visuelle Formalismen in C++ Bibliotheken kapseln, während visuelle Muster in DEViL auf Spezifikationsebene mittels attributierter Grammatiken realisiert sind.

Berechnungsrollen Die Methode, visuelle Muster-Varianten mittels Berechnungsrollen im Kalkül der attributierten Grammatiken zu kapseln, stammt aus dem Eli-System. Sie wurde dort zum ersten Mal angewendet, um Attributberechnungen zur Namensanalyse in textuellen Sprachen wiederverwendbar zu machen [33]. Die Methode wurde dann in VL-Eli zur Kapselung von Darstellungs- und Interaktionseigenschaften verwendet [54]. Aus Anwendersicht hat das Konzept Ähnlichkeit mit der Spezifikationsmethode im GIGAS-System. Dort stammen die Berechnungsrollen zwar nicht aus einer Bibliothek, sondern es gibt einen „fest eingebauten“ Satz an Darstellungsrollen. Diese lassen sich aber auf ähnliche Weise anwenden und parametrisieren.

Neu in DEViL ist das Kombinationskonzept der Muster und die Ausarbeitung der globalen Layoutstrategien. In VL-Eli wurden hier eine Ad-Hoc-Lösung benutzt. Ein generisches Ausdrucksmittel für die Schnittstellen in Rollendiagrammen fehlte. Ein Vorläufer der in dieser Arbeit vorgestellten Rollendiagramme ist in dieser Hinsicht das Modul zur Namensanalyse in Eli. Auch dort basiert die Schnittstellenbeschreibung auf geforderten und bereitgestellten Attributen. Lediglich die Gruppierung mehrerer Attribute zu Rollen-Schnittstellen und die grafische Beschreibungssprache ist neu. Die Semantik und das grundlegende Darstellungsprinzip der Rollendiagramme hat

UML-Klassendiagramme zum Vorbild. Durch spezialisierte Repräsentationen wichtiger struktureller Merkmale sind Rollendiagramme aber für den Sprachentwickler wesentlich übersichtlicher.

Spezialisierte Layoutberechnungen Die Grundlagen zu den Layoutberechnungen, die es zu kapseln und zu kombinieren galt, sind vielfältig. Die verschiedenen Methoden sind jeweils auf unterschiedliche grafische Darstellungsformen spezialisiert. So gibt es Systeme zum Graphlayout (z.B. [16]) und Linienrouting (z.B. [25]). Eine weitere wichtige Klasse von Layoutmethoden finden sich in Textsatzsystemen [34, 65] und in Formatierern für Programmtext [42]. Grafische Constraintsolver wie Parcon [20] eignen sich für eine größere Klasse visueller Darstellungen, sind aber auf lineare Gleichungs- und Ungleichungssysteme beschränkt. Der Beitrag von DEViL ist hier, all diese Methoden zu kombinieren.

Generische Zeichnungen Das Konzept der Generischen Zeichnungen habe ich zusammen mit Christian Schindler entwickelt [54]. Die Idee für die Generischen Vektorgrafik-Zeichnungen stammt aus dem VPE-System, das allerdings auf einer textbasierten Beschreibungssprache basiert (siehe Abschnitt 2.5.3). Ein ähnlicher Ansatz zur visuellen Spezifikation von Objekt-Repräsentationen findet sich im Meta-CASE Werkzeug MetaEdit (vgl. Abschnitt 2.5.4 und insbesondere Abbildung 2.19b auf Seite 59). Allerdings sind die dort eingesetzten Spezifikationsmittel im Gegensatz zu Generischen Zeichnungen wesentlich eingeschränkter. Weder das Layout noch der Inhalt der Container lassen sich so flexibel spezifizieren wie bei Generischen Zeichnungen.

Das Layoutkonzept der Generischen Vektorgrafik-Zeichnungen ist mit Bibliotheken zur Implementierung grafischer Editoren und Oberflächen vergleichbar. Das Framework *Unidraw* [63] z.B. benutzt das Paradigma der Federkräfte, um Repräsentationen mit flexiblem Layout zu beschreiben. Der Layoutalgorithmus selbst erinnert an den „Pack“ Layoutmanager von Tcl/Tk [43].

Die Idee der Generischen Kachel-Zeichnungen stammt aus der Handimplementierung der visuellen Programmiersprache *Streets* (siehe Abschnitt 2.2.4). Dort wurden die Pixelgrafik-Kacheln aus Effizienzgründen verwendet, denn die Erstellung komplexer Verzierungen wie Straßenverläufe mit Mittelstreifen, Abbiegungen usw. mit Vektorgrafik-Primitiven hätte zu Effizienzproblemen geführt. Auch im grafischen Programmiersystem *Agentsheets* [49] wer-

den Matrizen mit Grafikkacheln verwendet. Die Besonderheit der Generischen Kachel-Zeichnungen in DEViL ist, dass es lediglich als eins von mehreren Beschreibungsmodellen dient, um grafische Repräsentationen zu spezifizieren.

Grammatik-Anpassung Das in Abschnitt 4.3 vorgestellte Konzept zur Grammatik-Anpassung ist eng mit dem GIGAS System (siehe Abschnitt 2.5.2) verwandt. Im Unterschied zu DEViL wird die Grammatik-Abbildung und die grafische Repräsentation in GIGAS gemeinsam spezifiziert, d.h. diese beiden Spezifikationsaspekte sind miteinander verwoben. Des Weiteren werden in GIGAS beide Strukturebenen durch Grammatiken modelliert, während die Spezifikation der editierbaren Struktur in DEViL auf DSSL basiert, das über mehr Abstraktionskonzepte verfügt.

Auch im Eli-System [31] wird zwischen konkreter und abstrakter Grammatik unterschieden und der abstrakten Grammatik werden aus vergleichbaren Gründen häufig Kettenproduktionen hinzugefügt. In Eli gibt es zur Spezifikation der Kopplung von konkreter und abstrakter Struktur das so genannte *Maptool* [2]. Prinzipiell werden Fragmente der konkreten und abstrakten Grammatik separat voneinander angegeben. Die jeweiligen Entsprechungen und evtl. notwendige Ergänzungen werden dann durch das *Maptool* automatisch ermittelt. Im Vergleich zum *Maptool* ist der Spezifikationsmechanismus in DEViL eingeschränkter, aber auch einfacher zu benutzen.

Textuelle Bestandteile Ein besonderes Merkmal meines Ansatzes ist, dass textuelle Repräsentationen als Spezialfall visueller Repräsentationen verstanden werden, also mit den gleichen Spezifikationsmechanismen umgesetzt werden können. Im Gegensatz hierzu berücksichtigen viele andere Ansätze textuelle Teilrepräsentationen nicht oder benutzen dafür andere Spezifikationsmethoden.

In Systemen, die auch oder nur textuelle Repräsentationen unterstützen, finden sich Spezifikationsmechanismen, die SLTR recht ähnlich sind. In PGS (siehe Abschnitt 2.5.1) wird eine so genannte Format-Syntax verwendet, um die konkrete Repräsentation zu spezifizieren. Sie enthält spezielle Konstrukte für Zeilenumbrüche, Einrückungen und bedingte Formatierung. Bedingte Formatierungen können von der Existenz optionaler Unterstrukturen oder vom Typ der Unterstrukturen abhängen.

Dialogsichten In Systemen zur Implementierung visueller Strukturediteren wird kaum auf Dialogsichten als Darstellungsform für Teilstrukturen eingegangen. In VL-Eli lassen sich zwar die Eingabekomponenten von Dialogsichten spezifizieren, aber ansonsten ist die Struktur der generierten Dialogsichten fest. Im Gegensatz dazu lassen sich in DEViL Dialogsichten durch Zuordnung grafischer Rollen sehr flexibel und komfortabel spezifizieren und dabei auch dynamische Eigenschaften berücksichtigen.

In Systemen zur Entwicklung grafischer Benutzungsschnittstellen werden Dialoge bzw. Dialogsichten sehr gut unterstützt. Dort lassen sich Dialoge teilweise visuell erstellen. In der Datenbank-Software *Microsoft Access* lassen sich dialogbasierte Eingabeformulare für Tabellen grafisch spezifizieren und direkt mit den zugrundeliegenden strukturellen Elementen, d.h. den Tabellenfeldern, verbinden. Im Fall von *Microsoft Access* basiert die Struktur allerdings auf einem relationalen Datenbankschema, ist also im Vergleich zu DEViL wesentlich statischer.

Neue Beiträge Ein wichtiger methodischer Beitrag ist das Schnittstellen- und Kombinationskonzept visueller Muster. Im VL-Eli System gab es weder eine allgemeine Beschreibungssprache für die Musterschnittstellen noch eine ausreichende Zahl kombinierbarer musterübergreifender Layoutstrategien. Darüber hinaus habe ich durch die Implementierung einiger komplexer Muster-Varianten wie `Matrix`, `Tree` und `OrthoConnection` gezeigt, wie sich auch komplexere visuelle Konzepte wiederverwendbar kapseln lassen. Hieran wird besonders deutlich, dass in allen Fällen umfangreiches Expertenwissen wie Layoutmethoden und Interaktionsmethoden gekapselt wird.

Wichtige neu entwickelte Sprachen sind die beiden Varianten Generischer Zeichnungen. Wie in Kapitel 5 nachzulesen, haben sie sich bereits als sehr wirkungsvoll herausgestellt. Ich halte diese Spezifikationssprachen für einen wichtigen Schritt auf dem Weg, Sprachspezifikationsmethoden einfacher und auch für Nicht-Experten anwendbar zu machen.

Ein wichtiger Beitrag ist aber auch die Integration bereits existierender Konzepte und Methoden in ein neuartiges, übergeordnetes Gesamtkonzept. Durch die Integration der Einzelkonzepte „wiederverwendbare Berechnungsrollen“, „spezifizierbare Grammatik-Abbildungen“, „Dialogsichten“, „Generische Zeichnungen“ sowie unterschiedliche Layoutstrategien für visuelle und textuelle Repräsentationen ergibt sich eine sehr flexible und mächtige Spezifikationsmethode. Besonders wichtig finde ich, dass viele der Teilkon-

zepte optional sind, d.h. nicht berücksichtigt werden müssen, wenn sie für die jeweilige Anwendung nicht benötigt werden. Schließlich hoffe ich, durch die Integration der genannten Teilkonzepte zu einem besseren Verständnis der Zusammenhänge insgesamt beigetragen zu haben, so dass zukünftig noch mächtigere Werkzeugsysteme zur Sprachimplementierung entwickelt werden können.

5 Evaluation

Inhalt

Um praktikabel zu sein, muss eine Spezifikationsmethode einfach anwendbar und mächtig genug für praktisch relevante Aufgaben sein. Zusätzlich muss auch das aus der Spezifikation generierte Produkt bestimmten Qualitätsanforderungen entsprechen.

Da sowohl DEViL als auch die von DEViL generierten Editoren menschlichen Benutzern dienen sollen, können beide Ebenen auf der Grundlage allgemeiner Usability-Methoden evaluiert werden. Der Begriff Usability ist allgemein definiert als die Effektivität, Effizienz und Zufriedenheit, mit der bestimmte Benutzer bestimmte Ziele unter bestimmten Rahmenbedingungen erreichen (siehe Abschnitt 5.1.1).

Hohe Usability auf der Generator-Ebene bedeutet, dass das Spezifikationskonzept verständlich ist und dass sich Spezifikationen einfach und schnell erstellen lassen. Usability auf dieser Ebene umfasst weiterhin die softwaretechnischen Merkmale der Spezifikationssprachen, d.h. ob Spezifikationen änderungsfreundlich sind, ob sie für große Anwendungen skalieren oder ob sie Team-Entwicklung unterstützen.

Auf der Ebene der generierten Editoren misst die Usability, wie gut eine Sprachimplementierung benutzbar ist. Im Rahmen meiner Arbeit ist vor allem interessant, wie sehr die generierten Editoren die Benutzung einer Sprache erleichtern, d.h. ob sie intuitiv, effizient und praxistauglich sind. Natürlich hat auch die Sprache selbst einen Einfluss auf die Usability der Sprachimplementierung, aber von diesem Einfluss soll hier abstrahiert werden.

Die Usability auf Editor-Ebene ist von entscheidender Bedeutung für die Praxistauglichkeit eines Generators, denn visuelle Strukturediteren werden gerade deshalb eingesetzt, um eine Sprache einfacher und mit weniger Lernaufwand anwendbar zu machen. Dies wird besonders deutlich, wenn man an Endanwender als Zielgruppe der Spracheimplementierung denkt.

Nachfolgend werden zunächst die Grundlagen der Usability-Analyse behandelt. Neben zentralen Begriffen und Maßen in diesem Umfeld werden vor allem allgemein anwendbare Methoden zur Usability-Analyse vorgestellt. Des Weiteren wird auch auf Spezialisierungen im Bereichen der Programmier- und Spezifikationssprachen eingegangen.

In den Abschnitten zwei und drei wird dann die Usability auf der Ebene des Generators bzw. der generierten Editoren evaluiert. Beide Abschnitte sind gleich aufgebaut. Sie beginnen jeweils mit einer Diskussion der Zielsetzung, stellen dann die zur Evaluation durchgeführten Untersuchungen vor und beantworten schließlich die sich aus der Zielsetzung ergebenden Fragen.

5.1 Grundlagen der Usability

5.1.1 Der Begriff Usability

Usability bezeichnet die Nutzerfreundlichkeit oder Gebrauchstauglichkeit eines Gegenstands. Usability ist nach ISO-Norm 9241-11 folgendermaßen definiert:

> Die Usability eines Produktes ist das Ausmaß, in dem es von einem bestimmten Benutzer verwendet werden kann, um bestimmte Ziele in einem bestimmten Kontext effektiv, effizient und zufriedenstellend zu erreichen.

Die Effektivität gibt an, ob oder zu welchem Anteil ein vorgegebenes Ziel erreicht werden kann. Die Effizienz gibt an, wie viel Aufwand erforderlich ist, um ein Ziel zu erreichen. Die Zufriedenheit gibt das subjektive Empfinden des Benutzers bei der Umsetzung eines Ziels wider.

Bezogen auf das DEViL-System misst die Effektivität, ob oder mit welchen Einschränkungen eine gegebene visuelle Sprache umgesetzt werden kann. Die Effizienz misst, wie einfach und schnell ein Benutzer eine Spezifikation entwickeln, ändern, refaktorisieren oder verstehen kann.

Die Effizienz eines Generatorsystems lässt sich in verschiedene Faktoren zerlegen. Der Sprachentwurf bestimmt, wie effizient eine Aufgabe mit den verfügbaren Sprachmitteln gelöst werden kann. Die Entwicklungsumgebung (im Fall einer visuellen Spezifikationssprache) bestimmt, wie effizient die Spezifikation erstellt werden kann. Die Generator-Effizienz (im Sinne von Speicher- und Laufzeitkosten) schließlich bestimmt die Übersetzungseffizienz. Die Zufriedenheit des Benutzers wird z.B. dadurch bestimmt, ob das Spezifikationskonzept mit dem mentalen Modell des Benutzers übereinstimmt.

Wichtig ist, dass die Effektivität, Effizienz und Zufriedenheit immer vom Benutzer und dessen Ziel abhängt. Während ein Produkt z.B. für einen Anfänger sehr schwer benutzbar sein kann, könnte es für einen Nutzer mit Erfahrung im Problembereich einfacher benutzbar sein. Entsprechend kann ein Produkt evtl. gut zur Umsetzung eines bestimmten Ziels geeignet sein, während es zur Umsetzung eines anderen Ziels ungeeignet ist. Dies bedeutet für die Evaluation, dass sowohl die Benutzer als auch die Ziele klassifiziert werden müssen.

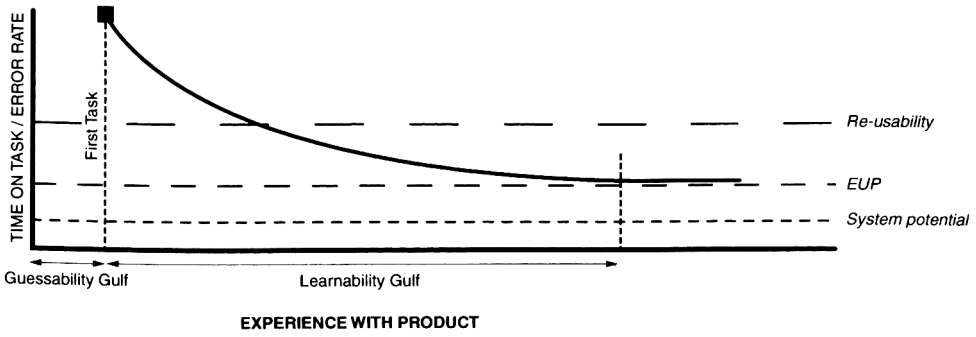


Abbildung 5.1: Einfluss der Erfahrung mit einem Produkt auf die Usability (aus [27])

Zur Klassifikation der Benutzer wurden bereits Benutzercharakteristiken identifiziert, die erwiesenermaßen starken Einfluss auf die Usability haben. Die wichtigsten sind (1) Erfahrung mit dem zu untersuchenden Produkt, (2) Erfahrung mit dem Problembereich, (3) kultureller Hintergrund, (4) Alter und Geschlecht sowie (5) Behinderungen [27].

Die Erfahrung mit dem zu untersuchenden Produkt nimmt dabei eine Sonderstellung ein, da sie sich im Verhältnis zu den anderen Charakteristiken schnell ändert. Mit steigender Erfahrung sinkt normalerweise die Fehlerrate bzw. der Zeitbedarf für eine Aufgabe. Dieser Zusammenhang ist in Abbildung 5.1 dargestellt. Die Kurve nähert sich schließlich einer Asymptote, der so genannten *experienced user performance* (EUP). Die EUP ist der Zeitbedarf, den ein erfahrener Benutzer zur Lösung einer Aufgabe benötigt.

Basierend auf der Lernkurve in Abbildung 5.1 lassen sich verschiedene Komponenten der Usability unterscheiden: Der Selbsterklärungsgrad (*guessability*) beschreibt die Kosten für einen Nutzer, der eine Funktion des Produkts zum ersten Mal benutzt. Erlernbarkeit (*learnability*) beschreibt die Kosten, um ein bestimmtes Kompetenzniveau bzgl. einer Produktfunktion zu erreichen. Die EUP (*experienced user performance*) charakterisiert die Kosten, die einem erfahrener Benutzer bei der Umsetzung einer Aufgabe entstehen. Das Systempotenzial (*system potential*) ist der theoretisch erreichbare Minimalwert der Kosten. Die *Re-usability* beschreibt schließlich die Kosten eines eingearbeiteten Benutzers, der das Produkt eine längere Zeit nicht mehr benutzt hat.

Zur Messung der Effektivität, Effizienz und Zufriedenheit wurden verschiedene Maße entwickelt [27]. Diese sind in Tabelle 5.1 zusammengefasst. Die Maße beziehen sich jeweils auf verschiedene Teile der Usability und sind für sich alleine recht einseitig. Daher sollten zur Evaluation eines Produkts stets

Tabelle 5.1: Allgemeine Maße zur Usability-Messung (aus [27])

Bezeichnung	Maß für...	Beschreibung
Task completion	Effektivität	Misst ob bzw. wie vollständig ein bestimmtes Ziel erreicht wird.
Quality of output	Effektivität	Misst, wie gut das Ergebnis bestimmten Qualitätsansprüchen genügt.
Derivation from the critical path	Effizienz	Misst, wie stark bei der Umsetzung eines Ziels von der effizientesten Vorgehensweise abgewichen wurde.
Error rate	Effizienz	Misst, wie viele Fehler bei der Umsetzung eines Ziels gemacht werden.
Time on task	Effizienz	Misst den Zeitbedarf zur Umsetzung eines Ziels.
Mental workload	Effizienz	Misst die mentale Beanspruchung bei der Umsetzung eines Ziels.
Qualitative attitude analysis	Zufriedenheit	Misst die Zufriedenheit des Benutzers auf qualitativem Niveau.
Quantitative attitude analysis	Zufriedenheit	Misst die Zufriedenheit des Benutzers auf quantitativem Niveau.

mehrere Maße verwendet werden. Für meine Evaluation benutze ich die Maße *task completion* (im Zusammenhang mit Beispiel-Anwendungen), *derivation from the critical path* und *time on task* (im Zusammenhang mit kontrollierten Experimenten), und *quantitative attitude analysis* (im Zusammenhang mit Benutzerbefragungen).

5.1.2 Allgemeine Methoden zur Usability-Evaluation

Zur Evaluation der Usability wurde eine Vielzahl von Methoden entwickelt. In meinen Untersuchungen setze ich die Methoden (1) Interview, (2) Feld-Beobachtung, (3) Fragebogen, (4) kontrolliertes Experiment und (5) Task-Analyse ein, die ich nachfolgend erklären möchte. Zur Auswahl der Methoden müssen besonders deren Vor- und Nachteile sowie deren Kosten/Nutzenverhältnis berücksichtigt werden.

Fragebogen Die Benutzer füllen einen Fragebogen aus, der konkrete Fragen zu verschiedenen Aspekten des Produktes enthält. Üblicherweise geben Ankreuzfelder ein festes Spektrum an Antworten vor. Ein besonderer Vorteil

der Methode ist, dass sie ohne großen Aufwand auf eine sehr große Gruppe von Testpersonen angewendet werden kann. Ein für mich wichtigerer Vorteil ist, dass die Testpersonen aufgrund der durch den Fragebogen gewährleisteten Anonymität ihre Aussagen nicht (bewusst oder unbewusst) abmildern.

Interview Ein einzelner Benutzer wird anhand eines Fragenkatalogs befragt. Vorteil dieser Methode gegenüber dem Fragebogen ist, dass Missverständnisse bzgl. der Fragen oder Antworten unwahrscheinlicher sind, da diese im direkten Gespräch geklärt werden können. Weiterhin kann im Interview spontan auf unerwartete Aussagen reagiert werden. Ein Nachteil gegenüber dem Fragebogen ist, dass Benutzer extreme Meinungen, die im Fragebogen klar heraustreten würden, im Interview evtl. nur in moderater Form äußern.

Kontrolliertes Experiment Ein einzelner Benutzer wird unter genau kontrollierten Bedingungen bei der Bearbeitung einer Aufgabe beobachtet. Durch Vorgabe der Arbeitsschritte kann die Vergleichbarkeit der Ergebnisse sichergestellt werden und Unsymmetrien können durch das Variieren der Arbeitsschritte vermieden werden. Ein Nachteil der Methode ist, dass die Ergebnisse aufgrund der künstlichen Versuchssituation evtl. nicht direkt auf eine echte Benutzungssituation übertragbar sind.

Feld-Beobachtung Bei Feld-Beobachtungen werden Benutzer bei ihrer täglichen Arbeit mit dem Produkt beobachtet. Somit ist die Versuchssituation im Gegensatz zum kontrollierten Experiment praxisrelevant. Ein Nachteil ist natürlich, dass die Rahmenbedingungen bei einer Feld-Beobachtung nicht beeinflusst werden können. Reproduzierbare quantitative Messungen werden dadurch erschwert.

Task-Analyse Bei dieser nicht-empirischen Methode wird untersucht, welche Arbeitsschritte durchgeführt werden müssen, um ein bestimmtes Ziel zu erreichen. Hieraus können dann Vorhersagen bzgl. des Schwierigkeitsgrades und des Aufwands der Umsetzung abgeleitet werden. Der große Vorteil bei dieser wie bei allen anderen nicht-empirischen Methoden ist, dass solche Untersuchungen relativ einfach und ohne Testkandidaten durchgeführt werden können.

5.1.3 Usability im Kontext von Programmier- und Spezifikationssprachen

Im Kontext von Programmier- und Spezifikationssprachen können spezialisierte, einfacher handhabbare Maße zur Messung der Usability verwendet werden. Nachfolgend definiere ich die Maße Anzahl der Codezeilen (LOC), Anzahl der Sprachkonstrukt-Klassen (SK-Klassen) und Anzahl der nichttrivialen Attributberechnungen.

LOC Ein sehr bekanntes Maß in der Softwaretechnik ist die Anzahl der Quelltextzeilen (*lines of code*), die zur Umsetzung eines bestimmten Ziels benötigt werden. Der Vorteil dieses Maßes ist, dass nicht die Bearbeitung einer Aufgabe selbst beobachtet, sondern nur die Lösung analysiert werden muss. Dabei ist allerdings zu beachten, dass sich Spezifikationszeilen in ihrer Komplexität erheblich unterscheiden können. In [28] wird dies am Beispiel attributierter Grammatiken in LIDO gezeigt. Lässt sich der Aufwand zur Erstellung einer durchschnittlichen Spezifikationszeile jedoch irgendwie abschätzen, ist das Maß ein wichtiger Indikator für die Komplexität und den Erstellungsaufwand einer Spezifikation.

Nachfolgend ist unter LOC die Anzahl der Spezifikationszeilen zu verstehen, wobei Kommentare und Leerzeilen nicht mitgezählt werden. Diese Definition ist sinnvoll, denn die Anzahl von Leerzeilen und Kommentaren kann selbst bei Spezifikationen eines Autors sehr stark variieren. Im Gegensatz dazu sind die Formatierungskonventionen selbst unterschiedlicher Autoren für eine gegebene Sprache meist relativ gut vergleichbar.

Zur Größenmessung visueller Teilspezifikationen wird die Anzahl der Sprachkonstrukt-Knoten der editierbaren Struktur gezählt. Im vorliegenden Fall wird das Maß zur Größenmessung von Generischen Zeichnungen verwendet. Zumindest dort ist die Größenordnung der resultierenden Werte mit der eines textuellen LOC-Maßes vergleichbar. Übersetzt man z.B. eine Generische Zeichnung mit 72 Sprachkonstrukt-Knoten, entsteht daraus im ersten Schritt eine textuelle Zwischenspezifikation gleichen Abstraktionsniveaus mit der Größe 136 LOC. Solche textuellen Repräsentationen wurden verwendet, bevor die visuelle Variante der Generischen Zeichnungen implementiert wurde.

Aufgrund der Vergleichbarkeit mit LOC-Werten textueller Spezifikationen bezeichne ich auch die Anzahl der Sprachkonstrukt-Knoten visueller Spezifika-

tionen mit LOC und gebe als Gesamtgröße einer Spezifikation die Summe aller LOC-Werte an.

SK-Klassen In der objektorientierten Programmierung wird neben dem Maß LOC häufig die Anzahl von Modell- oder Implementierungsklassen als Maß für die Systemkomplexität genutzt. Dieses Maß ist viel grober, misst jedoch besser die strukturelle Komplexität und kann auch in frühen Entwicklungsphasen schon angewendet werden.

Basierend auf diesen Überlegungen benutze ich nachfolgend die Anzahl der konkreten Klassen der editierbaren Struktur als Maß für die strukturelle Größe einer Sprache. Die abstrakten Klassen werden nicht mit gezählt, da es teilweise im Ermessen des Sprachentwicklers liegt, wie viele abstrakte Oberklassen er einführt. Die konkreten Klassen entsprechen dagegen direkt den Sprachkonstrukten, die dem Benutzer zur Verfügung stehen. Um Verwechslungen mit der Gesamtanzahl der Klassen auszuschließen, nenne ich die nicht-abstrakten Klassen *Sprachkonstrukt-Klassen* (SK-Klassen).

Nicht-triviale Attributberechnungen Die Komplexität von attributierten Grammatiken kann wie oben erwähnt bei gleicher Codegröße stark variieren. Da attributierte Grammatiken, die lediglich Anwendungen visueller Muster enthalten, vergleichsweise einfach sind, wird ein Maß benötigt, um dies auszudrücken.

Zur Parametrisierung von Musteranwendungen werden Kontrollattribute berechnet. Häufig ist der verwendete Ausdruck konstant (d.h. er hängt von keinem anderen Attribut ab) oder es wird lediglich der Wert eines anderen Attributs kopiert. Im Vergleich zu Attributberechnungen, die andere Attribute auf komplexe Weise miteinander verknüpfen, sind die oben genannten Arten von Attributberechnungen einfach und übersichtlich. Nachfolgend nenne ich Attributberechnungen, die entweder ein anderes Attribut kopieren oder einen konstanten Wert ergeben *trivial*, alle anderen Attributberechnungen *nicht-trivial*. Wie sich später zeigen wird, ist dies tatsächlich ein sehr treffendes Maß, um die niedrige Komplexität von Sichtdefinition zu zeigen, denn der Anteil trivialer Attributberechnungen liegt je nach Beispielspezifikation zwischen 60 und 100 Prozent.

Eine wichtige Eigenschaft dieser Klassifikation ist, dass nicht-triviale Attributberechnungen nicht durch eine Kombination mehrerer trivialer Attribut-

berechnungen ausgedrückt werden können. Dies bedeutet, nicht-triviale Berechnungen lassen sich nicht einfach „wegoptimieren“.

Weiterhin ist nützlich, dass diese Eigenschaft sehr leicht zu prüfen ist: Eine Attributberechnung ist genau dann nicht-trivial, wenn die rechte Seite (1) mindestens ein anderes Attribut benutzt *und* (2) mindestens eine Operation enthält. Unter Operationen sind hier Funktionsaufrufe, Konstruktoraufrufe, Verknüpfungsoperationen oder Fallunterscheidungen zu verstehen, die in LIDO sämtlich in Funktionsschreibweise geschrieben werden.

5.2 Usability des Generators

Aufgrund der Komplexität von DEViL ist es nicht praktikabel, alle Usability-Aspekte des Systems zu untersuchen. Ich habe daher zunächst Untersuchungsziele in Form konkreter Fragen definiert. Die Fragestellungen werden im nachfolgenden Abschnitt vorgestellt.

Basierend auf diesen Fragen habe ich dann spezifische Untersuchungen entwickelt, die die Fragen beantworten können. Die Untersuchungen basieren auf den bereits oben vorgestellten allgemein anwendbaren Methoden.

Da die Untersuchungen teilweise Beiträge zu mehreren Fragestellungen liefern und andererseits die Fragen nur durch Kombination mehrerer Untersuchungen umfassend beantwortet werden können, werden die Untersuchungen und die Auswertungen bzgl. der aufgestellten Fragestellungen nachfolgend getrennt behandelt.

In den Unterabschnitten 5.2.2 bis 5.2.5 werden die Entwurfsüberlegungen der angestellten Untersuchungen dargestellt und deren Durchführung beschrieben, ohne sie bereits zu bewerten. Die Hintergrundinformationen zur Durchführung sind wichtig, um die Ergebnisse interpretieren zu können.

In den Unterabschnitten 5.2.6 bis 5.2.10 werden die Untersuchungen schließlich im Kontext der Fragestellungen interpretiert und bewertet.

5.2.1 Zielsetzung

Bei der Evaluation habe ich mich auf folgende fünf Fragen konzentriert:

1. Wie einfach lassen sich Editoren für überschaubare Sprachen spezifizieren?

2. Wie wirksam sind visuelle Muster?
3. Wie einfach lässt sich die grafische Repräsentation nachträglich ändern?
4. Wie gut ist DEViL für große Projekte und Team-Entwicklung geeignet?
5. Wie gut lassen sich Sprachen umsetzen, bei denen semantische und editierbare Struktur unterschieden werden müssen?

Jeder dieser Punkte spielt eine bedeutende Rolle für die praktische Anwendung. Die möglichst einfache Spezifizierbarkeit kleiner Editoren ist wichtig, da im Rahmen des Entwurfs einer visuellen Sprache häufig kleine Prototypen entwickelt werden. Es ist vorteilhaft, wenn solche Prototypen schnell erstellt werden können, so dass sie als Grundlage zur Diskussion und weiteren Planung dienen können.

Der zweite Punkt, die Wirksamkeit der visuellen Muster, ist schon alleine deshalb interessant, weil visuelle Muster eine Besonderheit des vorgestellten Ansatzes sind. Es ist daher wichtig festzustellen, wie tragfähig dieses Konzept ist, welche Arten von Sprachen damit realisiert werden können und wo die Grenzen dieses Konzepts liegen.

Der dritte Punkt ist wichtig, wenn mit visuellen Repräsentationen experimentiert werden soll. Dies ist z.B. dann notwendig, wenn verschiedene Varianten einer Sprache verglichen werden sollen.

Der vierte Punkt behandelt die Frage, ob das Spezifikationskonzept skaliert und damit auch für große Projekte geeignet ist. Für praktisch relevante Projekte ist dies entscheidend, denn vor allem Spezialsprachen für spezifische Anwendungsfelder können unter Berücksichtigung aller Besonderheiten relativ umfangreich werden. Bei größeren Projekten ist es weiterhin wünschenswert, dass die Entwicklung im Team erfolgen kann.

Der letzte Punkt behandelt die Frage, wie gut DEViL zur Umsetzung von Sprachen geeignet ist, bei denen die semantische und die editierbare Struktur stark voneinander abweichen. Dies ist z.B. entscheidend, um Sprachen wie UML in vollem Umfang umsetzen zu können.

Wie in Abschnitt 5.1.1 dargelegt, muss zur Evaluation der Usability nicht nur das Benutzungsziel, sondern auch die Benutzergruppe klassifiziert werden. Bei meiner Untersuchung möchte ich die Fragen eins bis vier auf durchschnittlich eingearbeitete Nutzer und die Frage fünf auf Experten beziehen. Unter einem durchschnittlich eingearbeiteten Benutzer verstehe ich dabei

einen Benutzer, der schon erfolgreich mindestens eine visuelle Sprache mit DEViL implementiert hat.

Wenn Benutzer bereits Kenntnisse im Bereich von Sprachimplementierung und attributierten Grammatiken mitbringen, liegt die Einarbeitungszeit schätzungsweise im Bereich einiger Wochen. Falls das nicht der Fall ist, kann die Einarbeitungszeit durchaus einige Monate betragen.

Ein durchschnittlich eingearbeiteter Benutzer hat Kenntnisse über DSSL, Generische Zeichnungen, attributierte Grammatiken, visuelle Muster und die Anpassung der Repräsentationsstruktur. Im Gegensatz zu Experten kennt sich ein durchschnittlich eingearbeiteter Benutzer jedoch nicht unbedingt mit der internen Realisierungen der Muster-Varianten oder dem Konzept zur Trennung von semantischer und editierbarer Struktur aus.

5.2.2 Untersuchung 1: Implementierung von Beispielsprachen

Die wahrscheinlich wichtigste Methode zur Evaluation des Generators ist die Umsetzung von Beispielen. Nur so kann gezeigt werden, dass der Generator den Anforderungen praktisch relevanter Sprachen gewachsen ist. Nachfolgend sollen die umgesetzten Beispielsprachen kurz vorgestellt werden, damit der Leser einen Eindruck über das Anwendungsspektrum bekommt und die Statistiken in den nachfolgenden Abschnitten besser einordnen kann.

Teilweise habe ich die Beispielspezifikationen selbst entwickelt, einige andere gingen aus Diplom-, Studien- oder Projektarbeiten hervor. Manchmal wurden im Rahmen dieser Arbeiten neue Sprachen für bestimmte Anwendungsgebiete entworfen. In anderen Fällen wurden bekannte, teils standardisierte Sprachen umgesetzt.

Bei der Auswahl der umzusetzenden Sprachen wurde generell darauf geachtet, dass sie möglichst praxisrelevant sind und ein breites Spektrum von Darstellungskonzepten und grafischen Repräsentationen abdecken. Insgesamt umfassen die Beispielspezifikationen über 200 SK-Klassen und über 8.000 LOC. Die LOC-Werte beziehen sich dabei lediglich auf die Spezifikation des Editors. Evtl. zusätzlich implementierte Analysen und Codegenerierung wurden nicht berücksichtigt.

Die nachfolgende Aufstellung ist nach struktureller Komplexität (Anzahl SK-Klassen) sortiert. Die erste Hälfte ist demnach u.a. für die Frage relevant, wie einfach sich überschaubare Sprachen umsetzen lassen (siehe Abschnitt 5.2.6),

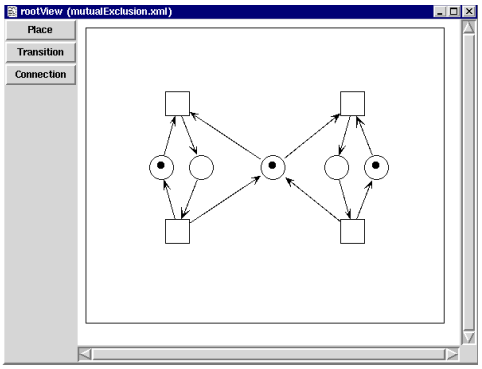
die zweite Hälfte für die Frage, wie gut sich DEViL für große Projekte und Team-Entwicklung eignet (siehe Abschnitt 5.2.9). Eins der großen Beispiele, UML, ist auch die Grundlage für Untersuchungen bzgl. der Separierbarkeit von semantischer und editierbarer Struktur (siehe Abschnitt 5.2.10).

Petri-Netze (4 SK-Klassen, 106 LOC, Abb. 5.2a) Petrinetze wurden in den 1960er Jahren entwickelt und dienen zur Modellierung von parallelen Abläufen [45]. Ein Petrinetz ist ein bipartiter Graph, dessen Knoten *Stellen* und *Transitionen* heißen. Stellen und Transitionen können durch gerichtete Kanten miteinander verbunden werden. Stellen werden durch Kreise und Transitionen durch Rechtecke dargestellt. Stellen können eine bestimmte Anzahl so genannter Marken sowie eine Kapazität besitzen. Die 4 SK-Klassen der Spezifikation sind Petri-Netz-Diagramme (Wurzelklasse), Stellen, Transitionen und Verbindungen.

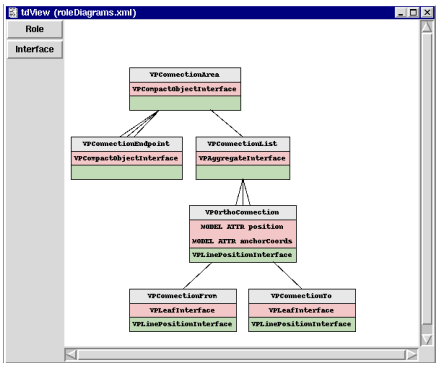
Rollen-Diagramme (6 SK-Klassen, 217 LOC, Abb. 5.2b) Rollen-Diagramme wurden bereits in Abschnitt 4.2.1 eingeführt. Sie beschreiben die Struktur und die Schnittstellen von Muster-Implementierungen in DEViL. Für diese Sprache wurde ein Editor entwickelt, um die Wartung der DEViL-Dokumentation zu vereinfachen. Die sechs SK-Klassen der Spezifikation sind Root (Wurzelklasse), Rollen-Diagramme, Wurzel-Rollen, Unter-Rollen, Schnittstellendefinitionen und Schnittstellenanwendungen.

Funktionsaufrufe (10 SK-Klassen, 101 LOC, Abb. 5.2c) Hierbei handelt es sich um eine frei erfundene textuelle Sprache, deren Notation an textuelle Programmiersprachen angelehnt ist. Die Spezifikation wurde u.a. entwickelt, um die Modellierung von Funktionsaufrufen und Parameterübergabe an einem kleinen Beispiel zu demonstrieren. Eine Besonderheit ist, dass die Repräsentation vollständig textuell ist. Die 10 SK-Klassen sind Programme (Wurzelklasse), Bibliotheken, Funktionsdefinitionen, Formale Parameter, Variablen-Definitionen, Zuweisungen, Konstanten, Variablen-Anwendungen, Funktionsaufrufe und aktuelle Parameter.

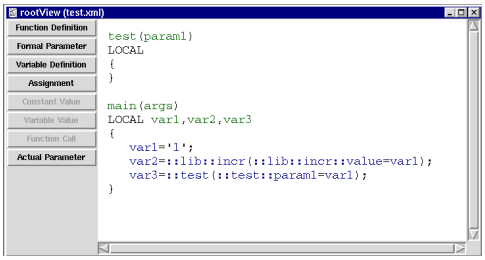
Mathematische Formeln (10 SK-Klassen, 189 LOC, Abb. 5.2d) Die Mathematik hat eine spezielle Schreibweise für Formeln, die wesentlich übersichtlicher als eine textuelle, eindimensionale Notation ist. Mathematische Ausdrücke enthalten z.B. zweidimensionale räumliche Relationen (z.B.



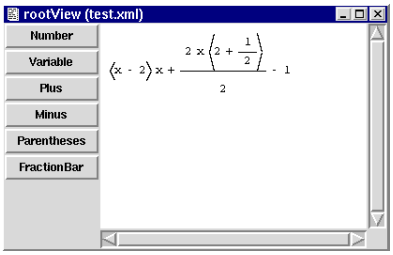
(a) Petri-Netze



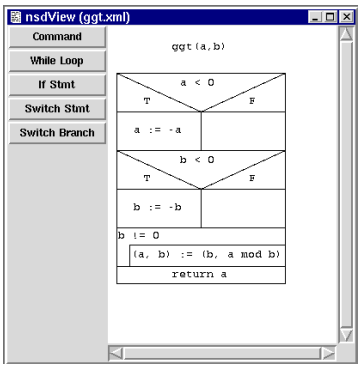
(b) Rollen-Diagramme



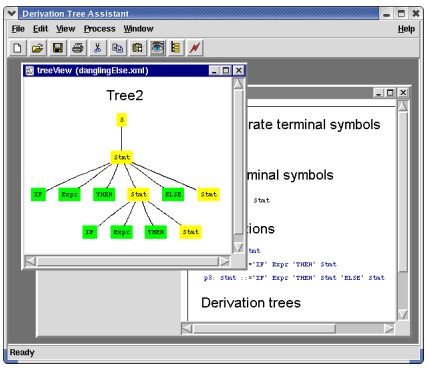
(c) Funktionsaufrufe



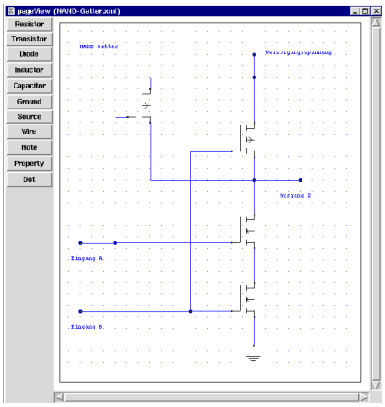
(d) Mathematische Formeln



(e) Nassi-Shneiderman Diagramme



(f) Derivation Tree Assistant



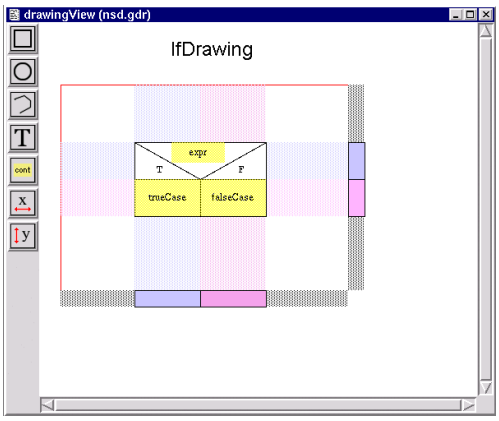
(g) Elektronische Schaltungen

Abbildung 5.2: Bildschirmfotos von generierten Beispieleditoren (Teil 1)

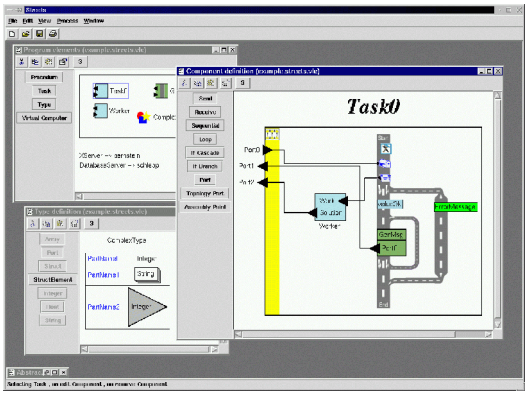
Brüche, Hoch- und Tiefstellungen, Vektoren und Matrizen), Symbole (z.B. Integral- oder Summenzeichen), Verzierungen (z.B. Vektorpfeile an Variablen) und Größenunterschiede (z.B. bei geschachtelten Ausdrücken). Die erstellte Beispielimplementierung realisiert nur einen kleinen Teil dieser Notation. Die 11 SK-Klassen sind Formel-Systeme (Wurzelklasse), Zahlen, Variablen, Operatoren, Bruchstriche, Klammern, Matrizen, Matrix-Zeilen, Matrix-Spalten sowie Matrix-Zellen.

Nassi-Shneiderman Diagramme (11 SK-Klassen, 311 LOC, Abb. 5.2e) Nassi-Shneiderman Diagramme, auch Struktogramme genannt, dienen der Darstellung imperativer Programmstrukturen (siehe Abschnitt 2.2.2). Ablaufstrukturen wie z.B. Schleifen oder Fallunterscheidungen werden durch einfache grafische Strukturen aus Linien und Text dargestellt. Durch Schachtelung und Aneinanderreihung können komplexere Strukturen gebildet werden. Die 11 SK-Klassen der Spezifikation sind Programme (Wurzelklasse), Nassi-Shneiderman Diagramme, Blöcke, bedingte Anweisungen, Switch-Konstrukte, Zweige in Switch-Konstrukten, While-Schleifen, Repeat-Schleifen, Schleifen ohne Abbruchbedingung, Break-Anweisungen und Kommandos.

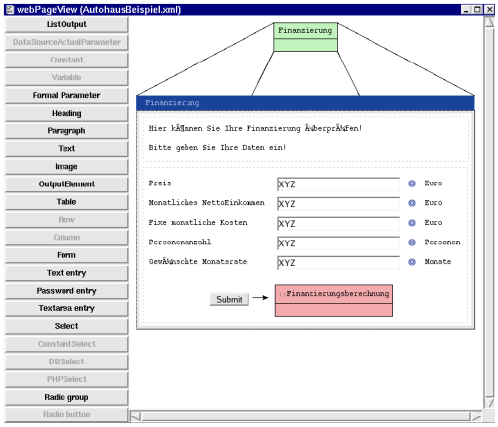
Derivation Tree Assistant (12 SK-Klassen, 275 LOC, Abb. 5.2f) Mit dieser Umgebung lassen sich grafische Repräsentationen für Ableitungsbäume konstruieren, die z.B. für Musterlösungen von Übungsaufgaben im Programmiersprachbereich häufig erstellt werden müssen. Hierzu wird im Editor zunächst eine Grammatik in Form einer textuellen Repräsentation konstruiert. Im Anschluss daran werden basierend auf dieser Grammatik Ableitungsbäume konstruiert. Dazu wird jedem Nichtterminal-Knoten eine Produktion zugeordnet. Die Unterknoten werden dann automatisch erstellt, so dass aus Sicht des Benutzers der Ableitungsbaum durch schrittweise Expansion der Baumknoten konstruiert wird. Die Ableitungsbäume werden bei nachträglichen Änderungen der Produktionen automatisch angepasst. Die 12 SK-Klassen der Spezifikation sind Root (Wurzelklasse), Terminal-Definitionen, Nichtterminal-Definitionen, Produktionen, literale Terminalsymbole, nicht-literale Terminalsymbole, Nichtterminal-Symbole, Ableitungsbäume, Wurzeln von Ableitungsbäumen, literale Terminal-Knoten, nicht-literale Terminal-Knoten sowie Nichtterminal-Knoten von Ableitungsbäumen.



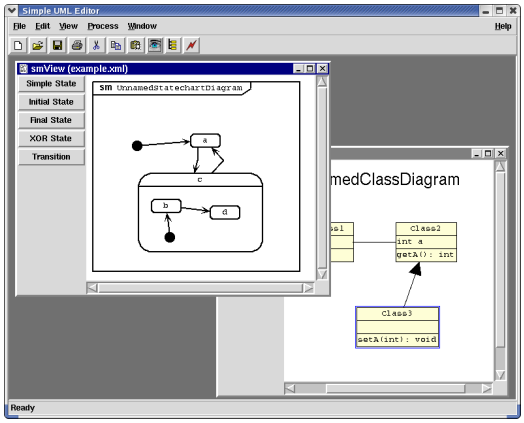
(a) Generische Zeichnungen



(b) Streets



(c) PaderWAVE



(d) UML

Abbildung 5.3: Bildschirmfotos von generierten Beispieleditoren (Teil 2)

Elektronische Schaltungen (21 SK-Klassen, 761 LOC, Abb. 5.2g) In der Elektrotechnik werden Schaltpläne durch eine visuelle Sprache beschrieben. Bauteile wie Widerstände, Kondensatoren oder Transistoren werden durch grafische Symbole und elektrische Leitungen zwischen ihnen durch Linien dargestellt. Die Bauteile haben je nach Beschaffenheit eine unterschiedliche Anzahl von Anschlüssen. Die 21 SK-Klassen der Spezifikation sind Root (Wurzelklasse), Schaltplan-Seiten, Drähte, Verbindungspunkte, Bauelement-Anschlusspunkte, Bauelement-Eigenschaften, Notizen sowie 14 Bauelemente wie Transistoren, Widerstände oder Stromquellen.

Generische Zeichnungen (27 SK-Klassen, 1090 LOC, Abb. 5.3a) Generische Zeichnungen wurden bereits in Abschnitt 4.4 vorgestellt. Sie dienen zur Spezifikation konkreter Repräsentationen für das Formular- und FormList-

Muster. Eine Besonderheit dieser visuellen Sprache ist, dass das Layout der konstruierten Ausdrücke in weiten Teilen semantisch relevant ist. Teile des Editors haben somit starke Ähnlichkeit mit einem Programm zum Zeichnen von Vektorgrafiken.

Streets (27 SK-Klassen, 665 LOC, Abb. 5.3b) *Streets* ist eine Sprache, die 1998 von einer Projektgruppe der Universität Paderborn entwickelt wurde (siehe Abschnitt 2.2.4). Sie soll auch Nicht-Experten die imperative Programmierung paralleler Berechnungen ermöglichen. Visuell fällt besonders die anspruchsvolle grafische Darstellung ins Auge. Ursprünglich wurde *Streets* von Hand implementiert. Die DEViL-Spezifikation für *Streets* enthält weder die Codegenerierung noch alle grafischen Konstrukte der Originalimplementierung. Sie soll in erster Linie die Machbarkeit anspruchsvoller grafischer Darstellungen demonstrieren.

UML (60 SK-Klassen, 1641 LOC, Abb. 5.3d) Die *Unified Modeling Language* wurde bereits in Abschnitt 2.2.1 vorgestellt. UML der Version 2.0 besteht aus 13 Diagrammsprachen, mit denen statische und dynamische Aspekte von Softwaresystemen modelliert werden können. Die UML-Spezifikation für DEViL umfasst nur die wichtigsten Diagrammarten, nämlich Klassendiagramme, Zustandsdiagramme und Sequenzdiagramme. Teile der Spezifikation wurden in Studienarbeiten entwickelt [44, 57]. Auf die Erfahrungen mit UML und insbesondere der Unterscheidung von semantischer und editierbarer Struktur gehe ich in Abschnitt 5.2.10 genauer ein.

PaderWAVE (92 SK-Klassen, 2930 LOC, Abb. 5.3c) Diese Sprache wurde in Jahr 2005 ebenfalls von einer Projektgruppe der Universität Paderborn entwickelt [67, 55]. Sie dient zur Generierung von Web-Anwendungen aus visuellen Spezifikationen. In PaderWAVE gibt es verschiedene Sichten, die jeweils bestimmte Aspekte einer Web-Anwendung beschreiben. Zu den wichtigsten zählen die Navigationssicht, die Webseiten-Sicht, die WebAction-Sicht sowie die Datenbanksicht. Im Unterschied zu *Streets* wurde bei der Entwicklung von PaderWAVE von Anfang an DEViL benutzt. Als Betreuer der Projektgruppe hatte ich die Möglichkeit, die Studenten während ihrer Arbeit zu beobachten (siehe Abschnitt 5.2.3).

5.2.3 Untersuchung 2: Feld-Beobachtung einer Projektgruppe

Wie bereits erwähnt wurde die Sprache PaderWAVE im Rahmen einer studentischen Projektgruppe entwickelt [67, 55]. Die Projektgruppe umfasste acht Teilnehmer und bestand über einen Zeitraum von 12 Monaten. Während dieser Zeit erfolgte die Einarbeitung in den Problembereich, die Einarbeitung in DEViL, der Sprachentwurf, die Implementierung des Editors, die Entwicklung von Tests und Beispielen sowie die Dokumentation der Ergebnisse. Der Entwurf und die Implementierung wurden in zwei Iterationen ungefähr gleicher Dauer durchgeführt. In der ersten Phase wurde ein Prototyp mit eingeschränkter Funktionalität entworfen und implementiert. In der zweiten Phase wurde basierend auf den Erkenntnissen der ersten Phase das endgültige System realisiert.

Zur Projektplanung wurde ein wöchentliches Statustreffen abgehalten, in denen Teilaufgaben an Gruppenmitglieder oder Untergruppen delegiert wurden. Generell haben sich drei der acht Teilnehmer auf die Spezifikation der visuellen Darstellung spezialisiert, während die anderen fünf Teilnehmer die Implementierung der Codegenerierung übernahmen.

Als Betreuer habe ich an den wöchentlichen Statussitzungen teilgenommen und stand für technische Hilfe zu DEViL zur Verfügung. Dadurch konnte ich den Einfluss von DEViL auf das konzeptionelle und technische Vorgehen der Projektgruppe sehr gut beobachten. Die Ergebnisse der Beobachtungen werden in Abschnitt 5.2.9 dargestellt.

5.2.4 Untersuchung 3: Fragebogen

Um Tests und Befragungen an Benutzern durchzuführen, wurde eine Test-Gruppe bestehend aus vier Kandidaten aufgestellt. Die vier Kandidaten waren zu diesem Zeitpunkt die einzigen greifbaren DEViL-Benutzer mit hinreichenden Kenntnissen bzgl. der Sicht-Spezifikation. Alle Testkandidaten waren Projektgruppen-Mitglieder. Drei der Kandidaten gehörten zum Team, das in der Projektgruppe das visuelle Frontend umgesetzt hat, der vierte Teilnehmer hatte aufgrund seiner Tätigkeit als studentische Hilfskraft Erfahrungen mit der Spezifikation visueller Editoren.

Um eine Benutzer-Bewertung bzgl. der Anwenderfreundlichkeit, Flexibilität und Mächtigkeit von DEViL zu erhalten, wurde der Test-Gruppe ein Fragebogen vorgelegt. Der Fragebogen enthielt Fragen zu folgenden Themenbereichen:

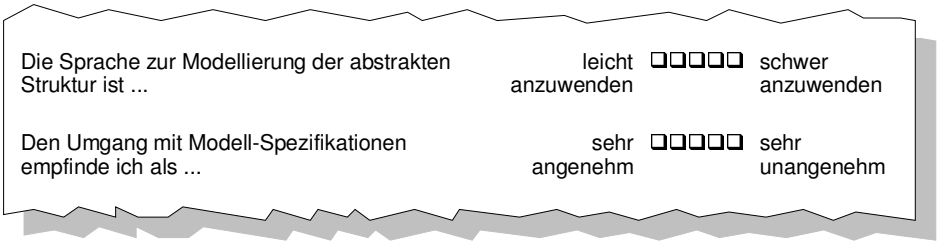


Abbildung 5.4: Ausschnitt aus dem verwendeten Fragebogen

- Aktueller Kenntnisstand
- Generelle Bewertung von DEViL
- Bewertung der Teilsprache zur Definition der abstrakten Struktur
- Bewertung der Teilsprache zur Definition Generischer Zeichnungen
- Bewertung der Teilsprache zur Definition visueller Sichten
- Bewertung des Konzepts der visuellen Muster
- Bewertung der Unterstützung großer Projekte und Team-Entwicklung

Fast alle Fragen konnten durch Ankreuzfelder beantwortet werden. Es wurde eine 5-stufige Skala verwendet, wie in Abbildung 5.4 zu sehen. An den Enden der Skala befinden sich zwei gegensätzliche Extremaussagen, die im Folgenden mit 1 (linke Seite) und 5 (rechte Seite) bezeichnet werden. Durch Ankreuzen eines der fünf Felder können die Extremaussagen in gewünschter Form abgeschwächt werden.

Die Aussagen der Teilnehmer zu den einzelnen Themengebieten werden weiter unten im Kontext der entsprechenden Evaluationsfragen dargestellt und diskutiert (siehe Abschnitte 5.2.6, 5.2.7, 5.2.8, und 5.2.9). Die Aussagen zum Kenntnisstand sind bereits in Tabelle 5.2 dargestellt, um die Einordnung der Test-Gruppe zu ermöglichen.

Datenaufbereitung Bevor auf den Inhalt eingegangen werden kann, muss zunächst die Datenaufbereitung geklärt werden. Da die Anzahl der Teilnehmer viel zu klein ist, um die Daten sinnvoll statistisch aufbereiten zu können, beschränke ich mich bei der Präsentation der Ergebnisse auf den Durchschnitt und die Standardabweichung der Antworten. Am Durchschnitt lässt sich die

Tabelle 5.2: Fragebogen Teil 1: Aktueller Kenntnisstand

Frage	Durchschnitt	Standardabw.
Ich verstehe das Konzept der Modell-Spezifikationssprache zu... (1) null Prozent ... (5) hundert Prozent	4,0	0,7
Ich verstehe das Konzept der Generischen Zeichnungen zu ... (1) null Prozent ... (5) hundert Prozent	5,0	0,0
Ich verstehe das Konzept der visuellen Muster zu ... (1) null Prozent ... (5) hundert Prozent	4,0	0,7
Ich verstehe das Konzept der Grammatik-Abbildung zu ... (1) null Prozent ... (5) hundert Prozent	2,8	0,4
Ich verstehe das Konzept der Sicht-Spezifikation zu ... (1) null Prozent ... (5) hundert Prozent	4,3	0,4
Ich verstehe DEViL insgesamt zu ... (1) null Prozent ... (5) hundert Prozent	4,0	0,0
Bezüglich der Spezifikation visueller Darstellungen mit DEViL bin ich ... (1) Anfänger ... (5) Experte	3,5	0,5

Tabelle 5.3: Beispiele zur Interpretation der Standardabweichung

Werteverteilung	Standardabw.
1; 1; 1; 2	0,4
1; 1; 2; 2	0,5
1; 2; 2; 3	0,7
1; 1; 2; 3	0,8
1; 1; 1; 3	0,9
1; 1; 3; 3	1,0
1; 2; 3; 4	1,1
1; 1; 1; 4	1,3
1; 1; 4; 4	1,5

Tendenz der Antworten erkennen, an der Standardabweichung, wie einig sich die Kandidaten über diese Aussage sind.

Um ein Gefühl für die Interpretation der Standardabweichung zu vermitteln, sind in Tabelle 5.3 einige Ergebnisvektoren sortiert nach Standardabweichung aufgeführt. Wie zu erkennen schwanken die Einzelwerte bis zur Standardabweichung 0,5 um höchstens ein Feld, bis zur Standardabweichung 1,0 um höchstens 2 Felder. Bei Standardabweichungen über 1,0 ist die Tendenz der Aussage nicht mehr eindeutig.

Verwendete Begriffe Um die Aussagen in Tabelle 5.2 verstehen zu können ist es wichtig zu wissen, dass gegenüber der Test-Gruppe eine andere Terminologie verwendet wurde als in dieser Arbeit. Anstelle des Begriffs „Syntax“ wurde dort der Begriff „Modell“ verwendet. Dementsprechend ist mit dem „Konzept der Modell-Spezifikationssprache“ das Konzept von DSSL gemeint. Nachfolgend werden trotzdem die Original-Formulierungen aufgeführt, da auch Formulierungen Einfluss auf das Ergebnis haben können.

Kenntnisstand der Test-Gruppe Aus Tabelle 5.2 wird deutlich, dass die Teilnehmergruppe recht homogen ist. Bei allen Aussagen ist die Standardabweichung kleiner als 0,8. Zusammenfassend attestieren sich die Teilnehmer insgesamt ein überwiegend gutes, wenn auch nicht perfektes Verständnis der Spezifikationskonzepte. Eine Ausnahme bilden lediglich die Generischen Zeichnungen, die die Teilnehmer nach eigenen Angaben zu hundert Prozent verstehen.

Es mag zunächst verwunderlich sein, dass die Testkandidaten sich in den anderen Bereichen kein hundertprozentiges Verständnis bescheinigen, obwohl sie monatelang mit dem System gearbeitet haben. Die Selbsteinschätzung ist aus meiner Sicht jedoch tatsächlich zutreffend, denn in den Experimenten konnte ich z.B. beobachten, dass einigen Teilnehmern bestimmte Prinzipien zur Anwendung von Muster-Rollen nicht bewusst waren, was darauf hindeutet, dass sie noch nicht alle Aspekte des Systems durchdrungen hatten.

Eine negative Ausnahme beim Verständnis ist das Konzept der Grammatik-Abbildung, bei dem die Bewertung mit einem Durchschnitt von 2,8 sehr niedrig ausfiel. Das kann daran liegen, dass dieser Mechanismus tatsächlich nur bei durchschnittlich 4 Prozent aller SK-Klassen benötigt wird.

Insgesamt halte ich das Verständnisniveau der Teilnehmer für eine gute Grundlage, um die Stärken und Schwächen des Systems aufzudecken.

5.2.5 Untersuchung 4: Kontrollierte Experimente mit nachfolgendem Interview

Um zu untersuchen, wie effizient Sprachspezifikationen mit DEViL erstellt und geändert werden können, wurden mit der oben vorgestellten Test-Gruppe kontrollierte Experimente durchgeführt. Es wurden ihnen insgesamt drei praktische Aufgaben gestellt, deren Umsetzung jeweils ca. eine bis zwei Stunden erforderte. Zu den Aufgaben zählten

- die Spezifikation eines Editors für eine kleine, vorgegebene Sprache,
- die Änderung der grafischen Repräsentation in einer bekannten Spezifikation und
- die Einarbeitung in eine unbekannte Spezifikation.

Die Bearbeitung der Aufgaben wurde während der gesamten Zeit beobachtet und protokolliert. Eingeleitet wurde die Arbeitsphase mit einer schriftlichen und verbalen Erklärung der Aufgabe. Des Weiteren wurde der Arbeitsablauf vorgegeben, um die Ergebnisse vergleichbarer zu machen. Für die erste Aufgabe lautete der Arbeitsablauf z.B. (1) entwickeln Sie ein Modell der Sprache, (2) testen Sie Ihr Modell, (3) entwickeln Sie Generische Zeichnungen für die grafische Darstellung, (4) entwickeln Sie die Sicht-Spezifikation und (5) testen Sie die Sicht-Spezifikation.

Zur Evaluation wurden die Effizienz-Maße *time on task*, und *derivation from the critical path* verwendet. Das letztgenannte Maß ist im Fall von Spezifikationssprachen allerdings schwer konkretisierbar. Das liegt daran, dass nichttriviale Programme und Spezifikationen normalerweise iterativ entwickelt werden. Hierbei wird wiederholt die schon erstellte Teilspezifikation gelesen und kleine Teile geändert oder ergänzt [19]. Es wird also in der Regel nicht von Anfang an gültiger Programmcode geschrieben, sondern es wird mit unvollständigem, unzusammenhängendem und teilweise falschem Code gearbeitet. Aus diesem Grund werde ich mich beim Maß *derivation from the critical path* auf qualitative Aussagen beschränken und nur Abweichungen erwähnen, die offensichtlich auf gedankliche Fehler der Testperson hinweisen.

Um die Messung der Bearbeitungszeit nicht zu verfälschen, wurden den Testpersonen während der Arbeit keine Fragen gestellt. Stattdessen wurde im Anschluss an das Experiment ein Kurzinterview durchgeführt. Darin wurde nachgefragt, als wie schwierig die Aufgaben empfunden wurden und wie angenehm die Bearbeitung aus Sicht der Testperson war. Falls sich im Laufe des Experimentes spezielle Probleme oder Besonderheiten ergeben haben, wurden auch diese angesprochen.

Nachdem nun die einzelnen Untersuchungen vorgestellt wurden, werden nachfolgend die oben aufgestellten Fragen beantwortet.

5.2.6 Wie einfach lassen sich Editoren für überschaubare Sprachen spezifizieren?

Benutzbarkeit der Spezifikationssprachen Tabelle 5.4 zeigt die Ergebnisse des Fragebogenteils, der sich auf die Einfachheit der Benutzung von DEViL bezieht. Im allgemeinen Teil wurden DEViL-Spezifikationen einheitlich als relativ gut verständlich und der Umgang mit DEViL als eher angenehm beurteilt. Die vier Teilaspekte (1) abstrakte Struktur, (2) Generische Zeichnungen, (3) visuelle Muster und (4) Sicht-Spezifikationen insgesamt werden ebenfalls durchgängig als relativ einfach und angenehm beurteilt. Bemerkenswert ist, dass Generische Zeichnungen durchweg als leicht anzuwenden und sehr angenehm empfunden wurden.

Das komplizierteste Konzept ist laut Test-Gruppe die Grammatik-Abbildung. Dies passt zu der Selbsteinschätzung der Kandidaten, dieses am schlechtesten verstanden zu haben.

Tabelle 5.4: Fragebogen Teil 2: Einfachheit der Benutzung

Frage	Durchschnitt	Standardabw.
DEViL-Spezifikationen sind ... (1) gut verständlich ... (5) schwer verständlich	1,8	0,4
Den Umgang mit DEViL empfinde ich als ... (1) sehr angenehm ... (5) sehr unangenehm	2,3	0,4
Die Sprache zur Modellierung der abstrakten Struktur ist ... (1) leicht anzuwenden ... (5) schwer anzuwenden	1,8	0,8
Statische Fehlermeldungen bzgl. des Modells sind ... (1) sehr gut verständlich ... (5) sehr schlecht verständlich	3,5	1,1
Den Umgang mit Modell-Spezifikationen empfinde ich als ... (1) sehr angenehm ... (5) sehr unangenehm	1,8	0,4
Generische Zeichnungen sind ... (1) leicht anzuwenden ... (5) schwer anzuwenden	1,0	0,0
Statische Fehlermeldungen zu Generischen Zeichnungen sind ... (1) sehr gut verständlich ... (5) sehr schlecht verständlich	1,5	0,9
Den Umgang mit Generischen Zeichnungen empfinde ich als ... (1) sehr angenehm ... (5) sehr unangenehm	1,0	0,0
Visuelle Muster sind ... (1) leicht anzuwenden ... (5) schwer anzuwenden	2,0	0,7
Statische Fehlermeldungen zur Anwendung visueller Muster sind ... (1) sehr gut verständlich ... (5) sehr schlecht verständlich	3,0	0,7
Den Umgang mit visuellen Mustern empfinde ich als ... (1) sehr angenehm ... (5) sehr unangenehm	2,5	0,5
Die Spezifikation einer visuellen Sicht ist ... (1) sehr einfach ... (5) sehr kompliziert	2,0	0,7
Das Konzept der Grammatik-Abbildung ist ... (1) sehr einfach ... (5) sehr kompliziert	3,0	0,7
Den Umgang mit Sicht-Spezifikationen empfinde ich als ... (1) sehr angenehm ... (5) sehr unangenehm	2,0	0,7

Die Qualität der statischen Fehlermeldungen wurde neutral mit leichter Tendenz zu schlechter Verständlichkeit beurteilt, wobei lediglich die Generischen Zeichnungen eine positive Ausnahme bilden. Überraschend ist, dass die Fehlermeldungen des Modells schlechter als die zu den Musteranwendungen abgeschnitten haben, denn nach meiner Einschätzung sind die Fehlermeldungen bzgl. des Modells sehr gut und die Fehlermeldungen bzgl. der Musteranwendungen teilweise schlecht verständlich.

Größenordnung des Arbeitsaufwands Um den Arbeitsaufwand zur Spezifikation eines einfachen Editors zu ermitteln, sollten die Testkandidaten in einem kontrollieren Experiment einen Editor für vereinfachte Nassi-Shneiderman Diagramme spezifizieren. Dazu wurde ihnen folgende Aufgabe gestellt.

> In diesem Experiment sollen Sie einen Nassi-Shneiderman Editor spezifizieren. Die Abbildung 5.5 zeigt einen Screenshot. Ein Nassi-Shneiderman Diagramm hat einen Namen und eine Liste von Anweisungen. Anweisungen können If-Abfragen oder Kommandos sein. If-Abfragen bestehen aus einer Bedingung und zwei Listen von Anweisungen, jeweils eine für den True-Zweig und eine für den False-Zweig. Kommandos sind Rechtecke, die einen Text enthalten. Bedingungen können ebenfalls als Zeichenkette modelliert werden.

Die Musterlösung für diese Aufgabe ist in Abbildung 4.4 auf Seite 136 dargestellt. Sie enthält ca. 16 LOC Struktur-Spezifikation, 22 LOC für Generische Zeichnungen und 41 LOC Sicht-Spezifikation. Insgesamt sind dies also ca. 79 LOC.

Keiner der Testkandidaten hatte zuvor mit diesem Anwendungsbeispiel zu tun. Allerdings gab es in der Dokumentation zu DEViL, die den Teilnehmern bekannt war, eine Generische Zeichnung zu Nassi-Shneiderman Diagrammen. Dies sollte das Ergebnis aber nicht wesentlich verfälschen.

Tabelle 5.5 zeigt die Ergebnisse der Untersuchung. Der Zeitbedarf für die Spezifikation von abstrakter Syntax, Generischen Zeichnungen und Sichten wurde separat gemessen. Insgesamt haben die Teilnemer durchschnittlich 95 Minuten für diese Aufgabe benötigt, wobei ca. 35% für die abstrake Struktur, 10% für Generische Zeichnungen und 55% für die Sicht aufgewendet wurden.

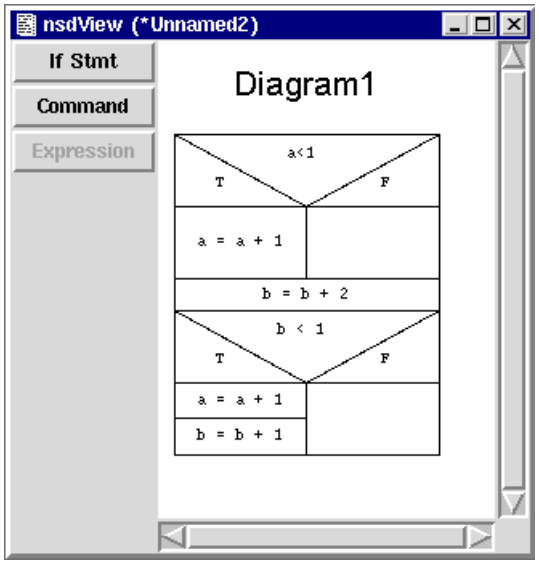


Abbildung 5.5: Bildschirmfoto eines Editors für vereinfachte Nassi-Shneiderman Diagramme

Tabelle 5.5: Messungen im Rahmen der Spezifikation eines Editors für vereinfachte Nassi-Shneiderman Diagramme

Messung	Durchschnitt	Standardabw.
Zeitbedarf zur Spezifikation der abstrakten Struktur [min]	32	16
Zeitbedarf zur Spezifikation der Gen. Zeichnungen [min]	11	6
Zeitbedarf zur Spezifikation der Sicht [min]	52	7
Summe Zeitbedarf [min]	95	26
Die Aufgabe war ... (1) sehr einfach ... (5) sehr schwer	1,8	0,4
Die Bearbeitung der Aufgabe empfand ich als ... (1) sehr angenehm ... (5) sehr unangenehm	2,5	0,5

In Anbetracht der Tatsache, dass die Gesamtlänge der Lösung lediglich 79 LOC beträgt, erscheint der Zeitbedarf auf den ersten Blick sehr hoch. Rechnerisch haben die Kandidaten über eine Minute für jede Codezeile benötigt. Während dieser Zeit haben die Kandidaten jedoch auch andere Beispielspezifikationen und das DEViL Benutzerhandbuch zu Rate gezogen, um daraus die Syntax oder die Struktur bestimmter Teillösungen zu entnehmen. Außerdem ist in dieser Zeit auch das Testen und Korrigieren der Spezifikation enthalten. Der Zeitbedarf von Entwicklung und Test konnte leider nicht getrennt gemessen werden, da beide Arbeitsschritte trotz gegenteiliger Arbeitsanweisung häufig sehr verzahnt stattfanden.

Ein erheblicher Teil des Zeitbedarfs ist aber nach meiner Beobachtung auch durch Abweichungen vom kritischen Pfad entstanden. Alle Kandidaten haben viele Fehler gemacht und das Vorgehen erschien teilweise wenig zielführend. Nach meinem Eindruck hätten alle Kandidaten die Aufgabe in der Hälfte der Zeit lösen können, wenn die Fehler hätten vermieden werden können. Die Gründe für die Fehler sind meiner Ansicht nach vielschichtig. Einige Kandidaten fühlten sich durch die Beobachtungssituation irritiert. Es ist denkbar, dass sie die Aufgabe ohne Beobachtung effizienter gelöst hätten. Des Weiteren verwirrte einige Kandidaten, dass im Beschreibungstext deutsche Bezeichnungen für die Sprachkonstrukte verwendet wurden, wohingegen die Schaltflächen in Abbildung 5.5 in englisch beschriftet sind. Schließlich ist der Ursprung bestimmter Fehlermeldungen des Generators nur schwer zu erkennen, was zusätzlich Zeit kostet.

Ein beträchtlicher Teil des Zeitbedarfs ist sicherlich damit zu erklären, dass die Kandidaten bestimmte Zusammenhänge des Spezifikationskonzepts noch nicht verinnerlicht hatten. Besonders eindrucksvoll lässt sich dies am Beispiel der Struktur-Spezifikation zeigen. Während ein Testkandidat hierfür lediglich 5 Minuten brauchte, benötigten die anderen drei jeweils ca. 40 Minuten. Dies ist damit zu erklären, dass der schnelle Kandidat im Rahmen der Projektgruppe oft abstrakte Strukturen entworfen hat, während die anderen überwiegend solche Entwürfe benutzt haben, um basierend darauf die visuelle Darstellung zu spezifizieren. Große Probleme hatten die in diesem Punkt unerfahrenen Kandidaten z.B. mit der Modellierung der Aussage „Anweisungen können If-Abfragen oder Kommandos sein“, die in DEViL durch das Erben von einer gemeinsamen Oberklasse ausgedrückt werden muss.

Erstaunlicherweise haben die Kandidaten selbst die Aufgabe als relativ einfach eingestuft und die Bearbeitung als eher angenehm empfunden. Es ist

möglich, dass der Arbeitsprozess aus Sicht der Kandidaten tatsächlich zielführend war, während er von außen betrachtet chaotisch wirkte.

Insgesamt hat die Untersuchung gezeigt, dass sich die Benutzung der Spezifikationssprachen (zumindest unter Testbedingungen) schwieriger darstellt als angenommen. Besonders am Beispiel der abstrakten Syntax wird deutlich, dass die Leistung der Kandidaten noch weit von der *experienced user performance* (EUP) entfernt ist. Die Folgerung ist, dass entweder die Kernkompetenzen zum Umgang mit DEViL noch besser vermittelt werden müssen oder die Spezifikation mit Hilfe von interaktiven Werkzeugen intuitiver gemacht werden muss. Die Größenordnung des Zeitbedarfs zur Spezifikation eines vollständigen Editors ist allerdings schon jetzt erfreulich niedrig.

Arbeitsaufwand in Abhängigkeit zum Implementierungsziel Tabelle 5.6 enthält einige Charakteristiken von Spezifikationen kleiner und mittlerer Beispiele, die Aussagen zur Komplexität der Spezifikation erlauben. Die erste Zeile zeigt zum Vergleich die Daten des oben vorgestellten Editors für vereinfachte Nassi-Shneiderman Diagramme. Alle anderen Beispiele wurden in Abschnitt 5.2.2 vorgestellt.

Die Spalte *SK-Klassen* gibt die Anzahl der Sprachkonstrukt-Klassen der Spezifikation an (siehe Abschnitt 5.1.3). Sie ist ein Maß für die Komplexität der abstrakten Syntax. Die Auswahl reicht von sehr kleinen (4 SK-Klassen) bis mittelgroßen Sprachen (27 SK-Klassen). In keinem dieser Beispiele wird zwischen semantischer und editierbarer Struktur unterschieden. Die Spalte *Abweichungen Repräsentationsstruktur* gibt die Anzahl der SK-Klassen an, die eine vom Standard abweichende Repräsentationsstruktur besitzen. Dieser Fall tritt nur bei mathematischen Formeln und beim Editor für Generische Zeichnungen auf und betrifft dort durchschnittlich nur jedes zehnte Sprachkonstrukt. Betrachtet man alle Spezifikationen zusammen, beläuft sich der Anteil der Abweichungen auf lediglich vier Prozent. Dies unterstreicht, dass die Standard-Abbildung auf die Repräsentationsstruktur sehr zweckmäßig ist und den Spezifikationsaufwand stark reduziert.

Die weiteren Spalten geben die Anzahl der Codezeilen für bestimmte Aspekte der Spezifikation an. Die Spalte *LOC Struktur* misst die Anzahl der Codezeilen zur Spezifikation der SK-Klassen. Das Verhältnis *LOC Struktur* zu *SK-Klassen* drückt im Wesentlichen aus, wie komplex eine durchschnittliche SK-Klasse ist, d.h. wie viele Attribute sie besitzt. Jedoch spielt auch die Anzahl der abstrakten Klassen eine Rolle, da abstrakte Klassen nicht als SK-Klassen gezählt

Tabelle 5.6: Komplexität kleiner Spezifikationen

Sprache	SK-Klassen	Abw. Repräsentationsstruktur	LOC Struktur	LOC Sichtspezifikationen	LOC Gen. Zeichnungen	LOC Prüfungen und Kopplung	LOC Hilfsfunktionen	LOC Gesamt	Nichttriv. Attributberechn.	LOC Gesamt / LOC Struktur
Vereinfachtes NSD-Beispiel	4	0	16	41	22	0	0	79	0	4,9
Petri-Netze	4	0	31	41	34	0	0	106	1	3,4
Rollen-Diagramme	6	0	41	129	36	0	11	217	14	5,3
Funktionsaufrufe	10	0	51	24	0	45	8	101	0	2,0
Mathematische Formeln	10	1	45	90	15	39	0	189	0	4,2
Nassi-Shneiderman Diagramme	11	0	51	143	103	14	0	311	0	6,1
Derivation Tree Assistant	12	0	73	134	24	44	0	275	9	3,8
Elektronische Schaltungen	21	0	111	377	228	0	45	761	15	6,9
Generische Zeichnungen	27	2	184	518	157	135	96	1090	48	5,9

werden. Einerseits können abstrakte Klassen die Anzahl der Codezeilen erhöhen, andererseits können dadurch, dass in Unterklassen durch Vererbung weniger Attribute definiert werden müssen, auch Codezeilen eingespart werden. Das Verhältnis liegt in allen Beispielen zwischen 4,0 und 7,8.

Der Wert *LOC Sicht-Spezifikationen* beschreibt die Größe der eigentlichen Sichtspezifikationen, wobei evtl. verwendete Generische Zeichnungen hier nicht mitgezählt werden. In den meisten Fällen besteht der Code zum überwiegenden Teil aus Attributberechnungen und Muster-Anwendungen, jedoch wird hier auch die Beschreibung der Sprachkonstrukt-Leiste und der textuellen Repräsentationen mitgezählt. Wie zu erwarten, trägt dieser Teil am meisten zur Gesamtgröße der Spezifikation bei.

Wie in Abschnitt 5.1.3 beschrieben, ist unter *LOC Generische Zeichnungen* die Anzahl der Sprachkonstrukt-Knoten in Generischen Zeichnungen zu verstehen. Konzeptionell gehören die Werte *LOC Generische Zeichnungen* und *LOC Sichtspezifikationen* sehr eng zusammen, denn Generische Zeichnungen sind prinzipiell lediglich Parametrisierungen des Formular-Musters. Die Besonderheit ist, dass diese Parametrisierungen mit einer Spezialsprache ausgedrückt werden, wohingegen die Parametrisierung anderer Muster direkt durch Attributberechnungen ausgedrückt werden. Das Verhältnis *LOC Sichtspezifikationen* zu *LOC Generische Zeichnungen* ist demnach davon abhängig, wie groß der Anteil des Formular-Musters an der grafischen Repräsentation ist. Man erkennt, dass dieser Wert in Nassi-Shneiderman Diagrammen besonders hoch ist.

LOC Prüfungen und Kopplung umfasst z.B. Konsistenzbedingungen und automatische Anpassungen von Sprachkonstrukten. Hohe Werte sind bei Funktionsaufrufen, mathematischen Formeln und dem *Derivation Tree Assistant* zu finden. Bei Funktionsaufrufen werden die zu den formalen Parametern passenden Platzhalter für aktuelle Parameter automatisch erstellt, bei mathematischen Formeln werden die Zellen von Matrizen an die Zeilen- und Spaltenzahl angepasst und beim *Derivation Tree Assistant* werden nach Änderungen der Grammatikproduktionen die Ableitungsbäume angepasst. In allen Fällen handelt es sich um Zusatzspezifikationen, die das Editieren komfortabler machen oder auf statische Fehler hinweisen. Die Editoren wären aber auch ohne diese Teilspezifikation benutzbar.

LOC Hilfsfunktionen umfasst im Wesentlichen in C oder Tcl implementierte Hilfsfunktionen, die zur Realisierung der grafischen Sichten benötigt werden. Wie zu erkennen, kommen fast alle Spezifikationen ohne solche Hilfsfunktionen aus.

Aus der Summe aller Codezeilen, *LOC Gesamt*, ist zu erkennen, dass die Spezifikationen für diese Beispiele tatsächlich recht klein sind. An dem Verhältnis *LOC Gesamt / LOC Struktur*, das ausgenommen des Funktionsaufruf-Beispiels im Bereich 3,4 bis 6,9 liegt, lässt sich erkennen, dass die Spezifikationsgröße bezogen auf die abstrakte Struktur relativ konstant und damit gut abschätzbar ist. Dies spricht dafür, dass der Spezifikationsansatz für alle vorkommenden Repräsentationsarten gleichermaßen geeignet ist und der Spezifikationsaufwand nur linear bzgl. der Sprachgröße wächst. Der Ausreißer „Funktionsaufrufe“ mit einem Verhältnis von lediglich 2,0 zeigt, dass die Spezialsprache zur Spezifikation für textuelle Repräsentationen diesen Sonderfall tatsächlich besonders kompakt spezifizierbar macht.

Über die Komplexität der Spezifikationen lässt sich anhand der bis jetzt betrachteten Größen noch keine genaue Aussage machen. Während bei Strukturdefinitionen und bei Generischen Zeichnungen die Komplexität sprachbedingt nicht stark variieren kann, können in attributierten Grammatiken in dieser Hinsicht große Unterschiede auftreten. Um zu zeigen, dass in diesem Fall auch die attributierten Grammatiken vergleichsweise einfach und übersichtlich sind, wurde die Anzahl der *nicht-trivialen Attributberechnungen* angegeben. Wie in Abschnitt 5.1.3 beschrieben sind dies Attributberechnungen, die weder ein anderes Attribut kopieren noch einen konstanten Wert zuweisen. Es ist zu erkennen, dass oft wenig bis gar keine nicht-trivialen Attributberechnungen auftreten.

An der Anzahl der nicht-trivialen Attributberechnungen in Generischen Zeichnungen ist zu erkennen, dass diese Spezifikation komplexer ist, als die Größe vermuten lässt. Ein Grund dafür ist, dass hier Darstellungseigenschaften in sehr komplexer Weise von Eigenschaften der Sprachobjekte abhängen, um ihre Bedeutung möglichst gut zu visualisieren.

Da die Komplexität der Spezifikationen relativ vergleichbar ist, scheint es sinnvoll, dendurch das kontrollierte Experiment mit dem vereinfachten NSD-Editor ermittelte Faktor von 1,2 Minuten Spezifikationsaufwand pro LOC auf die anderen Spezifikationen zu übertragen. Nach dieser Rechnung würden z.B. der *Derivation Tree Assistant* knapp 6 Stunden, der Editor für Nassi-Shneiderman Diagramme gut 6 Stunden Entwicklungszeit benötigen, wobei die Zeiten entsprechend der oben geführten Diskussion sehr großzügig geschätzt sind. Diese Größenordnung stimmt auch mit praktischen Erfahrungen überein.

Tabelle 5.7: Code-Wiederverwendung bei Anwendung von visuellen Mustern

Sprache	SK-Klassen	Musteranwendungen	Musteranwendungen / SK-Klassen	Geerbte Rollen	Geerbte Rollen / SK-Klasse	Triviale Attributberechn.	Nichttriv. Attributberechn.	Anteil triv. Attributberechn.
Vereinfachtes NSD-Beispiel	4	7	1,8	20	5,0	10	0	100%
Petri-Netze	4	5	1,3	17	4,3	2	1	60%
Rollen-Diagramme	6	8	1,3	28	4,7	19	14	57%
Funktionsaufrufe	10	16	1,6	76	7,6	0	0	-
Mathematische Formeln	10	11	1,1	28	2,8	19	0	100%
Nassi-Shneiderman Diagramme	11	19	1,7	55	5,0	31	0	100%
Derivation Tree Assistant	12	16	1,3	67	5,6	23	9	72%
Elektronische Schaltungen	21	56	2,7	124	5,9	121	15	89%
Generische Zeichnungen	27	41	1,5	393	14,6	92	48	65%

5.2.7 Wie wirksam sind visuelle Muster?

Code-Wiederverwendung Sehr häufig können fast alle Sprachkonstrukte mit Hilfe visueller Muster spezifiziert werden und häufig brauchen keine Layoutberechnungen überschrieben werden, sondern es genügt, speziell dafür vorgesehene Kontrollattribute zu überschreiben, um die Details der grafischen Repräsentation anzupassen. Tabelle 5.7 belegt diese Aussagen anhand der Beispiel-Implementierungen.

Zur Bestimmung der Anzahl der Musteranwendungen wurde bestimmt, an wie viele Symbole der Repräsentations-Grammatik die Haupt-Rolle eines Musters vererbt wurde. Normalerweise ist die Haupt-Rolle die Wurzel-Rolle des Musters. Eine Ausnahme sind die Varianten des Linien-Musters, bei denen die Rollen `VPConnection`, `VPOrthoConnection` bzw. `VPRefConnection` als Haupt-Rollen definiert wurden. Der Grund dafür ist, dass diese Rollen die Wurzeln der eigentlich dargestellten grafischen Relationen sind, wohingegen die Wurzel-Rolle der Varianten, `VPConnectionArea`

lediglich benötigt wird, um den erlaubten Darstellungsbereich zu definieren. Würde `VPConnectionArea` als Haupt-Rolle des Musters betrachtet, gäbe es z.B. in UML-Klassendiagrammen nur ein Auftreten des Linienmusters, wohingegen es nach der hier verwendeten Definition drei Auftreten, nämlich für Vererbungen, Assoziationen und Abhängigkeiten gibt.

Das Verhältnis zwischen Anzahl Musteranwendungen und Anzahl SK-Klassen liegt jeweils zwischen 1,1 und 1,6, zeigt also, dass Muster häufig und gleichmäßig angewendet werden. Der Wert ist größer als eins, da Muster-Hauptrollen nicht nur an Sprachkonstrukt-Symbole, sondern auch an Attribut-Symbole vererbt werden können. Da jede SK-Klasse im Allgemeinen mehrere Attribute hat, spielt bei dem Verhältnis also u.a. eine Rolle, wie viele Attribute der Klassen zur Darstellung beitragen. Theoretisch könnte auch bei vollständiger Musterabdeckung das Verhältnis zwischen Anzahl Musteranwendungen und Anzahl SK-Klassen unter eins liegen, denn es gibt Muster, deren Anwendung mehrere SK-Klassen umfasst. Aus diesem Grund enthält z.B. der Anwendungsfall mathematische Formeln, in dem das Matrix-Muster benutzt wird, relativ wenige Muster-Anwendungen.

Der Wert *geerbter Rollen* gibt die Anzahl der Muster-Rollen an, die (direkt oder indirekt) an Symbole der Repräsentations-Grammatik vererbt wurden. Der durchschnittliche Wert von 6,5 zeigt, dass die Anzahl der Rollen-Vererbung pro SK-Klasse sehr hoch ist. Der relativ niedrige Wert bei mathematischen Formeln ist damit zu erklären, dass beim Matrix-Muster nur relativ wenige Rollen pro SK-Klasse benötigt werden. Der sehr hohe Wert bei Generischen Zeichnungen kommt hauptsächlich daher, dass der Editor mehrere Sichten auf die abstrakte Struktur bietet.

Die Anzahl der trivialen bzw. nichttrivialen Attributberechnungen charakterisiert die Vollständigkeit der Musteranwendungen und gleichzeitig die Komplexität der Spezifikationen. Bei unvollständiger Musterabdeckung oder beim Überschreiben von Layoutberechnungen müssten unweigerlich nicht-triviale Attributberechnungen verwendet werden. Wie an den Zahlen zu erkennen ist, ist der Anteil der trivialen Berechnungen mit durchschnittlich 70 bis 100 Prozent sehr hoch und deutet damit auf eine niedrige Komplexität und einen hohen Wiederverwendungsgrad hin.

Im Allgemeinen können nichttriviale Attributberechnungen verwendet werden um (1) Kontrollattribute mit dynamisch errechneten Werten zu parametrisieren, (2) Layoutberechnungen von Mustern zu überschreiben oder (3) neue Attribute oder neue Repräsentationen zu berechnen. In den Beispielen

ist der überwiegende Teil der vorhandenen nichttrivialen Attributberechnungen vom ersten Typ, d.h. es werden lediglich Kontrollattribute mit dynamisch errechneten Werten parametrisiert.

Häufigkeitsverteilung Die verfügbaren Muster-Varianten werden keineswegs gleich häufig angewendet. Abbildung 5.6 zeigt die Anwendungshäufigkeit der Muster-Varianten in einer Auswahl der Spezifikationen. Maßgebend war wieder die Haupt-Rolle der jeweiligen Muster-Varianten. Im Diagramm wurden die Muster nach Gesamthäufigkeit sortiert, so dass sich im letzten Diagramm, das die Summe über alle Spezifikationen zeigt, eine abfallende Kurve ergibt.

An der Summe über alle Anwendungsfälle ist zu erkennen, dass die Muster-Varianten für Formulare und Listen ca. 75 Prozent der Musteranwendungen ausmachen. Im Gegensatz dazu werden viel weniger Sprachkonstrukte durch Linien oder komplexere Repräsentationen wie Tabellen, Matrizen oder Bäume visualisiert.

Die steil abfallende Kurve zeigt, dass man mit wenigen Musterimplementierungen bereits einen großen Teil der gewünschten Funktionalität erreichen kann. Einige der seltener angewendeten Muster-Varianten dienen lediglich dazu, den Anwendungskomfort zu steigern, z.B. könnte das Layout von `VPTree` auch durch eine Kombination aus `VPSet` und `VPConnection` realisiert werden. Allerdings ist das spezialisierte Baum-Muster einfacher handhabbar, da es das Layout automatisch berechnet.

Vergleicht man die Häufigkeitsverteilungen der Einzelspezifikationen fällt auf, dass sich das Sprachkonzept stark auf die Verteilung der Musteranwendungen auswirkt. Man erkennt z.B., dass Petri-Netze mengen- und linienorientiert sind, dass die Funktionsaufruf-Sprache auf linearem Textfluss basiert und dass z.B. Nassi-Shneiderman Diagramme und *Streets* sehr stark auf Listen-Strukturen basieren. Allen Spezifikationen ist gemein, dass sie häufig das Formular-Muster einsetzen. Man erkennt auch, dass die Muster immer wieder in anderen Kombinationen auftreten. Dies zeigt, dass die Muster flexibel miteinander kombiniert werden können und dies auch zur Umsetzung der verschiedenartigen Beispiele erforderlich ist.

Flexibilität Die Anwendung einer Muster-Variante schränkt die Freiheiten des Sprachentwicklers prinzipbedingt ein, d.h. bestimmte Darstellungsvari-

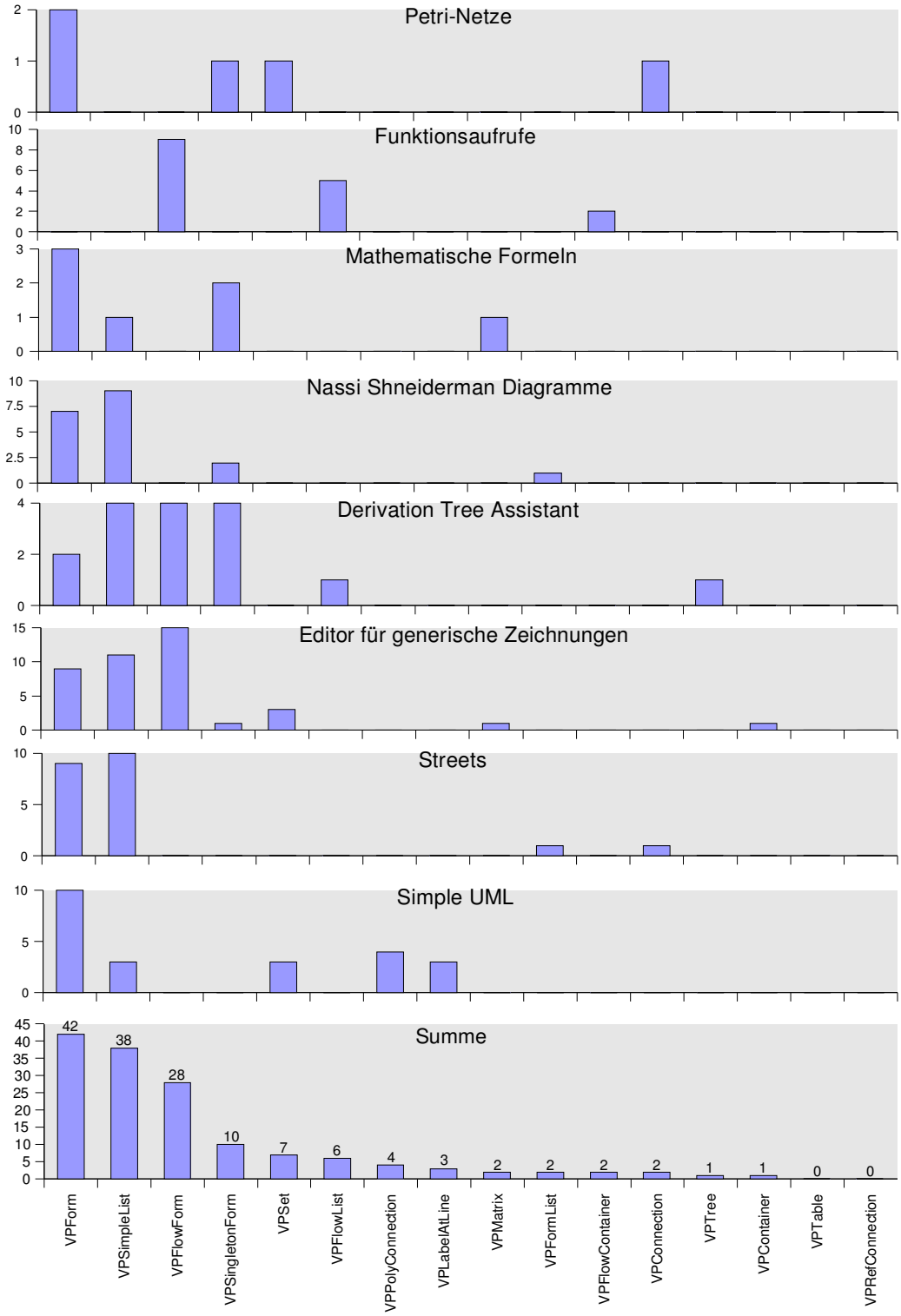


Abbildung 5.6: Anzahl der Muster-Auftreten in verschiedenen Sprachen

Tabelle 5.8: Fragebogen Teil 3: Einschränkung durch visuelle Muster

Frage	Durchschnitt	Standardabw.
Visuelle Muster schränken den Gestaltungsspielraum des Sprachentwicklers ... (1) sehr stark ein ... (5) überhaupt nicht ein	3,5	0,9

anten oder bestimmte Layouteigenschaften sind basierend auf einer gegebenen Muster-Variante überhaupt nicht oder nur sehr schwer zu realisieren. Theoretisch ist es in jedem Fall möglich, eine Muster-Variante durch Hinzufügen und Überschreiben von Berechnungen genau nach Wunsch zu konfigurieren. Im schlimmsten Fall entspräche dies der Neuentwicklung eines Layout-, Repräsentations- und Interaktionskonzepts. Natürlich steht der Nutzen bei diesem Vorgehen nicht in einem guten Verhältnis zu den entstehenden Kosten. Daher entscheiden sich Sprachentwickler in der Praxis häufig dafür, die beste Näherung umzusetzen, die mit Hilfe der verfügbaren Muster einfach realisierbar ist.

Die Frage ist also, wie stark der Sprachentwickler eingeschränkt ist oder sich eingeschränkt fühlt, wenn er (bewusst oder unbewusst) auf die Möglichkeit verzichtet, Teile der Darstellung von Hand zu spezifizieren. Tabelle 5.8 zeigt die Antwort der Testpersonen auf die Frage, wie stark visuelle Muster den Gestaltungsspielraum des Sprachentwicklers einschränken. Die Kandidaten haben die Einschränkung relativ einheitlich als mäßig charakterisiert.

Um die tatsächliche Flexibilität der Muster-Implementierungen genauer zu evaluieren, müssen zunächst zwei Fälle unterschieden werden, nämlich ob der Sprachentwickler eine vorgegebene oder eine selbst entworfene Sprache implementiert.

Für den Fall, dass der Sprachentwickler auch für den Sprachentwurf zuständig ist, kann er bei Bedarf einfach den Sprachentwurf anpassen, so dass dieser besser mit den verfügbaren visuellen Mustern realisierbar ist. Im Fall der Projektgruppe stellt sich also die Frage, ob und in welchem Umfang die Sprache anders ausgefallen wäre, wenn es keine Einschränkungen durch visuelle Muster gegeben hätte.

Leider ist dies schwer zu evaluieren, da der Entwurf einer größeren Spra-

che ein iterativer Prozess ist. Hinzu kommt, dass in der Praxis selbst dem Sprachentwickler das Wechselspiel zwischen Sprachentwurf und technischer Umsetzung nur sehr eingeschränkt bewusst ist. Wenn der Sprachentwerfer beispielsweise stark in visuellen Mustern denkt, kann es vorkommen, dass er im Entwurfsprozess von den verfügbaren visuellen Mustern geleitet wird, so dass er andere Darstellungsformen überhaupt nicht in Betracht zieht. Es bleibt aber festzuhalten, dass die bereits mit DEViL entwickelten Sprachen meiner Ansicht nach eine gesunde und variantenreiche Palette an Sprachkonzepten enthalten, hier also keine Probleme zu erkennen sind.

Für den Fall, dass eine vorgegebene Sprachdefinition umgesetzt werden muss, kann die Flexibilität der Muster besser evaluiert werden. Hier gibt es verschiedene Strategien, um mit Einschränkungen der Muster-Implementierungen umzugehen. Es können (1) Berechnungen überschrieben oder ergänzt werden, es kann (2) auf bestimmte Darstellungs- oder Layoutvarianten, die die Sprachdefinition vorsieht, verzichtet werden, oder es kann (3) von der Sprachdefinition abgewichen werden.

Die erste Strategie äußert sich u.a. durch das Auftreten nichttrivialer Attributberechnungen, die wie bereits oben diskutiert in den Beispielen eher selten vorkommen. Die beiden anderen Strategien sind nachträglich leider nur noch schwer zu entdecken. Die meisten oben eingeführten Beispiele sind zudem für diese Untersuchung ungeeignet, da sie bewusst nur Teile der entsprechenden Sprache realisieren. Mir sind folgende Einschränkungen der Beispielimplementierungen bekannt, die zur Vereinfachung der Implementierung in Kauf genommen wurden.

- In Zustandsdiagrammen lassen sich AND-Superstates nur horizontal oder vertikal nebeneinander anordnen. Die UML-Sprachdefinition erlaubt stattdessen jedoch beliebig angeordnete Regionen, die durch Trennlinien voneinander separiert sind.
- Bei Sequenzdiagrammen wurde durch die Struktur-Modellierung ausgeschlossen, dass Nachrichtenpfeile parallel nebeneinander angeordnet werden können. Dies ist jedoch nur eine Layout-Einschränkung, da mit dem PAR-Konstrukt Parallelität explizit modelliert werden kann.

In beiden Fällen hätten auch andere Muster-Varianten verwendet werden können, die ein flexibleres Layout gestatten. Dagegen sprach jedoch, dass das flexiblere Layout weniger automatisch herzustellen gewesen wäre und sich so die Wartbarkeit visueller Ausdrücke verschlechtert hätte.

Tabelle 5.9: Fragebogen Teil 4: Änderbarkeit der grafischen Repräsentation

Frage	Durchschnitt	Standardabw.
Eine nachträgliche Änderung der visuellen Darstellung ist ... (1) sehr einfach ... (5) sehr aufwändig	2,3	0,5
Eine nachträgliche Änderung der visuellen Darstellung erfordert ... (1) keine Modelländerung ... (5) eine umfassende Modelländerung	2,0	0,7

5.2.8 Wie einfach lässt sich die grafische Repräsentation nachträglich ändern?

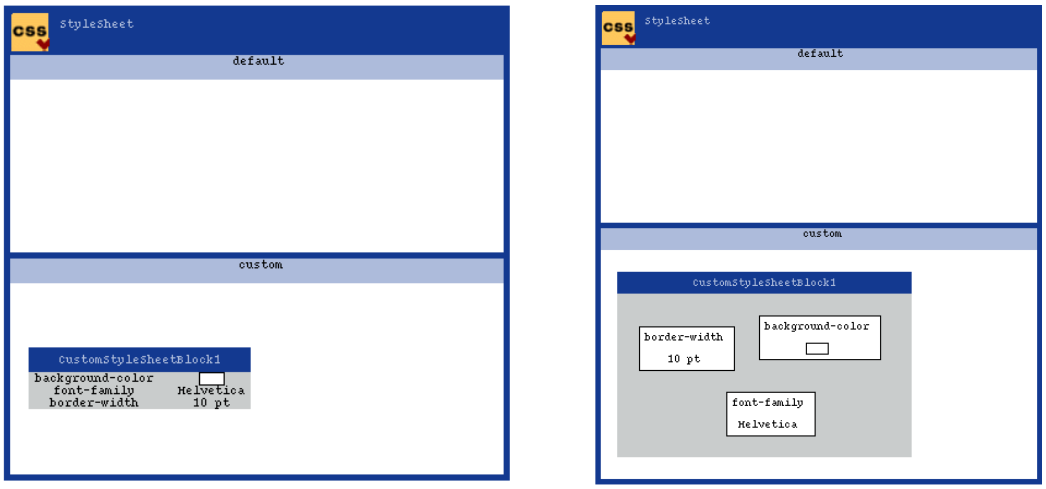
Zum Experimentieren mit alternativen visuellen Repräsentationen ist es wichtig, Sicht-Spezifikationen einfach und schnell ändern zu können. Idealerweise sollten nicht nur Darstellungsdetails, sondern auch grundlegende Darstellungskonzepte schnell geändert werden können.

Da die Test-Gruppe durch ihre Arbeit in der Projektgruppe bereits Erfahrungen mit Sprachänderungen hatte, war es sinnvoll, sie zu diesem Thema zu befragen. Tabelle 5.9 zeigt die Ergebnisse. Die Meinungen zu diesem Themenkomplex sind übereinstimmend recht positiv. Nachträgliche Änderungen der visuellen Darstellung sind nach Aussage der Testkandidaten relativ einfach möglich und erfordern wenig Änderungen der abstrakten Syntax.

Um die einfache Änderbarkeit grafischer Repräsentationen auch praktisch zu verifizieren wurde ein kontrolliertes Experiment durchgeführt, in dem die Kandidaten die visuelle Darstellung einer ihnen bekannten Sprachimplementierung ändern sollten. Die Aufgabenstellung lautete wie folgt.

> Nachfolgend sollen Sie die Stylesheet-Darstellung in PaderWAVE ändern, so dass die Attribute von Stylesheet-Blöcken nicht mehr als Tabelle dargestellt werden, sondern als Menge frei positionierbarer Elemente. Abbildung 5.7 zeigt die alte und neue Darstellungsart.

Wie in Abbildung 5.7a zu erkennen werden Attribute von Stylesheet-Blöcken (z.B. „background-color“) als zweispaltige Tabelle dargestellt. Die Tabelle ist



(a) Alte Darstellungsart (b) Neue Darstellungsart

Abbildung 5.7: Änderung der grafischen Repräsentation von Stylesheet-Blöcken in PaderWAVE

in das Sprachkonstrukt `StylesheetBlock` eingebettet, das wiederum in das `StyleSheet`-Konstrukt eingebettet wird. Die linke Spalte der Tabelle enthält Attributnamen und die rechte Attributwerte. Es existieren spezielle Repräsentationen für einige Attributwerte, z.B. werden Farbwerte durch ein Kästchen dargestellt, das mit der entsprechenden Farbe gefüllt ist. Dies macht deutlich, dass der zu ändernde Sprachteil in eine hinreichend komplexe Umgebung eingebettet ist.

In der neuen Darstellungsart sollten die Attribut/Wert-Paare nicht mehr als Tabellenzeilen, sondern als frei positionierbare Rechtecke innerhalb des `StylesheetBlock`-Konstrukts dargestellt werden. Dazu musste dem Symbol, dem vorher die Rolle `VPTable` zugeordnet war, die Rolle `VPSet` zugeordnet werden. Gleichfalls sind anstatt `VPTableRow` jetzt die Rollen `VPSetElement` und `VPForm` zu vererben. `VPTableCell` wird zu `VPFormElement`. Des Weiteren mussten bestimmte Kontrollattribute angepasst werden und es musste eine Generische Zeichnung hinzugefügt werden, die das Layout der Kästchen beschreibt. Die Syntax der abstrakten Struktur brauchte dagegen nicht geändert zu werden.

Tabelle 5.10 zeigt die Ergebnisse des Experiments. Obwohl einer der Kandidaten diesen Teil der PaderWAVE-Spezifikation nicht kannte, fiel die Einarbeitungszeit mit einem Durchschnitt von 5 Minuten sehr kurz aus. Die eigentliche Arbeitszeit zur Änderung belief sich inklusive Test und Fehlerkorrektur auf durchschnittlich 33 Minuten. Hier war die Varianz relativ hoch. Der

Tabelle 5.10: Messungen im Rahmen der Änderung der grafischen Darstellung von PaderWAVE

Messung	Durchschnitt	Standardabw.
Einarbeitungszeit [min]	5	3
Zeitbedarf zur Änderung [min]	33	14
Gesamtzeit [min]	38	12
Die Aufgabe war ... (1) sehr einfach ... (5) sehr schwer	2,5	0,9
Die Bearbeitung der Aufgabe empfand ich als ... (1) sehr angenehm ... (5) sehr unangenehm	2,5	0,5

schnellste Kandidat hat 10, der langsamste 48 Minuten benötigt. Die Aufgabe wurde insgesamt als mittelschwer und die Bearbeitung als eher angenehm beurteilt. Zusammenfassend zeigt dieses Experiment, dass auch eine tiefer gehende Änderung der visuellen Repräsentation in der Regel sehr lokal und einfach zu bewerkstelligen ist.

5.2.9 Wie gut ist DEViL für große Projekte und Team-Entwicklung geeignet?

Nicht jede Spezifikationsmethode skaliert auch für große Projekte. Hierzu müssen einige Bedingungen erfüllt sein. Zunächst darf der Spezifikationsaufwand für größere Anwendungen nicht überproportional wachsen. Weiterhin muss die Spezifikation wartungsfreundlich sein, d.h. sie sollte auch bei steigender Größe übersichtlich bleiben und alle Arten von Änderungen sollten möglichst lokal durchgeführt werden können.

Ein weiterer Themenkomplex ist der Sprachentwurf. Die Entwicklung einer wirkungsvollen visuellen Sprache ist eine anspruchsvolle Aufgabe, die nicht ad-hoc gelöst werden kann. Ein Generator sollte demnach einen geeigneten Entwicklungsprozess unterstützen und dafür technische Hilfsmittel bieten. Hier sind vor allem inkrementelle und modellbasierte Strategien zu nennen.

Schließlich ist Team-Entwicklung ein wichtiger Aspekt. Hierzu ist es erfor-

Tabelle 5.11: Fragebogen Teil 5: Große Projekte und Team-Entwicklung

Frage	Durchschnitt	Standardabw.
Zur Entwicklung im Team eignet sich DEViL ... (1) sehr gut ... (5) überhaupt nicht	1,8	0,4
Zur inkrementellen Entwicklung eignet sich DEViL ... (1) sehr gut ... (5) überhaupt nicht	2,0	0,0
Eine nachträgliche Änderung des Modells ist ... (1) sehr einfach (5) sehr aufwändig	2,3	1,1
Eine nachträgliche Änderung des Modells erfordert bzgl. der Sicht-Spezifikation ... (1) keine Änderung ... (5) eine umfassende Änderung	3,0	1,0

derlich, dass Schnittstellen definiert und Teilaufgaben separat voneinander bearbeitet werden können.

Benutzerbewertung Die Test-Gruppe hat durch ihre Mitarbeit in der Projektgruppe einige Erfahrung in diesem Themenkomplex gesammelt. Tabelle 5.11 zeigt die Auswertung des entsprechenden Fragebogen-Teils. Einig sind sich die Kandidaten darüber, dass DEViL zur inkrementellen Entwicklung und zur Entwicklung im Team gut geeignet ist. Dabei bewerten die Kandidaten nachträgliche Änderungen an der abstrakten Syntax als einigermaßen aufwändig, wobei bei dieser Beurteilung eine hohe Schwankungsbreite zu beobachten ist.

Skalierung Tabelle 5.12 beschreibt die Komplexität der großen Spezifikationen *Streets*, UML und PaderWAVE. Die Tabelle hat die gleiche Struktur wie Tabelle 5.6 auf Seite 229, die die Komplexität kleiner Spezifikationen beschreibt. Dort ist auch die Bedeutung der Spalten genauer beschrieben. Durch den Vergleich beider Tabellen lässt sich erkennen, dass der Spezifikationsaufwand weitgehend proportional zur Sprachkomplexität ist. Das zeigt sich insbesondere an der Spalte *LOC Gesamt / LOC Struktur*, deren Werte bei *Streets*, UML und PaderWAVE im Bereich von 4,8 bis 6,9 liegen. Sie bewegen sich damit im Rahmen von Tabelle 5.6 (3,4 bis 6,9). Auch die Abweichungen der Re-

Tabelle 5.12: Komplexität großer Spezifikationen

Sprache	SK-Klassen	Abw. Repräsentationsstruktur	LOC Struktur	LOC Sichtspezifikationen	LOC Gen. Zeichnungen	LOC Prüfungen und Kopplung	LOC Hilfsfunktionen	LOC Gesamt	Nichttriv. Attributberechn.	LOC Gesamt / LOC Struktur
Streets	27	2	135	217	238	30	45	665	8	4,9
UML	60	0	341	642	268	384	6	1641	28	4,8
PaderWAVE	92	12	425	1217	525	558	205	2930	24	6,9

präsentationsstruktur und die Anzahl der nichttrivialen Attributberechnungen nimmt bei großen Spezifikationen nicht überproportional zu.

Wartungsfreundlichkeit Wartungsfreundlichkeit umfasst die Übersichtlichkeit und Änderbarkeit von Spezifikationen. Um diese Eigenschaften zu messen, wurde ein kontrolliertes Experiment durchgeführt, in dem die Teilnehmer an einer ihnen unbekannten Spezifikation eine strukturelle Änderung durchführen sollten. Die Aufgabenstellung lautete wie folgt.

> In diesem Experiment sollen Sie eine Ihnen vorher unbekannte Spezifikation ändern. Sie erhalten dazu die Spezifikation eines UML-Editors. Klassendiagramme können dort nur Klassen, aber keine Interfaces enthalten. Ihre Aufgabe ist es, zur vorgegebenen Spezifikation Interfaces hinzuzufügen. Interfaces entsprechen Klassen, haben aber keine Attribute sondern nur Operationen.

Die vorgegebene Spezifikation bestand aus 19 SK-Klassen, enthielt 340 LOC und umfasste visuelle Sichten für Klassendiagramme und Zustandsdiagramme. Um diese Aufgabe zu lösen mussten die Kandidaten eine neue abstrakte und eine neue konkrete Klasse einführen, einige Referenz-Typen ändern

Tabelle 5.13: Messungen im Rahmen der Änderung einer unbekannten Spezifikation

Messung	Durchschnitt	Standardabw.
Einarbeitungszeit [min]	3	2
Zeitbedarf zur Änderung [min]	18	4
Gesamtzeit [min]	22	4
Die Aufgabe war ... (1) sehr einfach ... (5) sehr schwer	2,0	0,7
Die Bearbeitung der Aufgabe empfand ich als ... (1) sehr angenehm ... (5) sehr unangenehm	1,8	0,4

sowie die grafische Darstellung für das hinzugefügte Sprachelement spezifizieren und dazu eine Generische Zeichnung ergänzen. Schließlich musste die Sprachelement-Leiste um einen entsprechenden Knopf ergänzt werden.

Abbildung 5.13 zeigt die Ergebnisse des Experiments. Den Teilnehmern wurden keine Zeitvorgaben zur Einarbeitung gemacht. Die Einarbeitung umfasste soweit notwendig das Verstehen der gegebenen Spezifikation und das Suchen der Stellen, die zu ändern waren. Aus dem durchschnittlichen Zeitbedarf von nur drei Minuten lässt sich schließen, dass auch fremde DEViL-Spezifikationen sehr schnell überschaut werden können.

Der Zeitbedarf für die Änderung ist mit durchschnittlich 18 Minuten ebenfalls relativ niedrig, obwohl sich die Änderung durch viele Spezifikationsaspekte zog und das zu ergänzende Sprachkonstrukt auf relativ komplexe Weise mit anderen Konstrukten in Beziehung steht. Vereinfachend wirkte sich demgegenüber aus, dass das `Interface`-Konstrukt sehr große Ähnlichkeit mit dem `Class`-Konstrukt aufweist, so dass viele Codemuster übernommen werden konnten. Entsprechend wurde die Aufgabe als relativ einfach und die Bearbeitung als relativ angenehm eingestuft.

Die Frage, ob es schwer war, sich in der unbekannten Spezifikation zu orientieren, wurde überwiegend mit „nein“ beantwortet. Die Teilnehmer hatten dafür folgende Argumente.

- DEViL-Spezifikationen sind generell übersichtlich, weil sie eine einheitliche Struktur besitzen.

- Durch „Kopieren und Ändern“ des `Class`-Konstrukts konnte der existierende Code einfach wiederverwendet werden.
- Die vorgegebene Spezifikation war übersichtlich, weil sie genügend klein war.
- Die Sprache UML war den Teilnehmern gut bekannt, was die Einarbeitung erleichterte.

Nach meiner Meinung ergibt sich das gute Abschneiden bei diesem Experiment vor allem daraus, dass die abstrakte Syntax einer Spezifikation vergleichsweise kurz und übersichtlich ist, so dass man sich schnell einen strukturellen Überblick verschaffen kann. Basierend hierauf lassen sich dann leicht die Stellen in den anderen Spezifikationsteilen finden, die zu ändern sind.

Unterstützung des Sprachentwurfs DEViL erlaubt aufgrund seines Spezifikationskonzepts sowohl eine inkrementelle als auch eine modellbasierte Herangehensweise beim Sprachentwurf. Die Möglichkeit zum inkrementellen Sprachentwurf resultiert vor allem daraus, dass sich sowohl die Struktur als auch die visuelle Darstellung sehr leicht erweitern und relativ leicht ändern lässt. Die Möglichkeit zum modellbasierten Sprachentwurf ergibt sich aus der separat spezifizierbaren abstrakten Syntax.

Um gezielt Erfahrungen in diesem Bereich zu sammeln, wurde die Projektgruppe PaderWAVE in der Entwurfs- und Implementierungsphase beobachtet. Sie hat beide Entwurfsmethoden angewendet und kombiniert. Dazu wurde die Projektlaufzeit in zwei Phasen aufgeteilt. In der ersten Phase wurde ein eingeschränktes Prototyp-System entwickelt, in der zweiten Phase wurde basierend auf den Erkenntnissen der ersten Phase der Sprachentwurf überarbeitet und ergänzt. In beiden Phasen wurde zunächst die abstrakte Syntax entworfen und als zentrales Hilfsmittel zur Entwurfskommunikation eingesetzt.

Die Zwei-Phasen Strategie hat sich tatsächlich als sehr hilfreich beim Sprachentwurf erwiesen, da alle Projektgruppenteilnehmer nach meiner Beobachtung nur einen begrenzten Überblick über die Facetten des Entwurfs hatten. Durch den ersten Prototyp konnten Fehlentscheidungen und Unzulänglichkeiten des Entwurfs früh erkannt und korrigiert werden.

Der Prototyp der ersten Phase konnte einigermaßen problemlos als Basis zur Entwicklung des endgültigen Systems wiederverwendet werden. Natürlich

hat die Umstrukturierung der Syntax einigen Aufwand verursacht, da dadurch alle darauf basierenden Spezifikationen angepasst werden mussten. Die Probleme konnten allerdings gut gelöst werden. Ein Nachteil der Vorgehensweise war jedoch, dass die Bereitschaft zur Umstrukturierung der Syntax nicht besonders hoch war, um möglichst Anpassungen der darauf aufbauenden Spezifikationen zu vermeiden. Wäre die endgültige Syntax von Grund auf neu entwickelt worden, hätte eine noch konsistentere Sprache entstehen können.

Zum modellbasierten Sprachentwurf haben sich vor allem zwei Mechanismen in DEViL als nützlich erwiesen: Die automatisch verfügbare Baum-Sicht (siehe Abschnitt 3.2.2), mit der man die abstrakte Struktur bearbeiten kann, ohne dass grafische Sichten verfügbar sind, und die automatische Generierung von Dokumentationen aus der abstrakten Syntax. Dies hat dazu geführt, dass der Sprachentwurf fast übertrieben strukturbasiert war. Der Entwurf der visuellen Darstellung für die abstrakte Struktur wurde in vielen Bereichen fast vollständig der zuständigen Implementierungsgruppe überlassen.

Der Vorteil des modellbasierten Sprachentwurfs war, dass sich die Projektgruppe ganz auf die Struktur und die Semantik konzentrieren konnte. Diese Vorgehensweise ist allerdings nicht grundsätzlich positiv zu bewerten, da beim Sprachentwurf auch die Möglichkeiten und Konsequenzen der grafischen Repräsentation berücksichtigt werden müssen, so dass idealerweise Struktur und Repräsentation gemeinsam entworfen werden.

Team-Entwicklung Zur Team-Entwicklung ist es entscheidend, dass Schnittstellen definiert werden können und gleichzeitig an verschiedenen Teilaufgaben gearbeitet werden kann. Positiv wirkt sich aus, dass in DEViL die Spezifikationsaspekte Struktur, visuelle Repräsentation sowie Analyse und Transformation generell strikt getrennt werden. Die Spezifikationen der visuellen Sichten sowie der Analyse und Transformation sind untereinander vollkommen abhängig.

Die Aufgabenteilung hat in der Projektgruppe gut funktioniert und die abstrakte Syntax diente als angemessene Schnittstelle. Besonders nützlich war die Möglichkeit, Teilspezifikationen zeitweise vollständig auszublenden, solange sie nach Änderungen der abstrakten Syntax noch nicht angepasst waren.

5.2.10 Wie gut lassen sich Sprachen umsetzen, bei denen semantische und editierbare Struktur unterschieden werden müssen?

Die praktische Einsetzbarkeit der Entkopplung von semantischer und editierbarer Struktur wurde am bereits vorgestellten UML-Editor erprobt. Entsprechend der in Abschnitt 3.3.1 beschriebenen Anforderungen kann im spezifizierten UML-Editor ein Klassenmodell durch mehrere zusammengehörende Klassendiagramme modelliert werden, wobei die gleiche Klasse in mehreren Diagrammen oder mehrfach in einem Diagramm vorkommen kann. Alle Klassenrepräsentanten können dabei individuell positioniert werden. Deren Konsistenz bzgl. der Klasseneigenschaften wird durch die Kopplung automatisch sichergestellt. Beim Löschen eines Klassenrepräsentanten hat der Benutzer die Wahl, ob er nur den Repräsentanten oder auch die zugrundeliegende semantische Klasse löschen möchte. Assoziationen und Vererbungsbeziehungen können an beliebigen Klassenrepräsentanten enden und repräsentieren damit eine entsprechende Relation in der semantischen Struktur.

Wie bereits oben aufgeführt hat die Spezifikation des UML-Editors eine Größe von 1641 LOC. Ein äquivalenter UML-Editor ohne getrennte semantische und editierbare Struktur (und damit auch ohne die oben beschriebenen Eigenschaften) hat eine Größe von 1618 LOC, d.h. es werden lediglich 23 Zeilen Zusatzspezifikation benötigt. Der Zusatzaufwand ist im Allgemeinen unabhängig von der Komplexität der Sprache, vorausgesetzt die Anzahl der zu entkoppelnden Sprachkonstrukte ist konstant.

Die Kopplungsmethode wurde so entworfen, dass bei schon existierenden Spezifikationen jederzeit ohne großen Aufwand eine Trennung zwischen semantischer und editierbarer Struktur durchgeführt werden kann. Um dies zu evaluieren, wurde zunächst ein UML-Editor ohne getrennte semantische und editierbare Struktur spezifiziert und die Trennung dann nachträglich hinzugefügt. Die Trennung der Strukturen ließ sich in ca. einer Arbeitsstunde bewerkstelligen. Neben der Ergänzung der oben erwähnten 23 LOC waren auch einige systematische Umbenennungen durchzuführen. Die wichtigste inhaltliche Spezifikation war neben den in Kapitel 3 beschriebenen Angaben die Organisation der einzelnen editierbaren und semantischen Teilstrukturen. Wird z.B. eine neue editierbare Struktur für ein Klassendiagramm erstellt, muss diese basierend auf ihrem Kontext mit einer passenden semantischen Struktur in Beziehung gesetzt werden.

Der Schwierigkeitsgrad und das erforderliche technische Wissen zur Durch-

führung dieser Änderung sind nach meiner Einschätzung höher als bei anderen Spezifikationsaspekten in DEViL. Das liegt vor allem daran, dass die verschiedenen Namensräume für DSSL-Klassen berücksichtigt und sorgfältig unterschieden werden müssen. Aus diesem Grund wurde dieser Aspekt auch nicht im Rahmen der Test-Gruppe evaluiert, denn hierzu wäre eine zusätzliche, umfassende Einarbeitsungsphase der Testpersonen notwendig gewesen.

Zusammenfassend zeigt die prototypische UML-Implementierung, dass das Kopplungsmodell für diesen Anwendungsfall zweckmäßig ist und nur wenig Zusatzspezifikation erfordert. Vor allem für unerfahrene Benutzer ist die richtige Anwendung des Mechanismus allerdings nicht einfach.

Es wurde auch mit Entkopplungen experimentiert, die handgeschriebenen Code erforderlich machten. Die Erfahrungen zeigen, dass der Sprachentwickler bei Verwendung handgeschriebener Kopplungsfunktionen zwar sehr flexibel ist, dass zum richtigen Einsatz aber einige Erfahrung und vor allem die richtige „Denkweise“ gehört. In diesem Punkt noch unerfahrene Nutzer lösen Kopplungsaufgaben häufig „umständlicher“ als notwendig. Hier könnte eine regelbasierte visuelle Sprache helfen, die die Anschauung verbessert und die richtige Denkweise fördert.

5.2.11 Resümee

Zusammenfassend lässt sich sagen, dass die Benutzerfreundlichkeit des Spezifikationsansatzes durchaus positiv zu bewerten ist. Besonders hervorzuheben ist, dass der Generator sowohl zur einfachen Umsetzung kleiner Sprachen als auch zur Entwicklung größerer, komplexer Systeme im Team sinnvoll einsetzbar ist.

Die in DEViL verwendeten Abstraktionen und Spezifikationsmechanismen haben sich meiner Ansicht nach als zweckmäßig erwiesen und die im Vergleich zu VL-Eli vorgenommenen Verallgemeinerungen erhöhen die Flexibilität des Systems.

Unerfahrenen bis durchschnittlichen Benutzern bereitet das derzeitige System noch einige Schwierigkeiten. Einige davon sind sicherlich prinzipbedingt, denn eine genügend mächtige Spezifikationssprache zur Beschreibung visueller Editoren kann nicht beliebig einfach gehalten werden. Andererseits lässt die Kürze und die niedrige Komplexität der Spezifikationen darauf hoffen, dass sich die Anwendbarkeit für Anfänger noch erheblich verbessern lässt.

5.3 Usability der generierten Editoren

Die Usability der generierten Editoren ist ein entscheidender Faktor für die praktische Einsetzbarkeit des Generators, denn schließlich werden Strukturediteren gerade zu dem Zweck entwickelt, Benutzern den Umgang mit einer Sprache zu erleichtern.

Viele Eigenschaften von generierten Editoren ergeben sich den aus Eigenschaften des Generators. Daher lassen sich allgemeine Aussagen über die Usability von generierten Editoren ableiten, obwohl natürlich prinzipiell jeder Editor individuell einer Usability-Evaluation unterzogen werden müsste.

Nachfolgend werden ähnlich wie in Abschnitt 5.2 zunächst die Fragestellungen der Usability-Evaluation auf der Produkt-Ebene diskutiert.

In den Abschnitten 5.3.2 bis 5.3.6 werden dann die durchgeführten Untersuchungen und Experimente geschildert, bevor schließlich in den Abschnitten 5.3.7 bis 5.3.10 auf die Beantwortung der Fragen eingegangen wird.

5.3.1 Zielsetzung

Auch bei der Usability-Evaluation der generierten Editoren ist es im Rahmen dieser Arbeit nicht möglich, alle Aspekte zu untersuchen. Daher habe ich mich auf die folgenden vier Fragen konzentriert.

1. Sind die generierten Editoren einfach bedienbar?
2. Sind die generierten Editoren praxistauglich?
3. Hat die Anwendung visueller Muster positive Auswirkungen auf den Benutzungskomfort?
4. Sind die generierten Editoren ausreichend effizient?

Die erste Fragestellung umfasst die Intuitivität und den mentalen Aufwand zur Durchführung von Editieroperationen. Beides zusammen wirkt sich darauf aus, als wie angenehm der Benutzer den Umgang mit dem Editor empfindet und wie viel Aufmerksamkeit dem Benutzer bleibt, über semantische Aspekte der Sprache nachzudenken.

Der zweite Punkt, die praktische Einsetzbarkeit, geht über die elementare Editierfunktionalität hinaus. Darüber lassen sich erst Aussagen treffen, wenn der

Editor tatsächlich für praktische Anwendungen benutzt wird. Erst hier würde man beispielsweise den *Cut-and-Paste*-Mechanismus vermissen (siehe Abschnitt 3.2.3), oder die Möglichkeit, nach Zeichenketten oder Anwendungen von Definitionen suchen zu können.

Als dritter Punkt wird die Frage diskutiert, ob die Anwendung visueller Muster die Benutzbarkeit eines Editors positiv beeinflusst. In der Theorie lässt sich besonders die Viskosität und die Intuitivität durch spezialisierte Layout- und Interaktionsmechanismen entscheidend verbessern, woraus eine verbesserte Bedienbarkeit im Sinne von Punkt eins resultiert.

Der letzte Punkt schließlich behandelt die Frage, ob aus den Spezifikationen eine genügend effiziente Implementierung generiert werden kann. Hier ist vor allem die Reaktionszeit des Editors entscheidend, denn Benutzer interaktiver Systeme empfinden verzögerte Reaktionen schnell als störend.

Als Zielgruppe der Untersuchung betrachte ich hauptsächlich Sprachbenutzer, die bereits Erfahrung im Umgang mit den generierten Editoren besitzen. Diese Zielgruppe ist besonders wichtig für den realistischen Einsatz generierter Systeme. Auch die Benutzerfreundlichkeit ist für unerfahrene Nutzer wichtig, denn besonders anwendungsspezifische Sprachen werden häufig nur sporadisch eingesetzt. Benutzer erwarten von solchen Sprachen, dass sie ohne große Einarbeitungszeit benutzt werden können. Hierauf werde ich allerdings nicht im Detail eingehen.

5.3.2 Untersuchung 1: Fragebogen

Aus organisatorischen Gründen wurde auch die Evaluation der generierten Editoren von der in Abschnitt 5.2.4 vorgestellten Test-Gruppe durchgeführt, obwohl dazu natürlich keine Kenntnisse zu DEViL notwendig sind. Die Test-Gruppe hatte allerdings den Vorteil, dass sie bereits mit mehreren generierten Editoren vertraut war und zudem die Editoren im Rahmen der Projektgruppenarbeit zur Lösung praktischer Aufgaben verwendet hatte.

Die Kandidaten hatten auf verschiedenen Ebenen mit den generierten Editoren zu tun. Zunächst haben sie im Rahmen der Entwicklung mit DEViL den Editor für Generische Zeichnungen benutzt. Zum anderen mussten sie für den in der Projektgruppe entwickelten Editor PaderWAVE Beispiele entwickeln, um diesen zu evaluieren und zu testen.

Durch einen Fragebogen wurden die Meinungen der Gruppe zur Usability

der generierten Editoren ermittelt. Die Ergebnisse werden in Abschnitt 5.3.7 vorgestellt.

5.3.3 Untersuchung 2: Kontrollierte Experimente mit nachfolgendem Interview

Um quantitative Daten über die Vor- und Nachteile bestimmter Interaktionsmechanismen zu gewinnen, wurden mit der Test-Gruppe insgesamt drei kontrollierte Experimente zur Bedienung der generierten Editoren durchgeführt. In allen Experimenten wurden zwei alternative Interaktions- bzw. Layoutmechanismen gegenübergestellt und der jeweilige Zeitbedarf zur Konstruktion eines visuellen Ausdrucks gemessen.

Um die Reihenfolge der Messungen als Einflussfaktor auszuschließen, wurden diese variiert. Außerdem wurden die Messungen mehrfach durchgeführt, um herauszufinden, ob sich die Verhältnisse bei steigender Vertrautheit mit der Problemstellung ändern.

Vor jeder Aufgabe wurden die relevanten Interaktions- bzw. Layoutmechanismen mit den Kandidaten besprochen und an einem separaten Beispiel demonstriert. Die Kandidaten hatten auch Gelegenheit, die Mechanismen noch einmal praktisch auszuprobieren. Damit sollten die Kandidaten auf einen einheitlichen Wissensstand gebracht werden.

Die Ergebnisse der Untersuchungen werden in den Abschnitten 5.3.7 und 5.3.9 dargestellt.

5.3.4 Untersuchung 3: Einsatz des Editors für Generische Zeichnungen

Die meisten mit DEViL spezifizierten Editoren dienen nur als Demonstrator und werden nicht praktisch eingesetzt. Eine Ausnahme ist der Editor für Generische Zeichnungen, der selbst ein Bestandteil von DEViL ist und sich als wichtiges Spezifikationshilfsmittel herausgestellt hat. Insgesamt wurden bereits über 1.500 LOC Generische Zeichnungen erstellt.

Durch die eigene Benutzung und die Beobachtung anderer Benutzer konnten bereits viele Erkenntnisse zum praktischen Einsatz des Editors gesammelt werden, die teilweise bereits zu Verbesserungen und Ergänzungen des Generators führten. In Abschnitt 5.3.8 wird dies genauer behandelt.

5.3.5 Untersuchung 4: Feature Checkliste und Task-Analyse

Durch eine *Feature Checkliste* kann überprüft werden, ob die generierten Editoren wichtige produktivitätssteigernde oder vom Benutzer erwartete Funktionen enthalten. Durch *Task-Analyse* kann ermittelt werden, welche Schritte für bestimmte Manipulationen an den visuellen Ausdrücken notwendig sind und wie aufwändig sie für den Benutzer sind.

Die Frage ist, wie die zugrunde liegende *Feature Checkliste* ermittelt wird bzw. wie der Aufwand einer Manipulation bewertet wird. Hierzu ist es sinnvoll, von anderen etablierten Systemen auszugehen, die sich bereits als benutzerfreundlich erwiesen haben. Natürlich ist es möglich, dass ein neues System nicht mit einem vorhandenen vergleichbar, aber dennoch benutzerfreundlich ist. Andererseits ist aber die Vergleichbarkeit an sich schon ein Beitrag zur Benutzerfreundlichkeit, denn die Einarbeitungszeit in interaktive Systeme hängt in der Regel stark davon ab, wie sehr die Interaktionsmechanismen mit verwandten Systemen vergleichbar sind, denn meist haben Benutzer bereits einen breiten Erfahrungshintergrund.

Zur Ermittlung der *Feature Checkliste* wurden Editor-ähnliche Systeme wie Textverarbeitungen oder Grafikprogramme auf die verfügbare Standard-Funktionalität untersucht. Wichtige Punkte sind hier *Cut-and-Paste*, Suchen und Ersetzen, Mehrfensterbetrieb oder das Ausdrucken von Dateien. Ein Teil der *Feature Checkliste* stammt allerdings auch aus praktischer Erfahrung mit den Editoren.

Zur Bewertung der Interaktionsmechanismen wurden exemplarisch die Implementierungen einiger Muster mit den entsprechenden Lösungen etablierten Editoren verglichen. Als Vergleichskandidaten wurden stets Systeme ausgewählt, die einen hohen Verbreitungsgrad in ihrem jeweiligen Anwendungsumfeld besitzen.

Die Ergebnisse der Evaluation nach *Feature Checkliste* werden in Abschnitt 5.3.8, die der Muster-Evaluation in Abschnitt 5.3.9 diskutiert.

5.3.6 Untersuchung 5: Performance-Messungen

Ein wesentlicher Faktor für die Benutzbarkeit eines generierten Editors ist die Zeit, die benötigt wird, um nach einer Benutzerinteraktion die grafische Darstellung zu aktualisieren [9]. Ab einer Reaktionszeit von ca. 300 Millise-

kunden ist die Verzögerung für den Benutzer spürbar und bereits Verzögerungszeiten von einer Sekunde können Benutzer als störend empfinden.

Die Reaktionszeit eines von DEViL generierten Editors hängt vor allem vom Zeitaufwand zur Aktualisierung der geöffneten Sichten ab. Der Zeitaufwand zur Aktualisierung einer Sicht hängt wiederum von der Anzahl der Sprachkonstrukte in der Sicht und von der Komplexität des Layouts ab. Besonders zu beachten sind hier benutzerdefinierte Layoutänderungen, die bestimmte Randbedingungen der Darstellung wie Überlappungsfreiheit verletzen. Zur Wiederherstellung solcher Randbedingungen wird häufig zusätzliche Rechenzeit benötigt.

Um trotz dieser Abhängigkeiten halbwegs allgemeingültige Aussagen über die Effizienz der generierten Editoren treffen zu können, wurden die Reaktionszeiten mehrerer Beispieleditoren untersucht, die auf jeweils unterschiedlichen Mustern und Layoutkonzepten basieren. Um die Beispiele reproduzierbar zu machen, wurde jeweils nur eine Sicht geöffnet.

Alle Messungen wurden auf einem AMD Athlon XP 2000+ Prozessor mit 1,66 GHz Taktfrequenz unter Windows XP durchgeführt. Die generierten Editoren wurden im Release-Modus, d.h. ohne programminterne Plausibilitätsprüfungen unter Verwendung von Tcl/Tk 8.3 ausgeführt. Um den Einfluss anderer Prozesse auf die Messung möglichst auszuschließen, wurden die Messungen fünfmal wiederholt und der kleinste Wert verwendet. Die Ergebnisse dieser Untersuchung werden in Abschnitt 5.3.10 diskutiert.

5.3.7 Sind die generierten Editoren einfach bedienbar?

Der elementare Bedienungskomfort hängt stark vom zugrunde liegenden Bedienkonzept ab. Die von DEViL generierten Editoren basieren stark auf den Standard-Interaktionsmechanismen vieler aktueller Produkte, nämlich *direct manipulation*, Selektion und Menüauswahl sowie Eingabedialoge (siehe Abschnitt 4.1.4). In Punkto Vergleichbarkeit sind die generierten Editoren demnach generell positiv zu bewerten. Um dies ein wenig zu konkretisieren werden nachfolgend Benutzerbewertungen und zwei kontrollierte Experimente vorgestellt.

Benutzerbewertung Tabelle 5.14 zeigt die Aussagen der Test-Gruppe zu diesem Themenkomplex. Der obere Teil behandelt allgemeine Fragen bzgl.

Tabelle 5.14: Fragebogen Teil 6: Usability der generierten Editoren

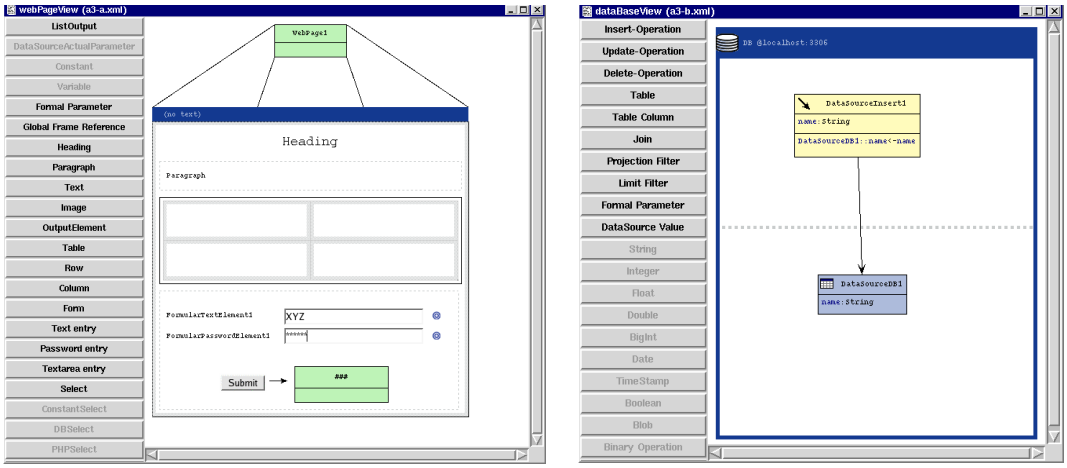
Frage	Durchschnitt	Standardabw.
Generierte Editoren sind ... (1) sehr einfach zu bedienen ... (5) sehr schwer zu bedienen	1,3	0,4
Den Umgang mit generierten Editoren empfinde ich als ... (1) sehr angenehm ... (5) sehr unangenehm	1,5	0,9
Generische Zeichnungen sind ... (1) leicht anzuwenden ... (5) schwer anzuwenden	1,0	0,0
Den Umgang mit Generischen Zeichnungen empfinde ich als ... (1) sehr angenehm ... (5) sehr unangenehm	1,0	0,0

der Benutzbarkeit der Editoren. Die Benutzer bezeichnen die Editoren hier übereinstimmend als angenehm und einfach zu bedienen.

Der untere Teil der Tabelle 5.14 enthält die Ergebnisse, die bereits in Abschnitt 5.2.6 im Kontext der Usability von DEViL vorgestellt wurden. An den Antworten kann auch die Benutzbarkeit des Editors für Generische Zeichnungen abgelesen werden. Wäre der Editor schwer oder umständlich zu benutzen, hätte dies sicherlich Einfluss auf die Bewertung gehabt. Er ist jedoch ohne eine einzige Ausnahme als sehr leicht anzuwenden und als sehr angenehm in der Benutzung bewertet worden.

Beiträge des Direct Manipulation Konzepts Ein wesentlicher Interaktionsmechanismus der generierten Editoren ist *direct manipulation*. Um die positiven Auswirkungen dieser Methode zu zeigen, wurden zwei kontrollierte Experimente durchgeführt. In beiden Experimenten sollte die gleiche Aufgabe einmal mit und einmal ohne *direct manipulation* gelöst werden.

Im ersten Experiment sollte eine Programmstruktur visuell konstruiert werden. Bei der *direct manipulation* Variante wird dazu ein Knopf in der Sprachelement-Leiste betätig und das entsprechende Sprachkonstrukt durch Auswahl einer passenden Einfügestelle in das bereits konstruierte Programm eingefügt (siehe Abschnitt 4.1.4). Bei der alternativen Editiervariante werden neue Sprachkonstrukte durch das Kontextmenü bereits existierender Sprachkonstrukte erzeugt.



(a) WebPage-Sicht (b) Datenbank-Sicht

Abbildung 5.8: Zu konstruierende visuelle Ausdrücke

Das Experiment wurde mit zwei verschiedenen Sichten des PaderWAVE Editors durchgeführt, nämlich mit der WebPage-Sicht (25 Sprachkonstrukt-Knöpfe) und der Datenbank-Sicht (27 Sprachkonstrukt-Knöpfe). Die zu konstruierenden Ausdrücke sind in Abbildung 5.8 zu sehen. Der oberen Teil von Tabelle 5.15 zeigt das Ergebnis der Messung. Man erkennt, dass der Geschwindigkeitsvorteil in beiden Fällen unterschiedlich groß ausfällt, aber mit ca. 65 bzw. 25 Prozent deutlich ist. Zwischen dem ersten und dem zweiten Durchgang ist eine wesentliche Leistungssteigerung der Kandidaten zu erkennen, am Verhältnis der Leistungen ändert sich aber dadurch nichts.

In einem weiteren Experiment wurde untersucht, wie schnell Anwendungen von Definitionen auf verschiedene Weisen konstruiert werden können. Abbildung 5.9 zeigt den Ausgangszustand der Aufgabe. Die noch offenen aktuellen Parameter für den Aufruf von `WebPage2` sollten mit Anwendungen der Eingabefelder `a` bis `e` belegt werden. Bei der *Drag-and-Drop* Methode müssen dazu lediglich die entsprechenden Eingabefelder auf die Platzhalter für die aktuellen Parameter gezogen werden. Bei der herkömmlichen Methode muss mittels des Knopfes `Variable` in der Werkzeugleiste ein entsprechendes Sprachkonstrukt erstellt werden und dann per Dialogsicht der entsprechende Bezug zum Eingabefeld hergestellt werden.

Der untere Teil von Tabelle 5.15 zeigt, dass durch die *Drag-and-Drop* Variante ein Geschwindigkeitsvorteil von ca. 50 Prozent erreicht werden kann. Natürlich muss man berücksichtigen, dass durch dieses Beispiel der maximal erreichbare Geschwindigkeitsvorteil ermittelt wird.

Tabelle 5.15: Einfluss des Direct Manipulation Konzepts auf die Editiergeschwindigkeit

Aufgabe	Zeitbedarf Direct Manipulation [sec]	Standardabw.	Zeitbedarf dialogbasiert [sec]	Standardabw.	Ersparnis Direct Manipulation	Standardabw.
Konstruktion WebPage-Sicht (1. Durchgang)	25	4	75	28	62%	15%
Konstruktion WebPage-Sicht (2. Durchgang)	14	2	40	6	63%	9%
Konstruktion Datenbank-Sicht (1. Durchgang)	60	24	76	15	19%	33%
Konstruktion Datenbank-Sicht (2. Durchgang)	33	6	45	7	25%	5%
Konstruktion von Referenzen (1. Durchgang)	13	4	27	2	53%	11%
Konstruktion von Referenzen (2. Durchgang)	13	3	23	2	42%	16%

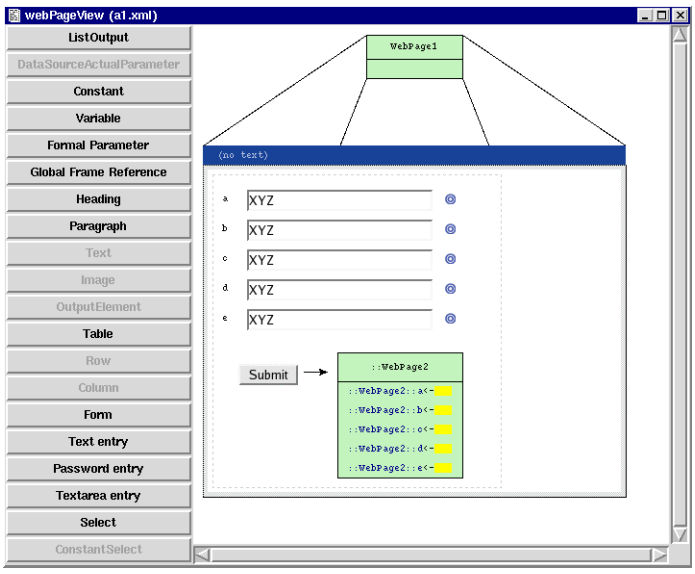


Abbildung 5.9: Ausgangssituation zur Konstruktion von Variablenanwendungen

Generell ist bei der Interpretation von Tabelle 5.15 zu beachten, dass unter *Ersparnis Direct Manipulation* die Mittelwerte der Ersparnisse der Einzelversuche eingetragen sind. Diese Werte stimmen nicht unbedingt mit der Ersparnis bzgl. der Durchschnittszeiten überein, da die Berechnung aufgrund der Quotientenbildung nicht linear ist. Der Unterschied ist umso größer, je größer die Standardabweichung des dialogbasierten Zeitbedarfs ist. Da dieser aber in fast allen Fällen klein ist, ergeben sich ähnliche Prozentwerte.

5.3.8 Sind die generierten Editoren praxistauglich?

Auch wenn mit einem Editor visuelle Ausdrücke einfach erstellt werden können heißt das noch nicht, dass er auch für die praktische Anwendung geeignet ist. Bestimmte Anforderungen zeigen sich erst dann, wenn die Editoren tatsächlich für realistische Arbeiten eingesetzt werden.

Betrachtet man andere etablierte Strukturediteren und Editor-ähnliche Systeme, lassen sich bestimmte Funktionen identifizieren, die offenbar wichtig für den praktischen Einsatz sind. Hierzu zählen

- Laden und Speichern von Dateien
- Suchen und Ersetzen
- Cut-and-Paste
- Undo-Funktionalität
- gleichzeitiges Bearbeiten mehrerer Dateien
- Ausdrucken von Dokumenten
- Shortcuts und Tastaturnavigation

Mit DEViL bzw. dessen Vorgänger VL-Eli wurden bereits vollständige Systeme für den praktischen Einsatz entwickelt. Neben dem Editor für Generische Zeichnungen ist hier das System *SIMtellignce Designer/J* [53] zu nennen, das mit dem Vorgänger System VL-Eli für die ORGA Kartensysteme GmbH entwickelt wurde. Mit beiden Editoren wurden bereits wichtige Erfahrungen in Punkto Praxistauglichkeit gesammelt. Viele dieser Erfahrungen sind bereits in den Generator eingeflossen und führten dazu, dass die meisten der oben genannten Punkte bereits angemessen unterstützt werden.

Bzgl. des Ladens und Speicherns war eine wichtige Erkenntnis, dass die verwendete XML-Repräsentation möglichst stabil sein sollte, d.h. sie sollte sich nach Editieroperationen möglichst wenig ändern. Beispielsweise sollte sich die Reihenfolge oder die interne ID bestimmter Elemente nicht grundlos ändern. Dies ist wichtig, wenn die XML-Dateien durch Versionsverwaltungssysteme wie CVS verwaltet werden.

Bei der Such-Funktionalität hat sich besonders das Suchen nach Anwendungen von Definitionen und das Suchen von Programmobjekten mit bestimmten Namen als sehr wichtig erwiesen. Beide Funktionen werden regelmäßig benötigt, sobald mit größeren, evtl. kooperativ erstellten Spezifikationen gearbeitet wird.

Die Funktionen *Cut-and-Paste* und *Undo* werden von vielen Benutzern grundsätzlich erwartet. *Cut-and-Paste* ist bei Strukturediteren nützlich, um bereits konstruierte Strukturen modifiziert wiederverwenden zu können.

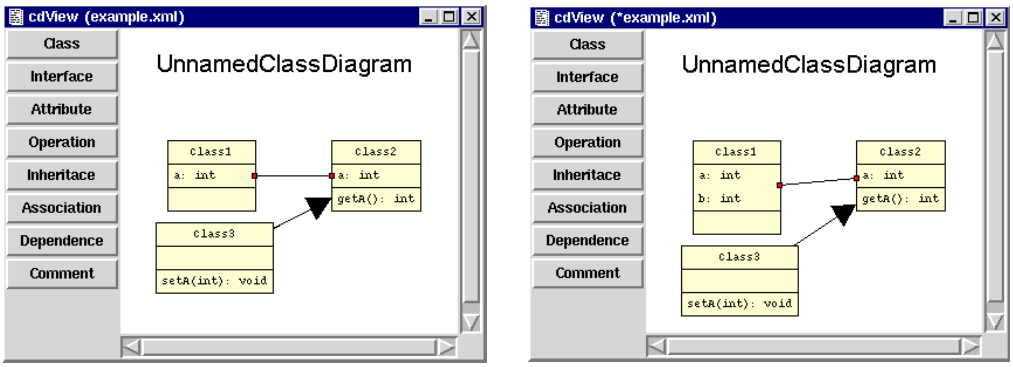
Das gleichzeitige Öffnen mehrerer Dateien ist wichtig, um Teilstrukturen unterschiedlicher Dateien kombinieren oder um mit verschiedenen Versionen eines Programms hantieren zu können. Besonders wichtig ist hier, dass auch das *Cut-and-Paste* über Dateigrenzen hinweg funktioniert.

All diese Funktionen werden ohne Zutun des Sprachentwicklers automatisch generiert. Sicherlich könnte die Praxistauglichkeit noch durch weitere Zusatzfunktionen verbessert werden. Entscheidend ist jedoch, dass die genannten Grundfunktionen im Gegensatz zu Handimplementierungen nicht „vergessen“ werden können, was eine erhebliche Einschränkung der Anwendbarkeit einer Sprachimplementierung bedeuten würde.

5.3.9 Hat die Anwendung visueller Muster positive Auswirkungen auf den Benutzungskomfort?

Visuelle Muster können maßgeschneiderte Layout- und Interaktionsmechanismen kapseln und bieten somit das Potenzial, die Editoren besonders benutzerfreundlich zu machen. Für zwei Musterimplementierungen wurde das genauer untersucht.

Einfügen von Listenelementen Listenelemente werden eingefügt, indem in der Sprachelement-Leiste ein Sprachelement ausgewählt und dann eine



(a) Ausgangszustand (b) Zustand Nach Einfügen von `Class1.b`

Abbildung 5.10: Beispiel für die Nicht-Überlappungs-Korrektur

passende Einfügestelle selektiert wird. Nach der Auswahl des Sprachelements wird automatisch die dem Mauszeiger nächstgelegene passende Einfügestelle grafisch hervorgehoben (siehe Abbildung 4.6 auf Seite 140). Die Einfügeoperation wird mit einem einfachen Mausklick abgeschlossen. Demnach besteht die Interaktion aus zwei Mauspositionierungen und zwei Klicks, wobei die Trefferfläche bei der zweiten Mauspositionierung je nach Ausprägung recht groß ist.

Der Interaktionsmechanismus lässt sich auch in vielen anderen grafischen Oberflächen finden, z.B. in *Mozilla Firefox* zur Manipulation der Lesezeichen, in *Microsoft Windows* zum Einfügen von Programmen in die Schnellstart-Liste oder in *OpenOffice Impress* zur Manipulation der Folien-Liste. Daher ist er vielen Nutzern bereits bekannt und leistet somit einen positiven Beitrag zur Benutzerfreundlichkeit.

Überlappungsfreiheit beim Mengen-Muster Bei Instanzen des Mengen-Musters wird standardmäßig die Überlappung von Mengenelementen automatisch verhindert. Ggf. werden dazu die Positionen von Mengenelementen dauerhaft verändert. Abbildung 5.10 zeigt ein Beispiel für diese Funktion. Wird der Klasse `Class1` das Attribut `b` hinzugefügt, ändert sich damit die Größe dieser Klasse. Um eine Überlappung mit `Class3` zu verhindern, wird letztere automatisch ein Stück nach unten verschoben. Der Benutzer braucht somit keine zusätzliche Aktion durchzuführen, um ein akzeptables Layout zu erreichen.

Erstaunlicherweise findet man eine entsprechende Funktionalität kaum in kommerziellen Systemen. Ohne diesen Mechanismus kann es jedoch leicht zu

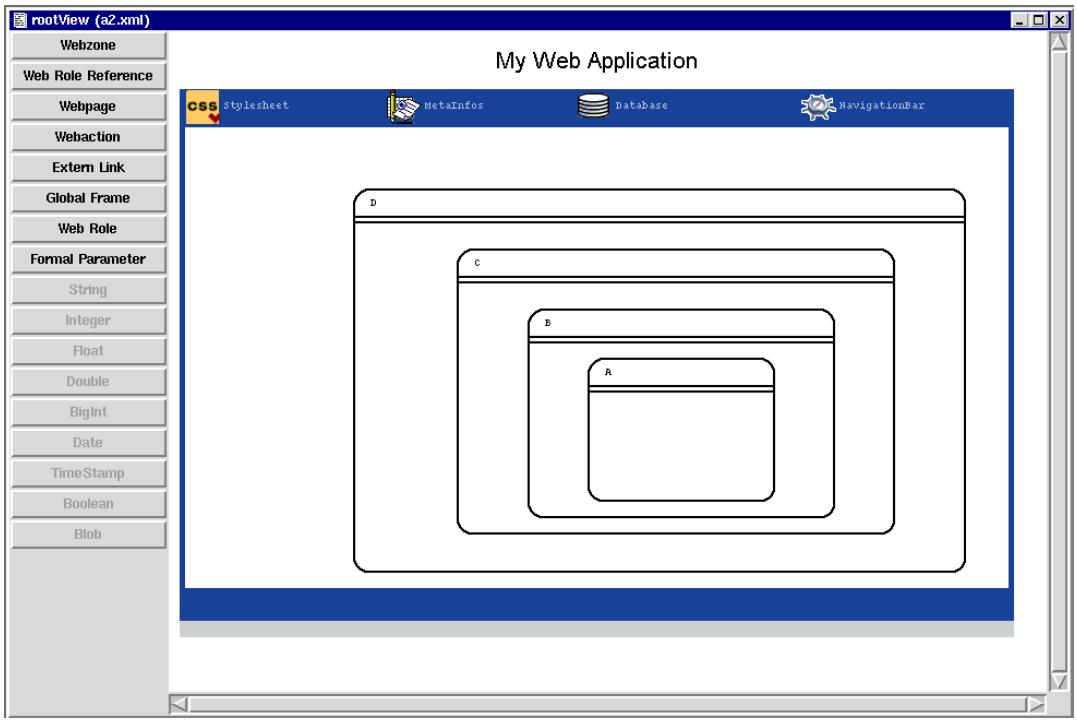


Abbildung 5.11: Ausgangssituation des Experiments zur Evaluation der Nicht-Überlappungs-Korrektur von Listenelementen

missverständlichen oder mehrdeutigen Repräsentationen kommen. Beispielsweise kann ein Sprachelement vollständig von einem anderen überdeckt werden, so dass man überhaupt nicht sieht, dass es vorhanden ist.

In einem kontrollierten Experiment wurde untersucht, ob die Nicht-Überlappungs-Korrektur die Benutzer-Effizienz beim Editieren beeinflusst. Dazu sollten die Testkandidaten die Schachtelungsfolge der in in Abbildung 5.11 gezeigten Objekte umkehren. Die Schachtelungsfolge D, C, B, A sollte also in die Schachtelungsfolge A, B, C, D überführt werden. Die Kandidaten hatten diese Aufgabe je zweimal mit und zweimal ohne Nicht-Überlappungs-Korrektur zu bearbeiten.

Das Ergebnis ist in Tabelle 5.16 dargestellt. Wie zu erkennen ist, ergab sich keine signifikante Wirkung der Nicht-Überlappungs-Korrektur auf die Bearbeitungszeit. Die durchschnittliche Bearbeitungszeit ist annähernd gleich und die durchschnittliche Zeitersparnis durch die Nicht-Überlappungs-Korrektur ist sogar negativ. In den Einzelexperimenten zeigte sich teilweise eine Verbesserung und teilweise eine Verschlechterung, was auch an der großen Standardabweichung abzulesen ist.

Tabelle 5.16: Einfluss der Nicht-Überlappungs-Korrektur auf die Editiergeschwindigkeit

Aufgabe	Zeitbedarf mit Nicht-Überlappungs-Korrektur [sec]	Standardabw.	Zeitbedarf ohne Nicht-Überlappungs-Korrektur [sec]	Standardabw.	Ersparnis durch Nicht-Überlappungs-Korrektur	Standardabw.
1. Durchgang	46	6	45	24	-30%	61%
2. Durchgang	36	10	34	13	-23%	53%

5.3.10 Sind die generierten Editoren ausreichend effizient?

Um den unterschiedlichen Aufwand beim Layout verschiedenartiger Repräsentationen zu berücksichtigen, wurde die Reaktionszeit verschiedener Beispieleditoren gemessen. Die Messdaten sind in Tabelle 5.17 aufgeführt. Die technischen Randbedingungen wurden in Abschnitt 5.3.6 erläutert. Die Spalte *SK-Knoten* gibt die Anzahl der Sprachkonstrukt-Knoten der zu aktualisierenden Sicht an. Die Spalte *Akt.zeit* bezeichnet die benötigte Aktualisierungszeit in Millisekunden. Die letzte Spalte enthält schließlich das Verhältnis der Aktualisierungszeit zu der Anzahl der Sprachkonstrukt-Klassen.

Die obere Hälfte der Tabelle repräsentiert den „Normalfall“, bei dem die grafische Darstellung direkt durch Attributberechnungen berechnet wird. Die untere Tabellenhälfte zeigt Ausnahmen, bei denen aufwändige Layoutberechnungen durchgeführt werden, so dass der Quotient in der letzten Spalte signifikant abweicht. Hierauf gehe ich später gesondert ein.

Zunächst zeigt Tabelle 5.17, dass die Reaktionszeit bei Sichtgrößen mit einigen hundert SK-Knoten akzeptabel ist. Dies genügt häufig auch für große, realistische Anwendungen, da in der Praxis größere Programme normalerweise auf mehrere Sichten verteilt werden, damit die Darstellung nicht unübersichtlich wird. Da die Implementierung nicht auf Geschwindigkeit optimiert ist und teilweise auf der Scriptsprache Tcl basiert, ließen sich die Reaktionszeiten zudem noch erheblich verbessern.

Am relativ konstanten Verhältnis zwischen Aktualisierungszeit und Anzahl der Sprachkontrukte in der oberen Tabellenhälfte erkennt man, dass die Aktualisierungszeit gut einschätzbar und weitgehend proportional zur struk-

Tabelle 5.17: Dauer der Sicht-Aktualisierung nach Programmänderungen

Sprache	SK-Knoten	Akt.zeit	Akt.zeit / SK-Knoten
Petri-Netze (Abb. 5.2a)	22	77	3,5
Mathematische Formeln (Abb. 5.2d)	18	81	4,5
NSD (Abb. 5.2e)	8	36	4,5
NSD (50 geschachtelte Schleifen)	50	274	5,4
Elektronische Schaltungen (Abb. 5.2g)	55	188	3,4
Zustandsdiagramme (Abb. 5.12)	104	380	3,7
Zustandsdiagramme (Abb. 5.12)	104	725	7,0
Rollen-Diagramme (Abb. 5.2b)	12	209	17,4

turellen Größe der Sicht ist. Der Berechnungsaufwand für unterschiedlichen Layoutarten (freie Positionierbarkeit von Elementen einerseits und geschachtelte größenoptimierte Darstellungen andererseits) ist ungefähr vergleichbar. Diese Ergebnisse entsprechen weitgehend den Messungen von Jung zur Evaluation des VL-Eli Systems [28]. Die Werte in Tabelle 5.17 sind ca. um den Faktor 1,7 kleiner als die Ergebnisse von Jung, aber er hat auch einen langsameren Prozessor (Pentium III mit 450 MHz) verwendet. Seit dieser Messung wurde auch zur Verbesserung der Wartbarkeit eine zusätzliche Abstraktionsschicht zur verwendeten Basis-Bilbilothek für die grafische Darstellung hinzugefügt, die den leichten Effizienzverlust erklärt.

Die Aktualisierungszeiten in der unteren Tabellenhälfte spiegeln einige anspruchsvollere Layoutberechnungen wider. Die Messung zur Aktualisierung eines Zustandsdiagramms in der unteren Hälfte basierte im Gegensatz zu den Messungen in der oberen Hälfte auf einer Änderung, die zur Überlappung zweier Zustände führt. Die automatische Korrektur solcher verbotener Überlappungen erfordert zusätzliche Rechenzeit. Wie zu erkennen ist, liegt die Aktualisierung aber auch in diesem Fall noch im akzeptablen Bereich.

Das verwendete Beispiel für Zustandsdiagramme ist dem von Jung verwendeten Beispiel zur Evaluation von VL-Eli nachempfunden. Wenn Constraints verletzt wurden, dauert die Aktualisierung in VL-Eli teilweise mehr als 3 Sekunden, wobei dieser Wert noch dazu in hohem Maße schwankt. Bei noch größeren Zustandsdiagrammen können dort die Constraints überhaupt nicht mehr in angemessener Zeit gelöst werden. Der Unterschied kommt dadurch zustande, dass VL-Eli einen allgemeinen Constraintsolver verwendet, während DEViL einen spezialisierten direkt implementierten Algorithmus zur Korrektur von Überlappungen benutzt.

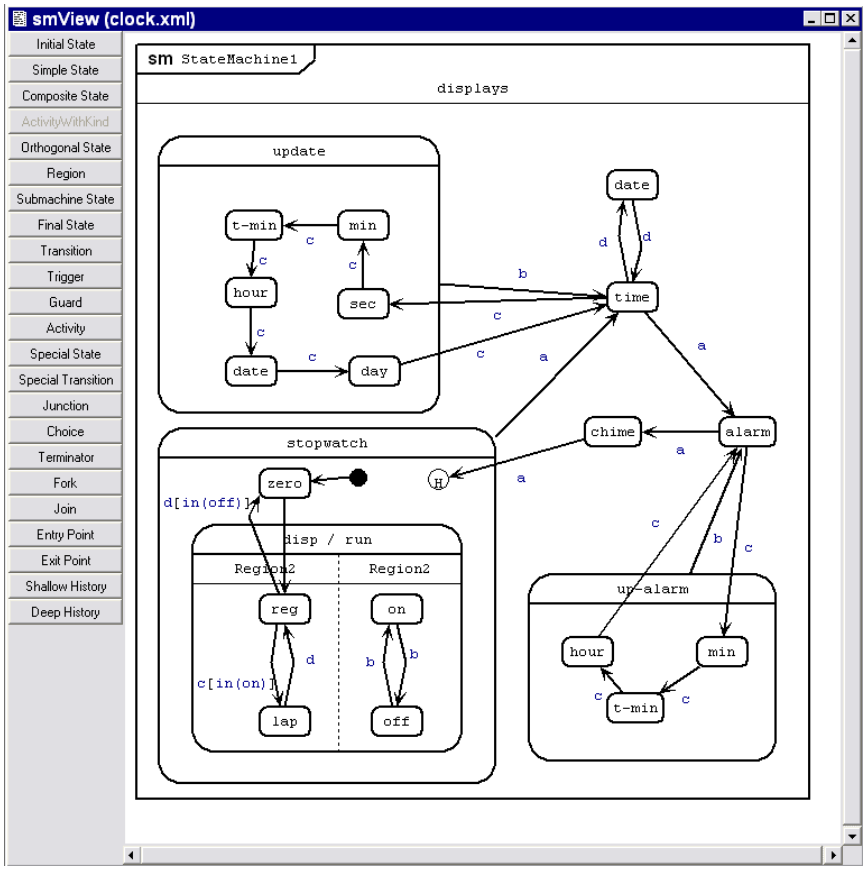


Abbildung 5.12: Zustandsdiagramm zum Vergleich der Editor-Effizienz mit VL-Eli

Die in Tabelle 5.17 zuletzt aufgeführte Messung ist ein weiterer Sonderfall, weil Rollen-Diagramme auf der Muster-Variante `VPTree` basieren, das das Graph-Layoutsystem *dot* [16] einsetzt. Da die Layoutberechnung eigentlich für allgemeine Graphen gedacht ist und auf einem mehrstufigen Verfahren basiert, ist sie prinzipiell aufwändiger. Hinzu kommt, dass *dot* der Einfachheit halber als separater Prozess ausgeführt wird, wodurch zusätzliche Kommunikationskosten entstehen. Die Toleranzgrenze wird aber selbst in diesem Beispiel erst bei einer Sichtgröße von ca. 100 SK-Knoten erreicht.

Aufwand der strukturellen Kopplung Die oben gemessenen Reaktionszeiten sind weitgehend proportional zur Größe der geöffneten Sichten. Wenn getrennte semantische und editierbare Strukturen verwendet werden, kommt eine weitere Verzögerung durch die strukturelle Kopplung hinzu. Im Gegensatz zur Sichtberechnung wird die strukturelle Kopplung normalerweise auch dann ausgeführt, wenn die entsprechende Sicht geschlossen ist. Prinzipiell könnte die strukturelle Kopplung zwar auf geöffnete Sichten beschränkt werden. Daraus würden sich aber zusätzlich Anforderungen an die Reihenfolge der strukturellen Anpassungen der abgeleiteten Sicht ergeben. Dies ist wichtig, weil im Allgemeinen eine sofortige strukturelle Anpassung nach jeder elementaren Änderung zu einem anderen Ergebnis führt, als wenn die strukturelle Anpassung erst nach einer Folge von Änderungen durchgeführt wird. Daher ist es konzeptionell sauberer, die Kopplung unabhängig vom Darstellungszustand der Sichten durchzuführen.

Wie in Tabelle 3.1 auf Seite 100 zu erkennen, hängen die Konsistenzbedingungen der vordefinierten Anpassungsschemata nur von lokalen Eigenschaften der gekoppelten Knoten ab, so dass die Kopplung ereignisbasiert erfolgen kann. Auf diese Weise muss bei jeder Änderungsoperation nur eine kleine Anzahl von Konsistenzüberprüfungen durchgeführt werden, wodurch der hierzu benötigte Zeitaufwand im Vergleich zur Sicht-Berechnung vernachlässigbar klein wird. Werden handimplementierte Funktionen zur Kopplung verwendet, erfolgt diese aber in der aktuellen Implementierung der Einfachheit halber nicht ereignisbasiert. Der Aufwand ist daher proportional zur Größe der abgeleiteten Struktur. Da die Spezifikation noch dazu auf einer Scriptsprache basiert, ergeben sich hierdurch beträchtliche zusätzliche Verzögerungen, die die Größe der Gesamtstruktur beschränken. Basierend auf einer regelbasierten Spezifikationssprache für benutzerdefinierte Kopplungen könnten aber auch diese ohne Zusatzaufwand für den Sprachentwickler ereignisbasiert durchgeführt werden.

5.3.11 Resümee

Ohne direkten Vergleich zu anderen Sprachimplementierungen kann natürlich nicht grundsätzlich behauptet werden, dass die von DEViL generierten Editoren uneingeschränkt benutzerfreundlich sind. Tatsächlich gibt es noch viele Details, die verbessert werden müssten, wenn die Editoren mit kommerziellen Systemen konkurrieren sollten.

Trotzdem hat die Evaluation gezeigt, dass das Spezifikationskonzept und das grundsätzliche Interaktionsprinzip der generierten Editoren das Potenzial hat, sehr benutzerfreundliche Systeme zu liefern. Hierzu tragen vor allem die grundlegenden Interaktionstechniken *direct manipulation* und *Drag-and-Drop* und nicht zuletzt die visuellen Muster bei, durch die spezielles Expertenwissen und bereits etablierte Techniken bzgl. Interaktion und Layout fast zum Nulltarif verwendet werden können.

Benutzer empfinden die generierten Editoren als angenehm und durch die Spezialimplementierungen der visuellen Muster sind die Editieroperationen mit anderen Systemen vergleichbar und somit relativ leicht zu erlernen.

Ein wichtiger Punkt ist ferner, dass grundlegende Basisfunktionalitäten wie das Suchen nach Anwendungen von Definitionen oder das Suchen nach benannten Objekten ohne Zutun des Sprachentwicklers automatisch generiert werden können, was für die Praxistauglichkeit des Systems von entscheidender Bedeutung ist.

5.4 Verwandte Arbeiten

Generator-Ebene Obwohl es viele Generatoren zur Implementierung visueller Sprachen gibt, sind Untersuchungen ihrer Benutzerfreundlichkeit kaum zu finden. Mir ist keine Evaluation bekannt, die mit der in Abschnitt 5.2 vergleichbar ist. Zumindest finden sich in einigen Publikationen vereinzelte Aussagen zu diesem Thema. So lässt sich in [37, S. 240] nachlesen, dass mit DiaGen II ein Nassi-Shneiderman Editor in weniger als einem Tag realisiert werden kann. An gleicher Stelle liest man auch, dass in DiaGen II bestimmte frühzeitig zu fällende Entwurfsentscheidungen maßgeblich für das Gesamtergebnis sind. Als Beispiel wird die Entscheidung genannt, welcher Anteil der Diagrammanalyse in der lexikalischen und welcher in der syntaktischen Analyse durchgeführt wird.

Die Größenordnung des Spezifikationsaufwands ist mit dem von DEViL vergleichbar. Der zweite Punkt lässt sich aufgrund des unterschiedlichen Spezifikationskonzepts nicht direkt mit DEViL vergleichen.

Produkt-Ebene Visuelle Editoren werden häufiger auf ihre Benutzerfreundlichkeit untersucht. Die Reaktionszeit generierter Editoren wurde, wie oben bereits erwähnt, auch von Jung [28] evaluiert. Die Ergebnisse entsprechen aufgrund des ähnlichen Ansatzes weitgehend den hier vorgestellten Ergebnissen. Jung hat zusätzlich untersucht, welchen Anteil die verschiedenen Verarbeitungsschritte an der Reaktionszeit haben. Hier stellt sich heraus, dass das generische Basismodul zur Verwaltung der grafischen Darstellung mit ca. 60 bis 70 Prozent einen hohen Anteil an der Gesamtlaufzeit hat. Bei größeren Repräsentationen wird allerdings die Verarbeitung der grafischen Constraints zum maßgeblichen Faktor.

Minas [37, S.227] untersucht den Zeitbedarf zum Parsieren und zum constraintbasierten Layout in DiaGen II. Zum Parsieren von Nassi-Shneiderman Diagrammen mit einer Größe bis ca. 150 Konstrukten wird weniger als eine Sekunde benötigt. Die Größenordnung verarbeitbarer Programme ist somit mit der von DEViL vergleichbar. Auch Minas identifiziert das constraintbasierte Layout als beschränkenden Faktor des Systems. Selbst bei unter 30 Sprachkonstrukten dauert das constraintbasierte Layout bereits über eine Sekunde.

Zur allgemeinen Evaluation visueller Sprachen und Sprachimplementierungen wurden verschiedene Modelle vorgeschlagen. Green und Petre [19] definieren die so genannten *Cognitive Dimensions*, die als Grundlage zur Untersuchung visueller Sprachen dienen können. Für den in dieser Arbeit intendierten Zweck ist das Modell allerdings problematisch, da dort die Evaluation der Sprache und die Evaluation der Sprachimplementierung vermischt werden. Die Sichtbarkeit von Abhängigkeiten bestimmter Sprachelemente ist z.B. ein reiner Sprachaspekt, wohingegen die Viskosität eines Programms auch von der Qualität der Sprachimplementierung abhängt.

Um die Viskosität verschiedener Sprachen zu vergleichen, benutzen Green und Petre Programme in verschiedenen visuellen und textuellen Sprachen, die die gleiche Aufgabe lösen. Sie bitten Testpersonen, bestimmte Änderungen an den Programmen durchzuführen und messen die dazu benötigte Zeit. Jung [28, S.205] hat diesen Test basierend auf VL-Eli-generierten Nassi-Shneiderman Editoren nachgestellt und gezeigt, dass die Editoren dank *di-*

rect manipulation und automatischem Layout eine relativ niedrige Viskosität aufweisen. Da DEViL auf den gleichen Interaktionsmechanismen basiert, ist dieses Ergebniss direkt auf DEViL übertragbar.

Grant [18] evaluiert das VPE-System (siehe Abschnitt 2.5.3), indem er Sprachen betrachtet, für die eine textuelle und eine visuelle Notation existieren. Er läßt kleine Programmfragmente in beiden Sprachvarianten konstruieren und misst jeweils die dazu benötigte Zeit. Auch dieser Test wurde von Jung [28, S.205f.] unter Verwendung des Nassi-Shneiderman Editors nachgestellt. Im Ergebnis benötigt die Erstellung eines Nassi-Shneiderman Diagramms lediglich ca. 30 Prozent mehr Zeit als die Erstellung eines vergleichbaren textuellen Programms. Auch dieser Wert ist sehr gut und belegt die Vorteile von *direct manipulation* und automatischem Layout.

6 Schlussbemerkungen

Inhalt

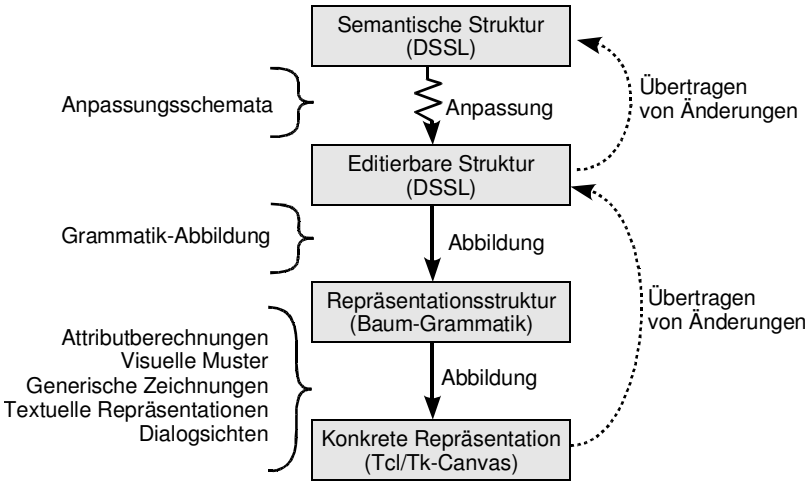


Abbildung 6.1: Gesamtkonzept des DEViL Systems

Im nachfolgenden Abschnitt fasse ich das in dieser Arbeit entwickelte Konzept zur Spezifikation visueller Strukturediteren übersichtsartig zusammen. Im Anschluss daran wird der Entwurf des Gesamtsystems abschließend diskutiert. Zuletzt werden einige Erweiterungsmöglichkeiten von DEViL vorgestellt, die mir als zukünftige Forschungsthemen besonders interessant erscheinen.

6.1 Das Konzept in Kürze

DEViL generiert Strukturediteren für visuelle Sprachen aus Spezifikationen hohen Niveaus. Die allgemeine Struktur eines mit DEViL generierten Editors ist in Abbildung 6.1 dargestellt. Die semantische Struktur repräsentiert den abstrakten Informationsgehalt eines visuellen Programms und abstrahiert vollständig von seiner Repräsentation. Die editierbare Struktur ist die Datenstruktur des visuellen Strukturediteors und beschreibt den Informationsgehalt der konkreten Repräsentation auf hohem Niveau. Die Repräsentationsstruktur bildet schließlich die Grundlage zur Berechnung und zum Layout der konkreten Darstellung.

Die Syntax der editierbaren und semantischen Struktur basiert auf der modellbasierten Spezifikationssprache DSSL, die mit sehr wenigen orthogonalen Modellierungskonzepten auskommt. Zur Definition der editierbaren Struktur werden im Allgemeinen nur die Abweichungen von der semantischen spezifiziert, so dass im Extremfall die beiden Strukturen überhaupt nicht unter-

schieden werden müssen. Zur Kopplung voneinander abweichender Strukturen wird ein asymmetrischer Mechanismus benutzt: Konsistenzbedingungen zwischen semantischer und editierbarer Struktur werden hergestellt, indem die editierbare Struktur an die semantische angeglichen wird. Zum Angleich der Strukturen existieren vordefinierte Kopplungsschemata, die je nach Bedarf für individuelle Sprachkonstrukte deaktiviert und ggf. durch alternative Implementierungen ersetzt werden können. In der Rück-Richtung werden die Strukturen gekoppelt, indem Änderungsoperationen weitergeleitet werden. Auch dieser Teil der Kopplung kann bedarfsgerecht individuell angepasst werden.

Die Syntax der Repräsentationsstruktur ist durch eine Baum-Grammatik spezifiziert. Im Allgemeinen wird jedes Sprachkonstrukt der editierbaren Struktur funktional auf ein Baumfragment der Repräsentationsstruktur abgebildet. Die Struktur wird dabei verfeinert. Auch diese Abbildung hat ein Standardverhalten, das bei Bedarf durch eine Zusatzspezifikation angepasst werden kann.

Aus der Repräsentationsstruktur wird durch Attributberechnungen eine konkrete Repräsentation berechnet. Die Attributberechnungen werden jedoch normalerweise nicht von Hand geschrieben, sondern es werden den Grammatiksymbolen vorgefertigte Berechnungsrollen zugeordnet, die visuelle Darstellungseigenschaften repräsentieren. Die Berechnungsrollen sind in einer Bibliothek so genannter visueller Muster gekapselt. Bei Bedarf können die Attributberechnungen der Berechnungsrollen individuell überschrieben werden, so dass die Implementierung eines visuellen Musters selbst in außergewöhnlichen Fällen weitgehend wiederverwendet werden kann. Visuelle Muster kapseln neben spezifischen Darstellungskonzepten auch maßgeschneiderte Layout- und Interaktionsmechanismen, so dass sehr benutzerfreundliche Editoren generiert werden können.

Zur Spezifikation bestimmter Teilrepräsentationen enthält DEViL Spezialsprachen. Generische Zeichnungen erlauben die visuelle Spezifikation von Sprachkonstrukten, die auf dem Formular-Muster basieren. Mit der Spezifikationssprache SLTR lässt sich sehr einfach die Repräsentation textuell darzustellender Teilstrukturen spezifizieren. Einfache Dialogsichten können durch ergänzende DSSL-Konstrukte spezifiziert werden. Für komplexere Dialogsichten lassen sich die gleichen Methoden einsetzen, die auch zur Spezifikation visueller Sichten verwendet werden.

6.2 Reflexion

Der wichtigste Beitrag dieser Arbeit ist der Entwurf eines flexiblen und zugleich benutzerfreundlichen Gesamtkonzepts zur Spezifikation visueller Strukturediteren. Die hierzu angestellten Überlegungen sollen nachfolgend aus verschiedenen Blickwinkeln betrachtet werden.

Das Spannungsfeld zwischen Einfachheit und Flexibilität Der Ausgangspunkt meiner Arbeiten war das VL-Eli System. Basierend auf dem Erfolg versprechenden Ansatz dieses Systems wollte ich ein flexibleres System realisieren, das genauso einfach anwendbar ist.

Das Spannungsfeld zwischen Einfachheit und Flexibilität galt es auf allen Entwurfsebenen zu berücksichtigen. Auf allen Ebenen wurden daher sowohl Verbesserungen entwickelt, die die Flexibilität betreffen, als auch solche, die die Benutzerfreundlichkeit betreffen.

- Auf der Ebene der strukturellen Repräsentation wurden die drei Abstraktionsebenen semantische Struktur, editierbare Struktur und Repräsentationsstruktur unterschieden, um die Flexibilität zu erhöhen. Damit die Benutzung des Generators nicht erschwert wird, ist die Unterscheidung der Strukturen optional.
- Auf der Ebene der visuellen Muster wurde das Konzept der globalen Layoutstrategien eingeführt, um ein stabiles Fundament zur Kombination von Muster-Varianten zu legen und sie flexibler und allgemeiner anwendbar zu machen. Die ebenfalls überarbeiteten Rollendiagramme vereinfachen die Anwendung der visuellen Muster.
- Auf der Ebene der konkreten Darstellung wurden Generische Zeichnungen eingeführt, um die Spezifikation von Verzierungen und Darstellungsdetails einfach und intuitiv zu machen. Den Generischen Zeichnungen wurden formale Parameter für Farbangaben und andere Attribute hinzugefügt, um sie noch flexibler anwendbar zu machen.
- Um auch textuelle Teilrepräsentationen spezifizieren zu können, wurden visuelle Muster dafür entwickelt, die flexibel mit anderen visuellen Mustern kombinierbar sind. Um die Spezifikation entsprechender Teilrepräsentationen zu vereinfachen, wurde die Spezialsprache SLTR eingeführt.

- Dialogsichten können mit einer in DSSL integrierten Spezialsprache sehr einfach spezifiziert werden. Um die Flexibilität zu erhöhen, wurden Spezial-Dialogsichten eingeführt, mit denen sich auch anspruchsvolle dynamische Dialoge realisieren lassen.
- In der Evaluation wurden sowohl die Flexibilität als auch die Benutzerfreundlichkeit des Ansatzes umfassend untersucht.

Um DEViL sowohl mächtig als auch benutzerfreundlich zu machen, wurden drei wichtige Entwurfsprinzipien angewendet, nämlich (1) die Verwendung optionaler Konzepte, (2) der Einsatz der 95-Prozent Methode und (3) die Kombination von Spezialmethoden. Hierauf möchte ich nachfolgend kurz eingehen.

Optionale Konzepte Um Anfängern die Benutzung eines Systems zu vereinfachen ist es eine gute Strategie, sie nicht mit einer großen Anzahl von Konzepten zu überfrachten. Aus diesem Grunde sind viele Konzepte in DEViL „optional“, d.h. sie brauchen erst dann zur Kenntnis genommen zu werden, wenn sie tatsächlich benötigt werden. Beispiele für optionale Konzepte sind die Unterscheidung zwischen editierbarer und semantischer Struktur, die Grammatik-Anpassung, die Spezifikationssprache für textuelle Repräsentationen sowie Spezial-Dialogsichten.

Um DEViL Anfängern näher zu bringen, können diese optionalen Konzepte zunächst weggelassen werden. Dadurch werden die Grundkonzepte des Systems unterstrichen und sind besser vermittelbar. Beschränkt man sich auf die Kernkonzepte von DEViL, so ist DEViL genauso bestechend einfach wie dessen Vorgänger VL-Eli: Man spezifiziert die abstrakte Struktur und ordnet den Strukturymbolen Rollen zu, die deren Repräsentation beschreiben.

Bei steigenden Anforderungen können dann schrittweise neue Konzepte hinzugenommen werden, um die Flexibilität zu erhöhen und auch anspruchsvollere Sprachen umsetzen zu können.

Der 95-Prozent Ansatz Um einfach anwendbare, intuitive Spezifikationsmethoden zu entwerfen, ist es manchmal vorteilhaft, selten vorkommende problematische Fälle bewusst auszuklammern. Häufig lohnt es sich, einen einfacheren Spezifikationsansatz zu wählen, mit dem sich lediglich 95 Prozent der Anwendungsfälle umsetzen lassen. Die restlichen fünf Prozent wer-

den bei diesem Ansatz mit entsprechend höherem Aufwand auf einer anderen Ebene gelöst. Insgesamt kann dadurch die Spezifikation im Vergleich zum „100-Prozent Ansatz“ einfacher und intuitiver gemacht werden, da beim „100-Prozent Ansatz“ der fünfprozentige Zuwachs an Mächtigkeit häufig auf Kosten der Einfachheit der restlichen 95 Prozent geht.

In DEViL wurde der 95-Prozent Ansatz sehr extensiv auf allen Spezifikationsebenen eingesetzt.

- Die Kopplung zwischen editierbarer und semantischer Struktur wird durch vordefinierte Anpassungsschemata beschrieben, durch deren Kombination ein Großteil der relevanten Fälle abgedeckt werden kann. In Ausnahmefällen kann die Spezifikation um handimplementierte Kopplungsoperationen ergänzt werden.
- Zur Abbildung auf die Repräsentationsstruktur wird ein Schema verwendet, das in vielen Fällen zum gewünschten Ergebnis führt. In Ausnahmefällen kann dieses Schema aber auch durch Zusatzspezifikationen angepasst werden.
- Die Spezifikation der visuellen Darstellung basiert im Normalfall auf visuellen Mustern, mit denen sich viele Sprachen vollständig beschreiben lassen. In Ausnahmefällen können Teile der Darstellung jedoch auch durch handimplementierte Attributberechnungen spezifiziert werden.
- Häufig brauchen die visuellen Muster-Varianten kaum parametrisiert zu werden, da viele Kontrollattribute mit sinnvollen Werten vorbelegt sind. In abweichenden Fällen können sie aber mit benutzerdefinierten Werten überschrieben werden.
- Im Normalfall werden Standard-Dialogsichten zur Spezifikation von Eingabedialogen verwendet. Bei Bedarf können aber auch die viel flexibleren Spezial-Dialogsichten verwendet werden.
- In Dialogen werden normalerweise vordefinierte Standard-Eingabekomponenten verwendet. In Ausnahmefällen können aber auch neue Varianten implementiert und der Spezifikation hinzugefügt werden.

Der Entwurf einer 95-Prozent Lösung ist oft eine besondere Herausforderung, denn es muss ein Kompromiss zwischen einfacher Spezifizierbarkeit

von Standardfällen und Minimierung der nicht spezifizierbaren Fälle gefunden werden. Um gute 95-Prozent Lösungen zu entwerfen, müssen relevante Einsatzszenarien durch empirische Untersuchungen praxisnaher Beispiele erarbeitet werden. Hierzu wurden häufig auftretende Abweichungen zwischen editierbarer und semantischer Struktur (Abschnitt 3.4), die Anforderungen an die Grammatik-Abbildung (Abschnitt 4.3), die Anforderungen an Standard- und Spezial-Dialogsichten (Abschnitt 4.6) sowie die Anforderungen an visuelle Muster (Abschnitt 4.2) untersucht. Diese Untersuchungen sind ein wichtiger empirischer Bestandteil dieser Arbeit.

Kombination von Spezialmethoden Ein weiteres wichtiges Entwurfsprinzip war die Kombination existierender Speziallösungen zu einem kohärenten Gesamtkonzept. Solche Speziallösungen, die teilweise in anderen Kontexten entwickelt wurden, enthalten häufig viel Expertenwissen und sind meist auch einfacher anwendbar als „alles abdeckende Universallösungen“. Einige der in dieser Arbeit kombinierten Speziallösungen sind Methoden zur Kopplung von Strukturen, objektorientierte Spezifikationsmethoden für Strukturen, attributierte Grammatiken, visuelle Muster, spezialisierte Layoutmethoden, das Konzept der Generischen Zeichnungen sowie Spezialspezifikationssprachen für textuelle Repräsentationen. Durch die Kombination all dieser Methoden konnte ein sehr leistungsfähiges Gesamtkonzept realisiert werden.

Praxisrelevante Kleinigkeiten Zusätzlich zu den oben genannten Entwurfskonzepten müssen zur Entwicklung eines praxistauglichen Generators viele Kleinigkeiten berücksichtigt werden. Dies ist mir vor allem während der Betreuung der Projektgruppe PaderWAVE bewusst geworden, da die entsprechenden Anforderungen erst beim Einsatz in realistischen Projekten zutage treten. Solche Kleinigkeiten sind z.B.

- der Umgang mit Bezeichnern und Namensräumen,
- automatische Strukturanpassungen innerhalb der semantischen Struktur sowie
- Mechanismen zur Definition vordefinierter Programmobjekte.

Der erste Punkt – der Umgang mit Bezeichnern und Namensräumen – ist wichtig, da auch in visuellen Sprachen Programmobjekte häufig einen Na-

men tragen. Bezeichner sind besonders für Dialog- und Baum-Sichten wichtig, da dort Querbeziehungen nicht auf andere Weise dargestellt werden können. Damit Programme gültig sind, müssen Bezeichner im Allgemeinen in einem bestimmten Kontext eindeutig sein. Um diesen Kontext festzulegen, lassen sich in DEViL u.a. Namensräume definieren.

Der zweite Spiegelpunkt drückt aus, dass es auch innerhalb der semantischen Struktur Konsistenzbedingungen geben kann, deren automatische Korrektur den Umgang mit einer Sprache erleichtert. Ein typisches Beispiel sind Funktionsaufrufe, bei denen die aktuellen und formalen Parameter konsistent zueinander sein müssen. In DEViL stehen Funktionen zur Verfügung, mit denen diese Programmstellen ähnlich wie bei der Kopplung zwischen semantischer und editierbarer Struktur automatisch gekoppelt werden können, so dass z.B. beim Erstellen eines Funktionsaufrufs sofort die richtige Anzahl aktueller Parameter erzeugt wird.

Der letzte Punkt spiegelt die Erkenntnis wider, dass es auch in visuellen Sprachen vordefinierte Programmobjekte gibt. Typische Beispiele sind Grundtypen wie *Integer* oder *String* in typisierten Sprachen. Zwar könnten die vordefinierten Programmobjekte auch syntaktisch modelliert werden, aber das würde die Sprachdefinition aufblähen und wesentlich schlechter wartbar machen. DEViL stellt deswegen Mechanismen zur Verfügung, um vordefinierte Programmobjekte einerseits wie „normale“ Programmobjekte behandeln zu können, deren Definition andererseits aber für den Sprachbenutzer transparent zu machen.

Diese und viele andere Kleinigkeiten konnten im Rahmen dieser Arbeit nicht umfassend behandelt werden, da die Arbeit ansonsten einen aufzählenden Charakter bekommen hätte. Trotzdem sind auch solche Details für die praktische Einsetzbarkeit eines Werkzeugsystems entscheidend.

Kompromisse Die praktische Entwicklung eines Systems wie DEViL ist unvermeidbar mit Zwängen und Randbedingungen verbunden, die bei rein theoretischen Arbeiten nicht existieren. Da es nicht praktikabel ist, alle Komponenten eines Systems von Grund auf neu zu entwickeln ist es unumgänglich, dass einige Komponenten nicht optimal auf den spezifischen Anwendungszweck abgestimmt sind. Häufig macht sich der Entwickler selbst nicht hinreichend bewusst, welche Eigenschaften eines Systems aus Entwurfsentscheidungen und welche aus äußeren Zwängen resultieren. Das liegt vermutlich an dem gleichen psychologischen Schutzmechanismus, der es einem

Individuum erst ermöglicht, sich in eine nicht immer ideale Umgebung zu integrieren. Trotzdem müssen die Kompromisse der Umsetzung soweit wie möglich offen gelegt werden, um zukünftigen Arbeiten die Verbesserung des Ansatzes zu erlauben. Die in dieser Hinsicht bedeutsame Kompromisse bei der Umsetzung von DEViL waren

- die Kodierung von Rollen in Symbolnamen, so dass der obere Kontext eines Symbols nicht als Rolle verfügbar ist, und
- die Beschränkung der Muster-Funktionalität auf Eigenschaften, die sich durch Attributberechnungen ausdrücken lassen.

Der erste Punkt besagt, dass die Rolle eines Symbols im Allgemeinen sowohl vom oberen als auch vom unteren Kontext bestimmt wird. Betrachten wir z.B. die Produktionen „`A ::= B`“ und „`B ::= C`“, dann ist es nützlich ausdrücken zu können, dass `B` im Kontext der ersten Produktion von `VPContainerElement` und im Kontext der zweiten Produktion von `VPTextPrimitive` erbt. In LIDO lässt sich hingegen nur ausdrücken, dass `B` immer beide Rollen erbt (genaueres siehe Abschnitt 4.1.2). Die Erfahrungen zeigen, dass die Konsequenzen dieser Einschränkung handhabbar sind. Allerdings würde die oben genannte Ideallösung einige Spezifikationsprobleme elegant vermeiden.

Der zweite Punkt stellt fest, dass die muster-spezifische Funktionalität in DEViL auf solche Eigenschaften beschränkt ist, die sich durch Attributberechnungen in Sichtspezifikationen ausdrücken lassen. Durch diese Technik lassen sich z.B. die Mechanismen zur Erzeugung neuer visueller Objekte nur eingeschränkt kapseln. Aus diesem Grund muss die Sprachkonstrukt-Leiste und deren Funktionalität separat spezifiziert werden. Beispielsweise die Festlegung, dass der Einfügeknopf für Linien einen besonderen Interaktionsmechanismus bereitstellen soll, kann nicht durch die Muster-Implementierung für Linien gekapselt werden. Auch musterspezifische strukturelle Anforderungen wie benötigte Layoutattribute oder Konsistenzbedingungen (z.B. bei Matrizen) können auf diese Weise nicht in den Muster-Implementierungen gekapselt werden. In DEViL ist der Sprachentwickler selbst dafür verantwortlich, die Spezifikation passend zu ergänzen. Um diesen Umstand zu ändern, müsste die Muster-Anwendung auf eine höhere Spezifikationsebene verlagert werden.

Zusammenfassung Abschließend lässt sich sagen, dass mit DEViL ein flexibles, praxistaugliches und zugleich recht anwenderfreundliches System realisiert werden konnte, das eine echte Alternative zur Handimplementierung visueller Strukturediteren darstellt. Der Ansatz ist auch für anspruchsvolle visuelle Sprachen wie UML geeignet, die eine Trennung zwischen editierbarer und semantischer Struktur erfordern.

Der wesentliche Beitrag der Arbeit und der Quell der Flexibilität von DEViL ist die Kombination vieler Einzelmethoden zu einem kohärenten Gesamtkonzept. Um die Spezifikation zu vereinfachen, wurden optionale Konzepte, 95-Prozent Ansätze und Spezialsprachen eingesetzt.

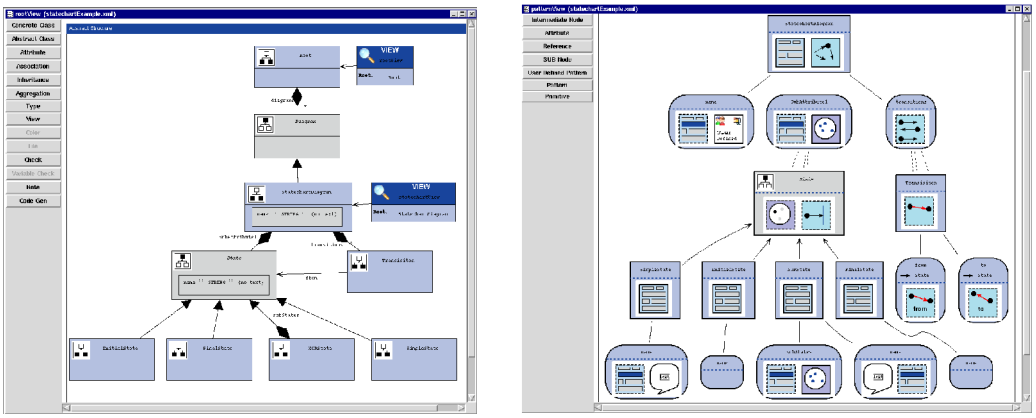
Durch die Evaluation konnte die Wirksamkeit und Benutzerfreundlickeit des Ansatzes sowohl auf Generator- als auch auf Produkt-Ebene gezeigt werden. Auf Generator-Ebene leisten vor allem die Generischen Zeichnungen und die wiederverwendbaren Muster-Implementierungen einen wichtigen Beitrag zur Benutzerfreundlichkeit, auf Produkt-Ebene sind es die gekapselten Interaktions- und Layoutmechanismen der Muster-Varianten.

6.3 Ausblick

Obwohl die Entwicklung von DEViL bereits recht weit fortgeschritten ist, sehe ich noch immer viel Verbesserungspotenzial. Schließlich ist das Gebiet der visuellen Sprachen ein sehr anspruchsvolles und wichtiges Forschungsthema. Die folgenden Punkte sind aus meiner Sicht als zukünftige Erweiterungen besonders interessant.

Visuelle Spezifikation visueller Sprachen Um vor allem Anfängern den Umgang mit DEViL zu erleichtern, scheint es zweckmäßig zu sein, eine visuelle Entwicklungsumgebung für DEViL zur Verfügung zu stellen, die den Sprachentwickler führt und die bereits während der Konstruktion bestimmte Konsistenzbedingungen der Spezifikation prüft.

Hierzu wurde bereits ein erster Prototyp erstellt. Eine mit dem so genannten *DEViL Designer* [11] erstellte visuelle Sprachspezifikation ist in Abbildung 6.2 dargestellt. Das System gestattet die visuelle Modellierung der abstrakten Syntax, der grafischen Darstellung und sogar einer einfachen Codegenerierung.



(a) Syntax-Spezifikation (b) Sicht-Spezifikation

Abbildung 6.2: Sprachspezifikation im *DEViL Designer*

Die abstrakte Syntax[1] wird durch eine Notation visualisiert, die weitgehend UML-Klassendiagrammen entspricht (siehe Abbildung 6.2a).

Die Spezifikation visueller Sichten basiert auf einer baumartigen Darstellung der Repräsentationsstruktur (siehe Abbildung 6.2b). Diese wird automatisch basierend auf der abstrakten Syntax erstellt und mit dieser konsistent gehalten. Der Repräsentationsstruktur lassen sich weitere Zwischenknoten hinzufügen, wodurch die Grammatik-Abbildung im Sinne von Abschnitt 4.3 angepasst wird. Den Symbolen der Repräsentationsstruktur werden visuelle Musterrollen zugeordnet, die in Abbildung 6.2b als Sinnbilder innerhalb der Baumknoten zu sehen sind. Die gleiche Art, Muster-Anwendungen zu visualisieren, wurde auch in dieser Arbeit, z.B. in Abbildung 4.18 auf Seite 170 verwendet.

Da die Umgebung die Struktur der verfügbaren Muster kennt, können Fehlanwendungen bereits während der Konstruktion gemeldet werden. Die Umgebung enthält des Weiteren verschiedene Assistenten, mit denen z.B. unvollständige Musteranwendungen ergänzt werden können. Eine initiale Evaluation mit Testpersonen hat gezeigt, dass dieser Ansatz die Benutzerfreundlichkeit des Generators nochmals wesentlich verbessert. Evtl. können demnächst auch unerfahrene Nutzer schon nach einer kurzen Einarbeitungszeit von wenigen Stunden einfache Strukturediteren spezifizieren.

Programm-Animation Im Kontext visueller Sprachen entsteht sehr schnell der Wunsch, visuelle Programme animieren zu können. Häufig wird durch

[1] editierbare und semantische Struktur werden noch nicht unterschieden

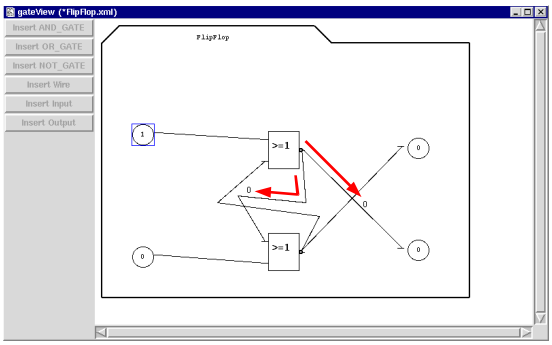


Abbildung 6.3: Bildschirmfoto einer Oberfläche zur Animation logischer Schaltungen

die Animation der Ablauf visueller Programme veranschaulicht. Besonders bei Systemen für Endbenutzer ist die Programm-Animation wichtig, um die Beziehung zwischen (statischem) Programm und (dynamischer) Ausführung zu verdeutlichen.

Auch in diesem Bereich wurden bereits erste Arbeiten durchgeführt. Abbildung 6.3 zeigt ein mit DEViL generiertes System zur Konstruktion und Animation von logischen Schaltungen [66]. Während der Animationsphase bewegen sich Einsen und Nullen entlang der Leitungen, um einen Wechsel des Spannungspegels zu visualisieren.

Zur Realisierung dieses Anwendungsbeispiels wurde ein Modul zur Simulation von logischen Schaltungen entwickelt und die Sichtspezifikation um Animationsfähigkeiten erweitert. Das Modul zur Simulation der Schaltung wurde aufgrund des anspruchsvollen Algorithmus von Hand implementiert und berechnet eine abstrakte Folge von Schaltungszuständen, wobei jeder Schaltungszustand durch die Belegungen aller Leitungen und Gatteranschlüsse charakterisiert ist. Die Sicht-Spezifikation berechnet nicht nur die (statische) visuelle Darstellung, sondern setzt im Animationsmodus auch den (abstrakten) Schaltungszustand in eine Animation um, indem der grafischen Darstellung Animationsobjekte und Pfade hinzugefügt werden. In weiterführenden Arbeiten sollten andere Animationsarten untersucht und gekapselt werden, wobei auch hier ein mit visuellen Mustern vergleichbares Spezifikationskonzept erfolgversprechend scheint.

Parsieren von Strukturen niedrigeren Niveaus Häufig wünschen sich DEViL-Anwender, dass textuelle Bestandteile visueller Repräsentationen frei eingegeben und parsiert werden können. Dieser Wunsch ist normalerweise

auf textuelle Teilrepräsentationen beschränkt, obwohl dies, wie z.B. DiaGen II zeigt, auch für visuelle Repräsentationen möglich ist. Da Parsing-Techniken nicht nur für freie Editoren, sondern bereits für Strukturediteren mit einer Editor-Syntax niedrigeren Niveaus benötigt werden, halte ich eine Erweiterung von DEViL in diese Richtung durchaus für sinnvoll. Da die hierfür benötigten Methoden bereits sehr weit entwickelt sind, besteht die eigentliche Herausforderung darin, sie in das Gesamtkonzept von DEViL zu integrieren.

Spezifikation komplexer Kopplungen Wie bereits in Abschnitt 3.5 diskutiert, kann die manuelle Implementierung komplexer struktureller Kopplungen in DEViL sehr aufwändig und fehleranfällig sein. Diese Erfahrung wurde vor allem im Zusammenhang mit der Entwicklung des *DEViL Designers* gemacht, bei dem die Spezifikationen der abstrakten Syntax und der visuellen Darstellungen auf sehr anspruchsvolle Weise miteinander gekoppelt sind. Wie bereits in Abschnitt 3.5 angesprochen, halte ich es für sinnvoll, solche Kopplungen durch Spezialsprachen zu vereinfachen, die auf Graphtransformation basieren. Beim Entwuf eines entsprechenden Spezifikationskonzepts könnten der *DEViL Designer* und die in Abschnitt 3.4 dargestellten Anwendungsbeispiele als Anforderungsdefinition dienen.

Entwurfsmethodik für visuelle Sprachen In Studien- und Diplomarbeiten stellt sich immer wieder die Frage, wie man gute visuelle Sprachen entwirft. Zwar gibt es hierfür einige Leitlinien [69, 19, 36], aber ich bin der Meinung, dass das Konzept der visuellen Muster auch beim Sprachentwurf einen wichtigen Beitrag leisten könnte. Da visuelle Sprachen in DEViL in der überwiegenden Zahl der Fälle fast ausschließlich durch Anwendung visueller Muster spezifiziert werden, ist es sinnvoll, die Auswirkungen der Muster-Wahl auf die Benutzbarkeit visueller Sprachen zu untersuchen. Diese Idee wurde bereits in [54] formuliert und rudimentär mit Leben gefüllt. Zumindest könnten die Vor- und Nachteile bestimmter visueller Muster sowie besonders wirkungsvolle Muster-Kombinationen genauer untersucht werden. Evtl. lassen sich visuelle Muster aber auch mit anderen Entwurfsmethodiken in Beziehung setzen. Die gewonnenen Erkenntnisse könnten auch in den *DEViL Designer* einfließen, um unerfahrene Sprachentwickler nicht nur bei der Umsetzung, sondern auch beim Entwurf einer visuellen Sprache zu unterstützen.

Anhang

Literaturverzeichnis

[1] *Eli Online Documentation: Pattern-based Text Generator*, 2005. `http://www.upb.de/cs/ag-kastens/elionline/ptg_toc.html`.

[2] *Eli Online Documentation: Syntactic Analysis*, 2005. `http://ag-kastens.uni-paderborn.de/elionline/syntax_toc.html`.

[3] D. H. Akehurst. An oo visual language definition approach supporting multiple views. In *VL2000, IEEE Symposium on Visual Languages*, September 2000.

[4] B. Backlund, O. Hagsand, and B. Pherson. Generation of visual language-oriented design environments. *J. of Visual Lang. and Comp.*, 1(4):333–354, 1990.

[5] R. Bahlke and G. Snelting. Design and structure of a semantics-based programming environment. *International Journal of Man-Machine Studies*, 37(4):467–479, 1992.

[6] Rolfe Bahlke and Gregor Snelting. The PSG system: from formal language definition to interactive programming environments. *ACM Trans. Prog. Lang. and Sys.*, 8(4):547–576, October 1986.

[7] Roswitha Bardohl. GenGed: A generic graphical editor for visual languages based on algebraic graph grammars. In *1998 IEEE Symp. on Visual Lang.*, pages 48–55, September 1998.

[8] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley, aug 2003.

[9] Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. Scaling up visual programming languages. *IEEE Computer*, 28(3):45–54, 1995.

[10] MetaCase Consulting. *MetaEdit+ User's Guide*, 2002. http://www.metacase.com/fs.asp?vasen=vasen.html&paa=products.html.

[11] Bastian Cramer. Generierung von graphischen Strukturediteren aus visuellen Spezifikationen. Diplomarbeit, Universität Paderborn, 2005. http://ag-kastens.uni-paderborn.de/paper/cramer2005.pdf.

[12] Sergey Dmitriev. Language oriented programming: The next programming paradigm. JetBrains 'onBoard' electronic monthly magazine, 2004. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/.

[13] Jürgen Ebert, Roger Süttenbach, and Ingar Uhe. Meta-CASE in practice: a case for KOGGE. In *CAiSE*, pages 203–216, 1997.

[14] Martin Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. http://martinfowler.com/articles/languageWorkbench.html.

[15] Paul Franchi-Zannettacci. Attribute specifications for graphical interface generation. In G. X. Ritter, editor, *Inform. Proc. '89*, pages 149–155. North-Holland, 1989.

[16] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[17] E. P. Glinert. Towards "second generation" interactive, graphical programming environments. In *Proc. of IEEE 2nd Fall Comp. Conf.*, pages 292–299, 1987.

[18] Calum A. M. Grant. Visual language editing using a grammar-based visual structure editor. *J. of Visual Lang. and Comp.*, 9:351–374, 1998.

[19] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. of Visual Lang. and Comp.*, 7(2):131–174, 1996.

[20] Peer Griebel et al. Integrating a constraint solver into a real-time animation environment. In *Proc. of the 1996 IEEE Symp. on Visual Lang.*, pages 12–19. IEEE Comp. Soc. Press, 1996.

[21] David Harel. On visual formalisms. *Comm. of the ACM*, 31(5):514–530, May 1988.

[22] Magnus Lie Hetland. *Practical Python*. Apress, Berkeley, CA, USA, 2002.

[23] Honeywell, Inc. Dome guide, 1999. `http://www.htc.honeywell.com/dome/DOMEGuide.pdf`.

[24] Paul Hudak, John Peterson, and Joseph Fasel. *A Gentle Introduction To Haskell*, 1998. `http://haskell.cs.yale.edu/tutorial/`.

[25] Bertrand Ibrahim and Hidenori Yoshizumi. Solving the spaghetti plate syndrome in a control-flow language with a VLSI-like solution. In *1999 IEEE Symp. on Visual Lang.*, pages 202–203, 1999.

[26] Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, and Stefan Queins. *UML 2 glasklar*. Hanser, November 2003.

[27] Patrick W. Jordan. *An Introduction to Usability*. Taylor & Francis, 1998. (paper) 0-7484-0794-4 (cloth).

[28] Matthias Jung. *Ein Generator zur Entwicklung visueller Sprachen*. Dissertation, Universität Paderborn, November 2000.

[29] Matthias Jung, Uwe Kastens, Christian Schindler, and Carsten Schmidt. A pattern-based generator for implementation of visual languages. In *Proceedings of IEEE 2000 International Symposium on Visual Languages*, pages 71–72, Seattle, Washington, September 2000. IEEE Computer Society Press.

[30] Uwe Kastens and Matthias Jung. Streets Abschlußbericht. Technical report, Universität Paderborn, 1998. `http://www.upb.de/cs/ag-kastens/paper/streets.ps.gz`.

[31] Uwe Kastens, Peter Pfahler, and Matthias Jung. The Eli system. In Kai Koskimies, editor, *Proceedings of 7th International Conference on Compiler Construction CC'98*, number 1383 in Lecture Notes in Computer Science, pages 294–297. Springer Verlag, March 1998.

[32] Uwe Kastens and Carsten Schmidt. VL-Eli: A generator for visual languages. In *Proceedings of Second Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, number 2027 in Electronic Notes in Theoretical Computer Science, Grenoble, France, 2002. Band 65, Elsevier Science Publishers.

[33] Uwe Kastens and William M. Waite. Modularity and reusability in attribute grammars. Technischer Bericht, Reihe Informatik tr-ri-92-102, Universität Paderborn Fachbereich Mathematik-Informatik, July 1992.

[34] D. E. Knuth. *Tex and METAFONT, New Directions in Typesetting*. Digital Press, Billerica, MA, 1979.

[35] K. Marriott, S. S. Chok, and A. Finlay. A tableau based constraint solving toolkit for interactive graphical applications. Technical Report 98/07, School of Computer Science & Software Engineering, Monash University, Australia 3168, June 1998.

[36] R. Mark Meyer and Tim Masterson. Towards a better visual programming language: critiquing prograph's control structures. *The Journal of Computing in Small Colleges*, 15(5):181–193, 2000.

[37] Mark Minas. *Spezifikation und Generierung graphischer Diagrammeditoren*. Shaker-Verlag, 2001.

[38] Brad Myers. Visual programming, programming by examples, and program visualization: A taxonomy. In *Proc. ACM Conf. Human Factors in Computer Systems, CHI*, pages 59–66, 1986.

[39] Bonnie A. Nardi and Craig L. Zarmer. Beyond models and metaphors: Visual formalisms in user interface design. *J. Vis. Lang. Comput*, 4(1):5–33, 1993.

[40] I. Nassi and B. Shneidermann. Flowchart techniques for structured programming. In *Proc. SIGPLAN'73*, pages 12–26, 1973.

[41] Object Management Group. *Unified Modeling Language (UML), version 2.0*, 2005. `http://www.omg.org/technology/documents/formal/uml.htm`.

[42] Dereck C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.

[43] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[44] Angela Pacifico. Generierung eines Strukturediteors für Sequenzdiagramme nach UML 2.0. Studienarbeit, Universität Paderborn, 2005. `http://ag-kastens.uni-paderborn.de/paper/studienarbeiten/pacifico2005.pdf`.

[45] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Münster, 1962.

[46] Jörg Poswig, Guido Vrankar, and Claudio Moraga. VisaVis: a higher-order functional visual programming language. *J. Vis. Lang. Comput*, 5(1):83–111, 1994.

[47] Waldemar Reisch. Implementierung von wiederverwendbaren Grid-Layout-Strategien am Beispiel eines Editors für elektronische Schaltpläne. Studienarbeit, Universität Paderborn, 2005. `http://ag-kastens.uni-paderborn.de/paper/studienarbeiten/reisch2005.pdf`.

[48] J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In *1996 IEEE Symp. on Visual Lang.*, pages 148–155. IEEE Comp. Soc. Press, 1996.

[49] A. Repenning and T. Sumner. Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer*, 28(3):17–26, 1995.

[50] Stefan Schiffer. *Visuelle Programmierung - Grundlagen und Einsatzmöglichkeiten*. Addison Wesley, 1998.

[51] Carsten Schmidt. Visuelle Muster in DEViL, 2005. `http://ag-kastens.uni-paderborn.de/forschung/devil/documentation/visualPatterns-html.gen/main.html`.

[52] Carsten Schmidt and Uwe Kastens. Implementation of visual languages using pattern-based specifications. *Software - Practice and Experience*, 33:1471–1505, December 2003.

[53] Carsten Schmidt, Peter Pfahler, Uwe Kastens, and Carsten Fischer. SIMtelligence Designer/J: A visual language to specify SIM toolkit applications. In *Proceedings of Second Workshop on Domain Specific Visual Languages (OOPSLA 2002)*, Seattle, WA, USA 2002, 2002.

[54] Carsten Schmidt and Christian Schindler. Muster-basierte Generierung von Struktur-Editoren für visuelle Sprachen. Diplomarbeit, Universität Paderborn, Germany, January 2000.

[55] Carsten Schmidt and Michael Thies. PaderWAVE Abschlußbericht. Technical report, Universität Paderborn, 2005. `http:`

//ag-kastens.uni-paderborn.de/lehre/paderwave/material/PaderWAVE-Abschlussbericht.pdf.

[56] Martijn M. Schrage. *Proxima – a presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands, October 2004.

[57] Elvira Schumacher. Systematische Spezifikation eines Editors für Zustandsdiagramme nach dem UML 2.0-Standard. Studienarbeit, Universität Paderborn, 2005. http://ag-kastens.uni-paderborn.de/paper/studienarbeiten/schumacher2005.pdf.

[58] Eike Schwindt. Ein graphischer Editor für Generische Zeichnungen. In *Fachwissenschaftlicher Informatikkongress - Informatiktage 2002*, Bad Schussenried, November 2003.

[59] Andy Schürr. Developing graphical (software engineering) tools with PROGRES. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 618–619. ACM Press, 1997.

[60] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

[61] D. C. Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhauser, Basel, 1977.

[62] Hung-Khoon Tan and Wentong Cai. VPEcons: A visual constructor for parallel programming. In *1st IEEE International Conference on Algorithms and Architecture for Parallel Processing*, volume 2, pages 565–574. IEEE Computer Society, April 1995.

[63] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[64] G. M. Vose and G. Williams. Labview: Laboratory virtual instrument engineering workbench. *BYTE*, pages 84–92, September 1986.

[65] W3C. *Cascading Style Sheets, level 2*, 1998. http://www.w3.org/TR/REC-CSS2/.

[66] Manuel Wickert. Einsatz des Generators DEViL zur Animation von Logikbausteinen. Studienarbeit, Universität Paderborn,

2005. http://ag-kastens.uni-paderborn.de/paper/studienarbeiten/wickert2005.pdf.

[67] Stephan Winter. Generierung von dynamischen Web-Anwendungen aus visuellen Spezifikationen. In *Fachwissenschaftlicher Informatikkongress - Informatiktage 2005*, Schloss Birlinghoven - Sankt Augustin, April 2005.

[68] XML Path Language (XPath) Version 1.0, W3C Recommendation, November 1999. http://www.w3c.org/TR/xpath.

[69] Sherry Yang, Margaret M. Burnett, Elyon DeKoven, and Moshé M. Zloof. Representation design benchmarks: A design-time aid for VPL navigable static representations. *J. Vis. Lang. Comput*, 8(5-6):563–599, 1997.

Abbildungsverzeichnis

Tabellenverzeichnis