Generierung von graphischen Strukturediteren aus visuellen Spezifikationen

Diplomarbeit vorgelegt bei
Herrn Prof. Dr. Uwe Kastens

Bastian Cramer
Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

Paderborn, November 2005

Danksagung

Zunächst gilt mein Dank Prof. Dr. Uwe Kastens, der diese Arbeit ermöglichte und konstruktive Ratschläge beisteuern konnte.

Insbesondere möchte ich mich bei Carsten Schmidt bedanken, der mir während der gesamten Arbeit mit wertvollen Hinweisen und technischer Hilfe zur Seite stand. Ganz besonders bedanke ich mich für das geduldige Lesen und die Kommentare, die die Qualität dieser Arbeit enorm gesteigert haben.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte fremde Hilfe sowie ohne Benutzung anderer als den angegebenen Quellen angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

_____

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

Mit graphischen Strukturediteren können Softwareprodukte auf einem hohen Abstraktionsniveau entwickelt und in kurzer Zeit Prototypen erstellt werden. Beliebt sind CASE Tools zur Softwareentwicklung, wie UML Editoren, die es erlauben, Software mittels standardisierter Konstrukte zu entwerfen. Auch in anderen Anwendungsgebieten werden visuelle Sprachen zur Entwicklung eingesetzt, so können mit Kontaktplan speicherprogrammierbare Steuerungen entworfen werden. Der Einsatz von visuellen Konstrukten, die in textuellen Programmiersprachen nicht existieren, erlaubt es, eine visuelle Sprache speziell für eine Domäne anzupassen. Komplexe Strukturen können durch den Gebrauch von mehrdimensionalen graphischen Darstellungen besser abgebildet werden. Intuitive Konzepte und bekannte Strukturen, wie Mengen oder Listen, sollen auch Anfänger bei der Entwicklung unterstützen. Allerdings sind visuelle Sprachen nicht selbsterklärend, und die Gesamtheit eines komplexen Programms kann mit einer einzigen Sicht ebenfalls nicht erfasst werden. Die mehrdimensionale Darstellung kommt dem Menschen jedoch näher, da dieser darauf eingestellt ist, sich in einer vierdimensionalen Welt zu bewegen.

Graphische Strukturediteren können von Generatoren erzeugt werden. Das in dieser Arbeit benutzte Werkzeug DEViL ist ein solcher Generator. Der Benutzer spezifiziert das abstrakte Strukturmodell eines Editors und wendet auf dieses Modell visuelle Muster an, die die graphische Repräsentation festlegen. Die Anwendung dieser visuellen Muster erfolgt dabei im Wesentlichen durch das Spezifizieren von allgemeinen Verhaltensweisen. So wird zum Beispiel ausgedrückt, dass ein bestimmtes Strukturobjekt wie eine Menge dargestellt werden soll. Zusätzliche Attribute und eine große Bibliothek visueller Muster erlauben es dem Benutzer, eine visuelle Sprache ganz nach seinem Geschmack zu entwerfen. Die entworfenen Spezifikationsdateien werden dann DEViL übergeben und der Generator erzeugt daraus den gewünschten Strukturediteor. Dieser Spezifikationsprozess hat den Vorteil, dass er auf einem hohen Abstraktionsniveau stattfindet und der Benutzer keine grundlegenden Kenntnisse in der Programmierung von graphischen Benutzeroberflächen mehr benötigt. Es lassen sich intuitiv bedienbare Oberflächen erstellen. Der Generator stellt diese Funktionalität zur Verfügung und macht sie transparent für den Benutzer.

Der generierende Ansatz für visuelle Sprachen ist nicht neu, jedoch sind Generatoren für Anfänger häufig schwierig zu benutzen. Auch die Spezifikation von graphischen Repräsentationen ist oft nur durch komplexe mathematische Ausdrücke möglich. Mit dem DEViL Generator steht ein mächtiges Werkzeug zur Verfügung, das über eine reichhaltige Bibliothek an visuellen Mustern verfügt und so eine große Vielfalt visueller Sprachen implementieren kann. Allerdings sind die Spezifikationssprachen von DEViL nicht leicht verständlich. Konzepte des Generators sind gerade für Anfänger nur schwer zu erkennen.

In dieser Arbeit soll für diese textuellen Spezifikationsdateien eine visuelle Umgebung, im Folgenden *DEViL-Designer* genannt, entwickelt werden. Besonders Anfänger

im Umgang mit visuellen Sprachen sollen so befähigt werden, Konzepte und Techniken zur Generierung von Strukturediteren mit dem DEViL Generator zu erlernen.

In der vorliegenden Arbeit werden zunächst die Grundlagen des DEViL Systems erläutert. Dabei wird besonders auf das DEViL-Spezifikationskonzept eingegangen. Danach werden existierende Systeme daraufhin untersucht, inwiefern visuelle Ausdrucksmittel in diese Arbeit einfliessen können. In Kapitel 4 werden dann Anforderungen an die zu entwerfende Umgebung gestellt, bevor konkret auf die visuellen Teilsprachen eingegangen wird, die im Rahmen dieser Arbeit entworfen wurden. Diese Teilsprachen erlauben eine vollständig visuelle Modellierung von graphischen Strukturediteren auf Basis der Spezifikationskonzepte von DEViL. Auf Details der Implementierung wird im darauffolgenden Kapitel eingegangen. Zum Schluss wird die hier entworfene Umgebung anhand eines formalen Modells und in einem kontrollierten Experiment evaluiert.

2 Aufgabenbeschreibung

Zur Spezifikation graphischer Strukturediteren muss der Benutzer eine Reihe von Spezifikationssprachen anwenden, die dem Generator DEViL als Eingabe dienen. Diese Spezifikationen sind anfangs häufig schwer zu verstehen und es ist Expertenwissen aus diversen Bereichen nötig: Der Entwickler muss mit dem Umgang von Grammatiken vertraut sein, er muss etwas von Sprachspezifikation verstehen und nicht zuletzt muss er Kenntnisse in diversen Programmiersprachen besitzen. Insgesamt ist die Spezifikation von graphischen Strukturediteren ein komplexer Vorgang, der viel Erfahrung voraussetzt und Anfänger abschreckt. Die Entwicklung einer visuellen Repräsentation soll den Spezifikationsprozess erleichtern.

In der vorliegenden Arbeit sollen dazu visuelle Teilsprachen für den Generator DEViL entwickelt werden, die diesen komplexen Vorgang mit all den textuellen Spezifikationsdateien ersetzen sollen. Dazu wird mit Hilfe des DEViL Generators ein visueller Editor, der DEViL-Designer, entwickelt, der es erlaubt, beliebige graphische Strukturediteren zu spezifizieren. Diese Spezifikationen werden dann auf textuelle DEViL-Spezifikationen abgebildet und dienen dem Generator dann wiederum als Eingabe.

Dem Benutzer soll es ermöglicht werden, graphische Strukturediteren visuell zu spezifizieren. Ebenso soll eine visuelle Spezifikation der Codegenerierung möglich sein. Zudem soll der DEViL-Designer dem Benutzer dabei helfen, schnell konsistente Spezifikationen für Strukturediteren zu entwerfen. Die visuelle Sprache soll intuitiv bedienbar sein und den Benutzer in bestimmten Situationen helfend unterstützen. Die Umgebung soll insbesondere von Anfängern im Umgang mit visuellen Sprachen benutzt werden, so dass Konzepte von Generatoren für graphische Strukturediteren vermittelt werden und auch eine spätere textuelle Spezifikation von Strukturediteren möglich wird. Dies soll am Schluss dieser Arbeit durch eine Evaluationsphase überprüft werden. Zudem soll der DEViL-Designer flexibel genug gestaltet werden, dass spätere Änderungen am DEViL Generator, wie neue visuelle Muster, schnell in die Umgebung mit einfließen können. Der Benutzer soll außerdem jederzeit in der Lage sein, seine bisherigen Teilergebnisse zu testen.

Es werden in dieser Arbeit also eine Reihe von Teilsprachen implementiert, die die Spezifikation von graphischen Strukturediteren und deren Codegenerierung auf visuellem Weg realisiert. Zudem werden Teilsprachen entworfen, die den DEViL-Designer sehr flexibel in Hinsicht auf zu erwartende Erweiterungen machen. Diese Erweiterungen umfassen z.B. die Spezifikation von neuen visuellen Mustern. Zudem werden zusätzliche Sprachkonstrukte definiert, die von speziellen Klassen von visuellen Sprachen benötigt werden. Es wird jedoch nicht möglich sein, den gesamten Funktionsumfang von DEViL zu unterstützen. Deshalb werden in dieser Arbeit hauptsächlich Sprachen betrachtet, die einen nicht allzu großen Umfang haben. Trotzdem wird der DEViL-Designer in der Lage sein, unterschiedliche Strukturediteren zu generieren, sei es mit einem hohen graphischen oder mit einem hohen textuellen Anteil.

3 Grundlagen

In diesem Kapitel soll Grundlegendes zu visuellen Sprachen und deren Vorteile erläutert werden. Außerdem soll das Prinzip der Generierung von graphischen Strukturediteren und der Generator DEViL im Speziellen erklärt werden. Damit wird die Grundlage für die Arbeit gelegt.

3.1 Visuelle Sprachen

Visuelle Sprachen spielen eine bedeutende Rolle bei der Modellierung von Softwaresystemen und beim Einsatz in speziellen Anwendungen (DSL = Domain Specific Language). Durch die Wahl von bestimmten graphischen Repräsentationen und visuellen Mustern können komplexe Strukturen, wie beispielsweise Algorithmen oder größere Fallunterscheidungen wie in Nassi-Shneiderman Diagrammen so dargestellt werden, dass sie von Menschen leicht erkannt und verstanden werden. Zusätzlich können domänenspezifische Metaphern verwendet werden, die es Experten erlauben, ihre eigenen Abstraktionsniveaus und Beschreibungen zu nutzen. Visuelle Sprachen sind nicht selbsterklärend, haben jedoch gegenüber textuellen Sprachen einige Vorteile. Das hohe Abstraktionsniveau der Modelle erlaubt eine einfache Wartbarkeit und eine hohe Entwicklungsgeschwindigkeit, zudem wird die Softwarequalität gesteigert. Ein Technologiewechsel ist ebenso einfach möglich, da die innere Struktur, die jedem Softwareprodukt zugrunde liegt, nicht durch Details der Programmiersprachen versteckt wird.

Visuelle Programmiersprachen erlauben es, durch prägnante Notationen leicht verständliche Modellbeschreibungen zu erstellen. Sie vermeiden, soweit möglich, textuelle Bestandteile. Eine visuelle Repräsentation ist nicht mehr nur die reine Illustration des Programms, sondern sie stellt das ausführbare Programm selbst dar. Visuelle Sprachen werden in unterschiedlichen Gebieten eingesetzt und können sehr unterschiedliche visuelle Stilmittel benutzen: Editoren für UML [9] Klassen- oder Zustandsdiagramme werden zur Modellierung und Implementierung objektorientierter Softwaresysteme eingesetzt. Die visuelle Darstellung ist hier auf Rechtecke, Linien und Pfeile beschränkt (siehe Abbildung 1). Nassi-Shneiderman Diagramme, die der Darstellung imperativer Programmstrukturen dienen, stellen Schleifen, Anweisungen und bedingte Ausdrücke durch die Aneinanderreihung und Unterteilung von Rechtecken dar. Die Spezifikationssprache Streets [19] zur imperativen Programmierung paralleler Berechnungen unter Verwendung des Message Passing Prinzips bewegt sich auf einem sehr hohen Abstraktionsniveau. Die Metapher “Straße“ dient der Visualisierung des Programmablaufs. Schleifen und Bedingungen des Programms werden durch Kreuzungen und Schleifen einer Straße visualisiert (siehe Abbildung 2). Die Aufrufe von Programmobjekten, Komponenten genannt, werden visuell auf der Straße platziert. Auch im Bereich der Mikrocontrollerprogrammierung werden visuelle Sprachen eingesetzt. Kontaktplan (KOP) beispielsweise dient der Programmierung speicherprogrammierbarer Steuerungen. Dabei kann der Stromfluss anhand zweier paralleler Schienen modifiziert und visualisiert werden.

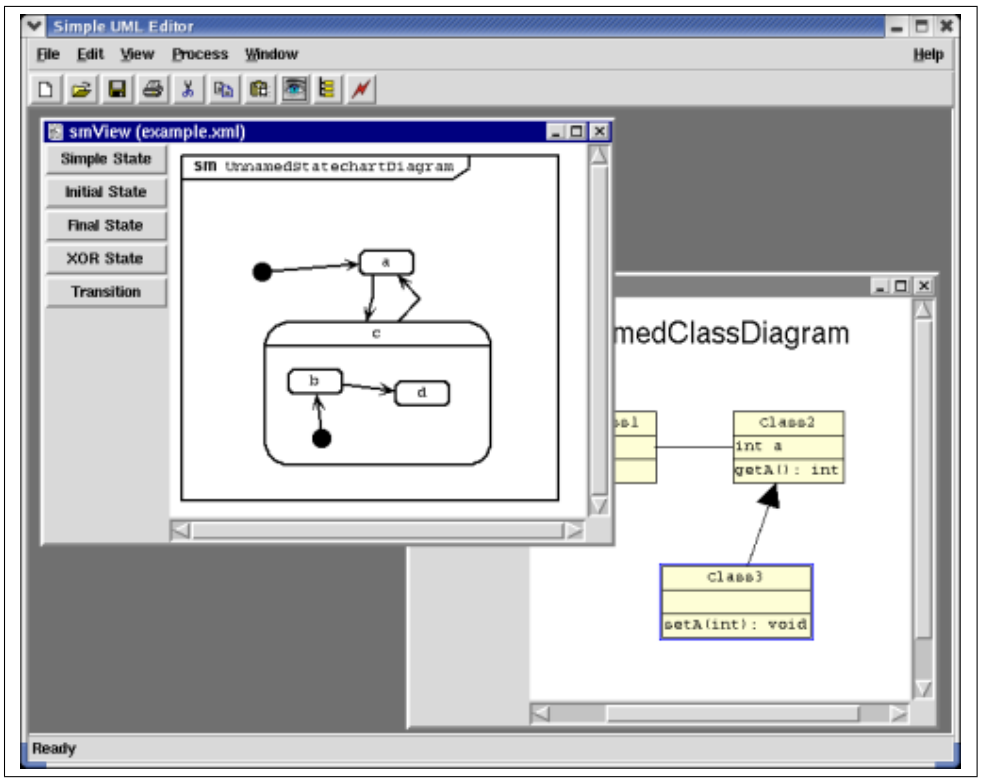

Abbildung 1: Strukturediteor für UML Diagramme

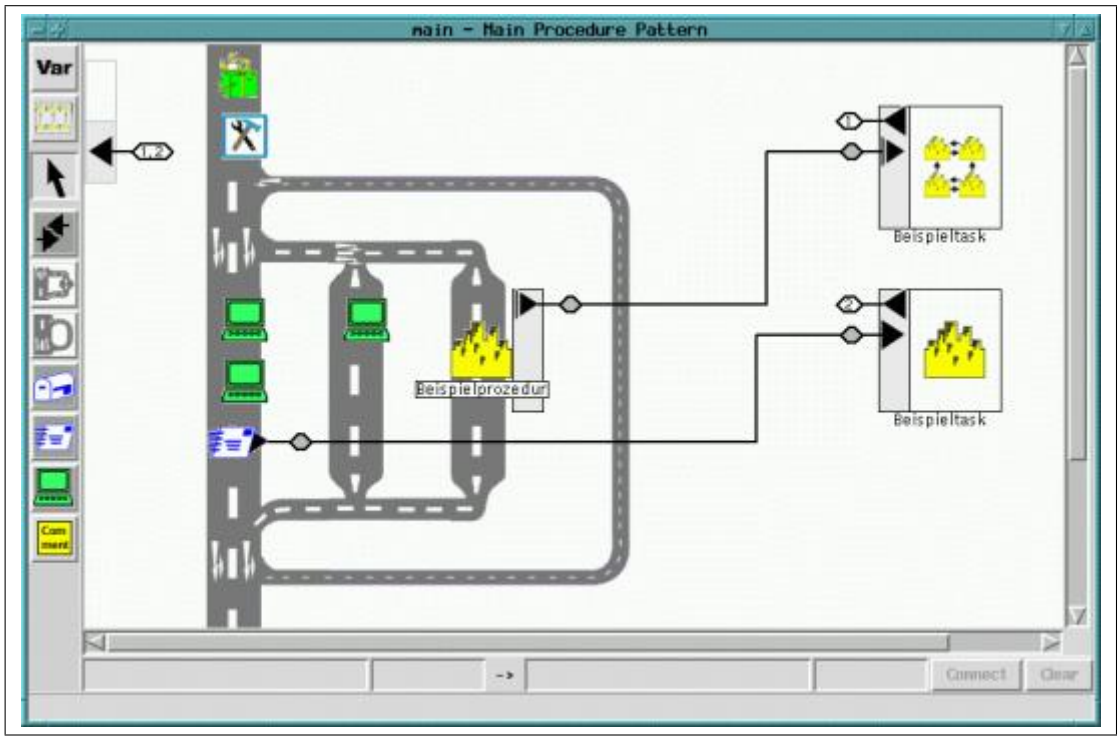

Abbildung 2: Strukturediteor für Streets

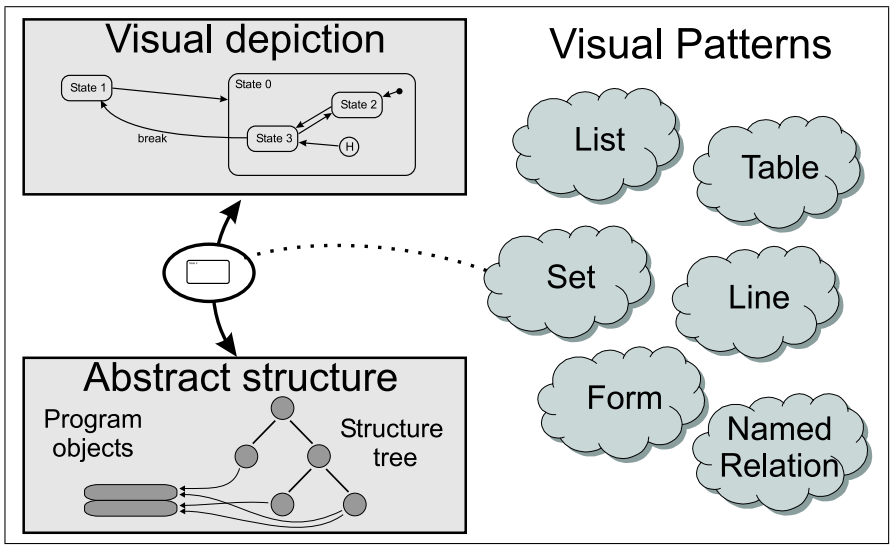

Abbildung 3: Patternbasierte Spezifikation (aus [12])

3.2 DEViL

DEViL (Development Environment for Visual Languages) [12] wird in der Fachgruppe "Programmiersprachen und Übersetzer" an der Universität Paderborn entwickelt und erzeugt visuelle Entwicklungsumgebungen aus Spezifikationen der Sprachstruktur und der graphischen Darstellung einer visuellen Sprache, wie in Abbildung 3 zu sehen. Strukturediteren werden dabei zunächst durch ein formales Modell beschrieben. Dieses Modell beschreibt den Editor konzeptionell. Eine Instanz des Modells ist die abstrakte Struktur und beschreibt einen spezifischen Strukturediteur. An Objekte der abstrakten Struktur können visuelle Muster angewendet werden. Die Muster geben an, wie Teile der abstrakten Struktur strukturiert und visualisiert werden können. Auf die Spezifikation eines graphischen Strukturediteurs soll nun eingegangen werden.

Spezifikation des Sprachmodells Der Entwurf einer visuellen Sprache beginnt mit der Spezifikation der abstrakten Struktur, die DEViL als Ausgangspunkt dient. Die Spezifikation des Modells basiert auf dem Konzept von Klassenhierarchien. Klassen des Sprachmodells enthalten Attributdefinitionen und können Eigenschaften von allgemeineren (abstrakten) Klassen erben. Zur Laufzeit können Instanzen von Klassen erzeugt werden. Es soll nun konkret auf die Eigenschaften der Spezifikationssprache des Modells eingegangen werden. Dazu ein Beispiel in Abbildung 4, das auch im weiteren Verlauf dieser Arbeit immer wieder aufgegriffen werden soll. Das Beispiel aus Abbildung 4 zeigt die abstrakte Struktur eines UML [9] Strukturediteurs für Statechart Diagramme.

Jede abstrakte Struktur besitzt eine Wurzelklasse *Root*, die als Ausgangsklasse angesehen wird. In dem Beispiel enthält die Wurzelklasse eine Folge von Unterklassen vom Typ *Diagram*.

```
CLASS Root {
  diagrams: SUB Diagram*;
}

ABSTRACT CLASS Diagram {
  name: VAL VLString;
}

CLASS StatechartDiagram INHERITS Diagram {
  setSize: VAL VLPoint? INIT "400 300"  EDITWITH "None";
  states: SUB State*;
  transitions: SUB Transition*;
}

ABSTRACT CLASS State {
  position: VAL VLPoint?  EDITWITH "None";
}

CLASS InitialState INHERITS State {}

CLASS FinalState INHERITS State {}

CLASS SimpleState INHERITS State {
  name: VAL VLString;
}

CLASS XORState INHERITS State {
  name: VAL VLString;
  subStates: SUB State*;
  setSize: VAL VLPoint? INIT "140 80"  EDITWITH "None";
}

CLASS Transition {
  position: VAL VLPoint? EDITWITH "None";
  from: REF State;
  to: REF State;
  anchorPoints: VAL VLPointArray? EDITWITH "None";
}
```

Abbildung 4: Spezifikation der abstrakten Struktur

```
RULE pVIEWROOT_statechartView: VIEWROOT ::= statechartView_StatechartDiagram END;
RULE pstatechartView_SimpleState: statechartView_SimpleState ::= ID
            statechartView_SimpleState_name END;
RULE pstatechartView_SimpleState_name: statechartView_SimpleState_name ::= END;
RULE pstatechartView_InitialState: statechartView_InitialState ::= ID  END;
RULE pstatechartView_XORState: statechartView_XORState ::= ID
            statechartView_XORState_subStates statechartView_XORState_name END;
RULE pstatechartView_XORState_subStates: statechartView_XORState_subStates
         LISTOF statechartView_XORState | statechartView_SimpleState |
         statechartView_FinalState | statechartView_InitialState END;
RULE pstatechartView_XORState_name: statechartView_XORState_name ::= END;
RULE pstatechartView_FinalState: statechartView_FinalState ::= ID  END;
```

Abbildung 5: Ausschnitt aus der Grammatik für Statechart Diagramme

Kardinalitäten Das Sternchen hinter der Typangabe gibt die Kardinalität an. Hier sind also beliebig viele Unterelemente möglich. Weitere Kardinalitäten und ihre Bedeutung:

- “*“ bedeutet, dass das Attribut eine Liste von Knoten speichert (End-Multiplizität 0..*, Editier-Multiplizität 0..*).
- “!“ bedeutet, dass das Attribut genau einen Knoten speichert, beim Editieren aber zwischenzeitlich auch leer sein kann (End-Multiplizität 1..1, Editier-Multiplizität 0..1).
- “?“ bedeutet, dass das Attribut höchstens einen Knoten der angegebenen Klasse speichert (End-Multiplizität 0..1, Editier-Multiplizität 0..1).
- “%“ bedeutet, dass genau ein Unterobjekt fest mit dem Attribut verbunden ist (End-Multiplizität 1..1, Editier-Multiplizität 1..1). Das Unterobjekt wird automatisch beim Erzeugen des übergeordneten Knotens angelegt. Damit das möglich ist, muss der Typ eine nicht-abstrakte Klasse sein.

Diagram ist eine abstrakte Klasse, es können keine Instanzen erstellt werden. *Statechart-Diagram* ist eine Unterklasse von *Diagram*. Mehrfachvererbung wird unterstützt. Von einer konkreten Klasse darf nicht geerbt werden.

StatechartDiagram enthält ein Attribut *setSize* vom Typ VLPoint. Weitere unterstützte Typen sind VLString, VLBoolean, VLLong, VLDouble, VLRegion, VLList und VLArray. Neben Aggregationen sind Referenzen auf andere Klassen zulässig. Referenzen werden durch ein Attribut vom Typ *REF* gekennzeichnet, wie in der Klasse Transition zu sehen. Attribute können bei der Initialisierung auf einen Vorgabewert gesetzt werden. Dies wird durch eine *INIT* Klausel ausgedrückt, wie beim Attribut *setSize* der Klasse StatechartDiagram zu sehen ist.

Diese klassenbasierte Struktur wird intern in eine kontextfreie Grammatik übersetzt und dann von Eli [10] weiterverarbeitet (in Abbildung 5 ist ein Ausschnitt des Statechart Beispiels zu sehen). Es ist zu sehen, dass der klassenbasierte Entwurf weitaus anwenderfreundlicher ist. Komplexe Beziehungen über mehrere Klassen hinaus sind

jedoch nicht gut zu erkennen. Dies wird besonders bei Klassenhierarchien deutlich, die viele Vererbungsbeziehungen beinhalten. Indirekte Unterklassen können so nur schwer erkannt werden. Dies gilt ebenso für zyklische Abhängigkeiten, also bei Klassen, die wiederum in ihren Unterklassen aggregiert enthalten sind. Die textuelle Darstellung wird zudem bei größeren Projekten schnell unübersichtlich.

Visuelle Muster Visuelle Muster stellen, wie in Abbildung 3 zu sehen, Teile der abstrakten Struktur graphisch dar. Sie charakterisieren spezifische visuelle Darstellungskonzepte, d. h. es wird beschrieben, wie eine bestimmte Informationsstruktur visuell repräsentiert werden kann. Ein häufig auftretendes visuelles Muster ist zum Beispiel das Mengenmuster (in DEViL *VPSet* genannt). Es beschreibt, dass ein graphisches Element Unterelemente enthält, die frei in einem bestimmten Bereich der graphischen Repräsentation positionierbar sind. Ein anderes Beispiel ist das Listenmuster (*VPSimpleList*). In der abstrakten Struktur existiert in diesem Fall eine endliche Folge von Elementen. Die visuelle Struktur ist die Aufreihung dieser Elemente in einer bestimmten Richtung des zweidimensionalen Darstellungsbereichs, wobei sich die Darstellungen der einzelnen Elemente nicht überschneiden. Die visuelle Darstellung erlaubt das Löschen von Elementen und das Einfügen neuer Elemente an den Zwischenpositionen. Sogenannte *Kontrollattribute*, die bestimmte Eigenschaften der Muster festlegen und vom Benutzer überschrieben werden, können das Layout der Muster variieren. Ein Kontrollattribut beim Listenmuster gibt zum Beispiel an, ob die Liste horizontal oder vertikal dargestellt werden soll.

Visuelle Muster beschreiben also nur konzeptionelle Eigenschaften einer visuellen Sprache und sind so allgemein, dass mit einer relativ kleinen Menge von Mustern eine große Zahl visueller Sprachen beschrieben werden kann [12]. Eine Sicht bezeichnet eine graphische Repräsentation der abstrakten Struktur. Zu einer gegebenen Struktur ist es durchaus möglich, mehrere Sichten zu definieren. Des Weiteren können auch separate Sichtmodelle erstellt werden. Mit diesen speziellen Sichtmodellen ist es möglich, ein eigenes abstraktes Modell, ein sogenanntes abgeleitetes Modell, zu spezifizieren, das auch nur in dieser Sicht zur Verfügung steht. In dieser Arbeit wird diese Methodik verwendet, um aus der Sicht für die Spezifikation des Modells eine geeignete Sicht für die Anwendung von visuellen Mustern zu generieren. Dabei wird die abstrakte Struktur eines Editors entsprechend transformiert. Dazu jedoch in Kapitel 5.1 mehr.

Wichtige visuelle Muster sind Listen (*VPSimpleList*), Mengen (*VPSet*), Formulare (*VPForm*), Tabellen (*VPTable*), Matrizen (*VPMatrix*), Linieverbindungen (*VPConnection*), Beschriftungen (*VPLabel*), Fließtext (*VPFlowRoot*) und Primitive (z.B. *VPTextPrimitive*) (Abb. 6). DEViL beinhaltet eine Bibliothek von konkreten Implementierungen dieser wichtigen visuellen Muster. Die Implementierungen konkretisieren das allgemeine Muster in spezifischer Weise. Jede Musterimplementierung beinhaltet dazu

- einen Mechanismus zur Parametrisierung der visuellen Darstellung, der auch die Grenzen des Musters bestimmt,

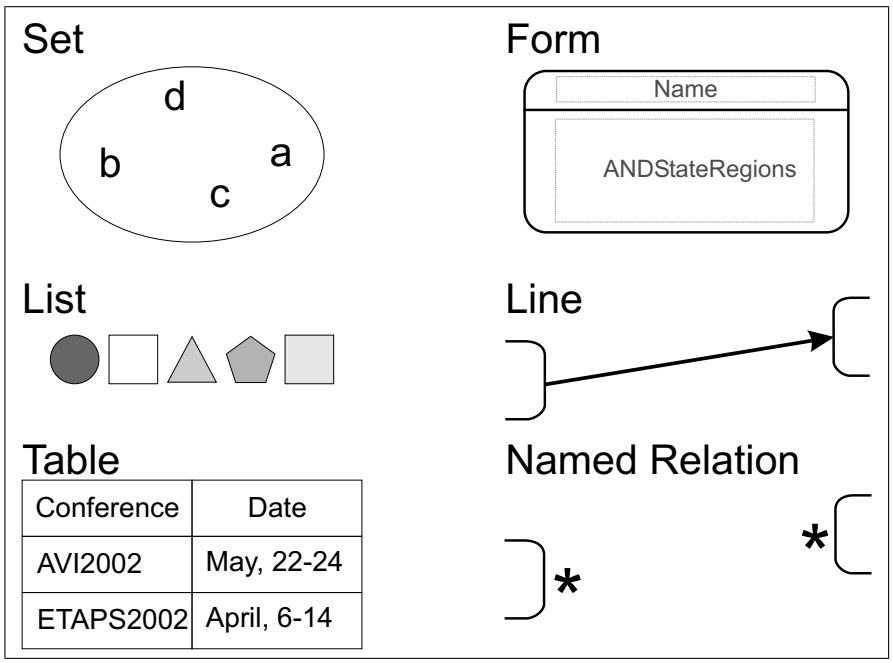

Abbildung 6: Visuelle Muster (aus [12])

- eine konkrete Realisierung des Layouts,
- eine konkrete Methode zur Interaktion,
- verschiedene Rollen, die an Strukturobjekte gebunden werden können.

Für einige Muster enthält DEViL mehrere Implementierungsvarianten, die sich dann in mindestens einem der oben genannten Punkte unterscheiden.

Anwendung visueller Muster Nun soll gezeigt werden, wie Muster bzw. Rollen an die abstrakte Struktur gebunden werden.

Visuelle Muster werden angewendet, indem bestimmte Grammatiksymbole der abstrakten Struktur von bestimmten Symbolrollen des visuellen Musters erben. Dies wird mittels attributierter Grammatiken erreicht, die Berechnungen im abstrakten Strukturbaum realisieren und durch den Baum transportieren. Attributierte Grammatiken werden in LIDO, einer funktionalen Sprache, spezifiziert.
Um z.B. auszudrücken, dass ein Zustandsdiagramm graphisch als “Menge“ und die Zustände graphisch als frei bewegliche “Mengenelemente“ repräsentiert werden sollen, muss das Grammatiksymbol “view_Statechart_states“ von “VPSet“ und das Grammatiksymbol “view_State“ von “VPSetElement“ erben. Durch Überschreiben von Attributen lassen sich Darstellungsdetails wie Farben oder Ausrichtungen anpassen. Dies soll an folgendem Beispiel eines Ausschnittes der abstrakten Struktur verdeutlicht werden:

```
CLASS Statechart {
```

```
 states: SUB State*;
}

CLASS State {
}
```

Das *VPSet* Muster wird nun angewendet um darzustellen, dass die Klasse *State* als Menge innerhalb der Klasse *Statechart* visuell repräsentiert werden soll. Nun der entsprechende Ausschnitt der attributierten Grammatik in LIDO:

```
stateChartView_Statechart INHERITS VPForm
COMPUTE
 SYNT.drawing = ADDROF("StatechartDrawing");
END;

stateChartView_Statechart_states INHERITS VPFormElement, VPSet
COMPUTE
 SYNT.formElementName = "statesContainer";
END;

stateChartView_State INHERITS VPForm, VPSetElement
COMPUTE
 SYNT.drawing = ADDROF("StateDrawing");
END;
```

In diesem Beispiel wird eine Sichtberechnung mit dem Namen *stateChartView* berechnet. Berechnungen für Klassen der abstrakten Struktur werden durch den Namen der Klasse mit dem vorangestellten Präfix des Sichtnamens eingeleitet (*stateChartView_Statechart*). Berechnungen für Attribute von Klassen der abstrakten Struktur werden eingeleitet, indem der Name der Sicht und die Klasse, in der sie enthalten sind, als Präfix deklariert wird (*stateChartView_Statechart_states*). Hier werden zwei visuelle Muster benutzt. Zunächst wird an die Klasse *Statechart* das *VPForm* Muster gebunden. Das *VPForm* Muster stellt ein Formular dar, das bestimmte Einfügestellen für Unterelemente, sogenannte Container, enthält. Mit dem Kontrollattribut

```
SYNT.drawing = ADDROF("StatechartDrawing");
```

wird dem Muster eine “generische Zeichnung“ zugewiesen. Auf generische Zeichnungen wird noch weiter unten in diesem Kapitel eingegangen. In einem zweiten Schritt wird dann das *VPSet* Muster auf die Mengenelemente *states* angewendet. Jede Klasse *State* hat dann wiederum ein *VPForm* Muster und die Rolle *VPSetElement*, die das *VPSet* Muster für Mengenelemente bereitstellt.

Struktur der visuellen Muster Wie bereits bei den Mustern *VPSet* und *VPForm* gesehen, bestehen visuelle Muster aus verschiedenen Rollen. Diese Rollen stellen

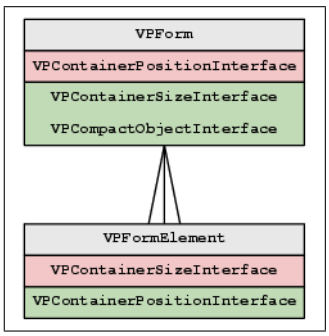

Abbildung 7: Rollendiagramm des *VPForm* Musters

spezifische Teileigenschaften des Musters graphisch dar. *VPSet* besteht aus *VPSet* und *VPSetElement*, *VPForm* aus *VPForm* und *VPFormElement*. Es gibt aber auch wesentlich komplexere Muster, die aus mehr Rollen bestehen und deren Rollen auch nicht an allen Knoten angewendet werden dürfen. Zur Spezifikation von Mustern werden sogenannte *Rollendiagramme* verwendet.

Diese Spezifikationen basieren auf der Implementierung der Muster und werden üblicherweise nicht vom DEViL Benutzer angewendet. Es soll lediglich möglich sein, auch später entworfene Muster dem DEViL-Designer komfortabel hinzuzufügen. In Abbildung 7 ist das Rollendiagramm des *VPForm* Musters zu sehen. Es besteht aus der Rolle *VPForm* und der untergeordneten Rolle *VPFormElement*. Die drei aufgefächerten Linien zwischen den Rollen geben an, dass unter einer Instanz der *VPForm* Rolle beliebig viele *VPFormElement* Rollen hängen dürfen. Ist hier nur eine Linie eingezeichnet, so bedeutet dies, dass sich nur genau eine Instanz dieser Rolle unterhalb befinden darf. Die Rollendiagrammknoten sind neben dem Namen der Rolle in zwei weitere Bereiche unterteilt. In beiden Bereichen stehen Namen von Interfaces. Interfaces geben Auskunft über bestimmte Eigenschaften einer Rolle, wie z.B. Größe und Position. Diese Eigenschaften können dann von anderen Rollen gelesen werden. Die grünen Bereiche (unterer Teil) geben an, welche Interfaces die Rolle bereitstellt, die roten Bereiche (oberer Teil) zeigen an, welche Interfaces benötigt werden. In diesem Fall benötigt die *VPForm* Rolle noch das Interface “VPContainerPositionInterface“. Es muss also an demselben Knoten, an dem die *VPForm* Rolle angewendet wurde, noch eine zusätzliche Rolle gehängt werden, die genau dieses Interface bereitstellt, im grünen Bereich also diesen Interface Namen stehen hat. In diesem Fall wäre die Rolle *VPSimpleListElement* eine gute Wahl.

Zwischenknoten Zu jeder Sicht werden Grammatikproduktionen generiert, auf deren Basis dann die graphische Darstellung spezifiziert wird. Die Eigenschaften der Grammatik lassen sich den Anforderungen der jeweiligen Sicht anpassen. Aus dem oben eingeführten Beispiel, das Statechart Diagramme spezifiziert, wird beispielsweise automatisch folgende Grammatikabbildung generiert:

```
Root(diagrams);
Diagram(names);
```

```
StatechartDiagram(setSize, states, transitions);
State(position);
InitialState();
FinalState();
SimpleState(name);
XORState(name, subStates, setSize)
Transition(position, from, to, anchorPoints);
```

In der Grammatikabbildung werden Klassen der abstrakten Struktur dargestellt und deren Attribute aufgelistet. *StatechartDiagram(setSize, states, transitions)* bedeutet somit, dass die Klasse StatechartDiagramm drei Attribute besitzt, nämlich *setSize*, *states* und *transitions*. Von welchem Typ diese Attribute sind kann hier nicht ausgedrückt werden.

Diese Grammatikabbildung kann modifiziert werden, indem zum Beispiel Attribute weggelassen werden, die für das Anwenden visueller Muster nicht benötigt werden. Dies erzeugt performantere Strukturediteren, da der Umfang der Grammatik so verkleinert werden kann. DEViL bietet des Weiteren die Möglichkeit, sogenannte Zwischenknoten zu definieren. Wird beispielsweise das Baummuster angewendet, so sind Zwischenknoten nötig, um einen Wurzelknoten zu erstellen:

```
MyView(treeRoot(rootNode(subnodeList)));
```

Hier werden zwei zusätzliche Knoten, *treeRoot* und *rootNode*, eingeführt, die in der abstrakten Struktur nicht existieren. In der Sichtdefinition können diese Knoten nun genauso benutzt werden wie bereits existierende Baumknoten, es können zusätzliche Rollensymbole angewendet werden.

Spezifikation textueller Bestandteile Bei einigen Strukturediteren kann es nötig sein, dass Teile der abstrakten Struktur textuell dargestellt werden sollen, weil diese Darstellungsweise möglicherweise ausdrucksstärker ist. Ein Beispiel wäre die Spezifikation von Funktionen und Funktionsaufrufen einer beliebigen Programmiersprache, wobei Funktionsdefinitionen als Text dargestellt werden sollen. Hierfür bietet DEViL eine eigene Sprache zur Beschreibung an:

```
XORState*: "XORState:" "!SPACE" name ", " subStates ";".
```

Dies ist zum Beispiel die textuelle Repräsentation eines Sprachelements *XORState*. *XORState* ist in der abstrakten Struktur durch eine Klasse vertreten. Es hat unter anderem die Attribute *name* und *subStates*. Hier werden also textuell der Name und die Unterelemente eines Sprachelements *XORState* ausgegeben. Es ist zu sehen, dass Attribute zusammen mit Zeichenketten (in Anführungszeichen) dazu verwendet werden können, bestimmte Sprachelemente zu beschreiben. Diese spezielle Repräsentationsvariante soll auch in den DEViL-Designer einfließen.

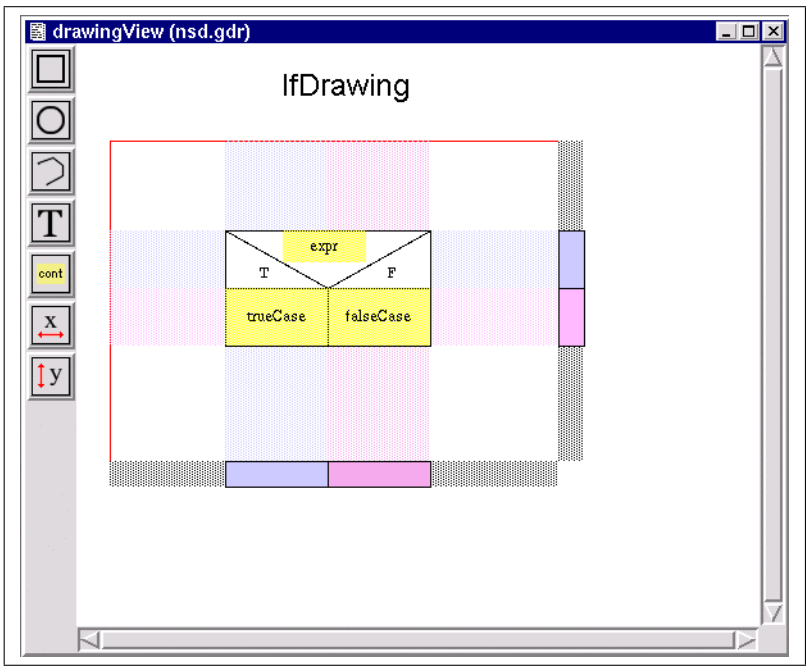

Abbildung 8: Der Editor für generische Zeichnungen (GDED)

Generische Zeichnungen Bei generischen Zeichnungen handelt es sich um graphische Spezifikationen, mit denen sich Ausprägungen bestimmter Formular-Darstellungen einfach und komfortabel beschreiben lassen. Diese Zeichnungen können bei Anwendung des Formularmusters durch das Kontrollattribut

SYNT.drawing = ADDROF(Name der Zeichnung);

referenziert werden. Die Zeichnungen bestehen aus drei Teilen:

- Grafische Primitive wie Linien, Rechtecke oder Kreise legen das grundlegende Erscheinungsbild des Fomulars fest. Die Zeichnung in Abbildung 8 besitzt beispielsweise einige Rechteck- und Linien-Elemente, sowie die Text-Primitive “T“ und “F“.

- Container (gelb dargestellt) bestimmen, wie die Unterelemente des Formulars positioniert werden sollen. Die Zeichnung aus Abbildung 8 hat die Container “expr“, “trueCase“ und “falseCase“. In Eigenschafts-Dialogen kann die Ausrichtung des Containerinhalts festgelegt werden.

- Dehnungsintervalle bestimmen, wie die Zeichnung transformiert werden soll, um Platz für den Inhalt der Container zu schaffen. Die Zeichnung aus Abbildung 8 enthält je zwei Dehnungsintervalle auf der X- und Y-Achse. Damit wird ausgedrückt, dass die entsprechenden Bereiche unabhängig voneinander gedehnt werden können.

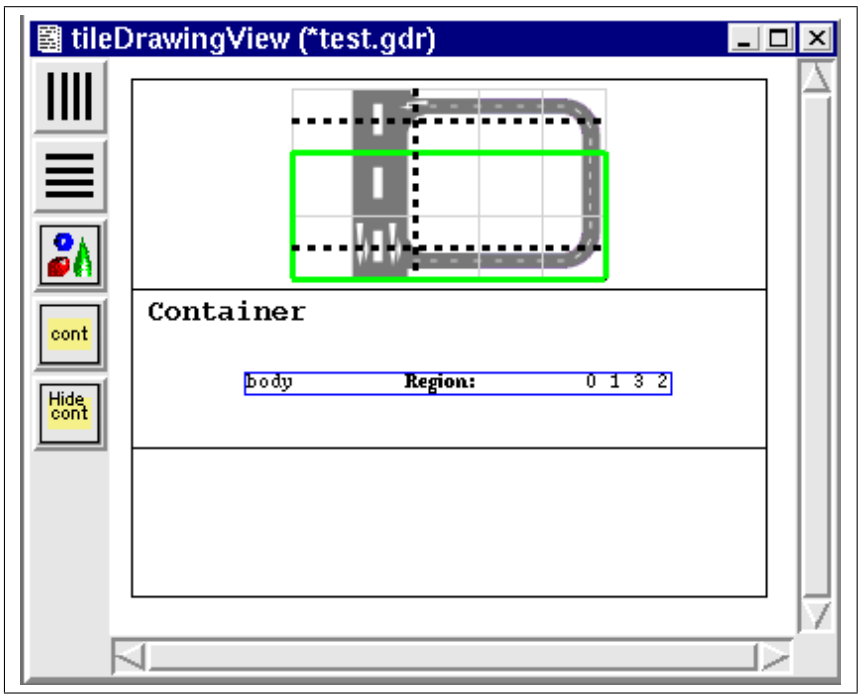

Abbildung 9: Der Editor für Kachelzeichnungen

Im Gegensatz zu Vektor-Zeichnungen basieren Kachel-Zeichnungen auf Pixelgrafik-Kacheln, die zusammengesetzt ein bestimmtes Bild ergeben. Kacheln können dabei mehrfach dupliziert werden, um den Platzbedarf von Unterobjekten zu befriedigen. Abbildung 9 zeigt eine Kachel-Zeichnung, die das Aussehen von Schleifen in der Sprache “Streets“ [19] beschreibt. Das grüne Rechteck zeigt einen Container, in dem der Schleifenkörper dargestellt werden soll. Die schwarz gestrichelten horizontalen und vertikalen Linien sind sogenannte Ausdehnungslinien. Sie entsprechen in gewisser Weise den Dehnungsintervallen in Vektor-Zeichnungen. Durch die Ausdehnungslinie wird spezifiziert, dass die entsprechende Zeile oder Spalte der Matrix beliebig oft repliziert werden kann, um den Container zu vergrößern. Die Funktionalitäten der Sprachelemente entsprechen im Wesentlichen denen der Vektor-Zeichnung. So können Containern Ausrichtungen zugewiesen und auch unsichtbare Container hinzugefügt werden.

Abgeleitetes Modell und Kopplung DEViL bietet die Möglichkeit, Teile des Modells aneinander zu koppeln und modifiziert in den Sichten wieder zu verwenden. Dies ist u. a. nötig, wenn verschiedene Sichten zueinander konsistent gehalten werden sollen. Für jeden Objektknoten einer Instanz des Modells, auch Basismodell genannt, existiert ein Objektknoten in der abgeleiteten Struktur, wie in Abbildung 10 zu sehen. Dieser Objektknoten kann dabei, falls nötig, eine vollkommen andere interne Struktur als der Ausgangsknoten im Modell besitzen. Es existiert jedoch immer eine Referenz auf den Knoten im Basismodell. Ist diese Referenz nicht gesetzt, deutet das darauf hin, dass der Knoten im Basismodell nicht mehr existiert. Die Knoten im abgeleiteten Modell werden dann automatisch gelöscht. Intern wird für den Kopplungsmechanismus eine Tcl [14] Funkti-

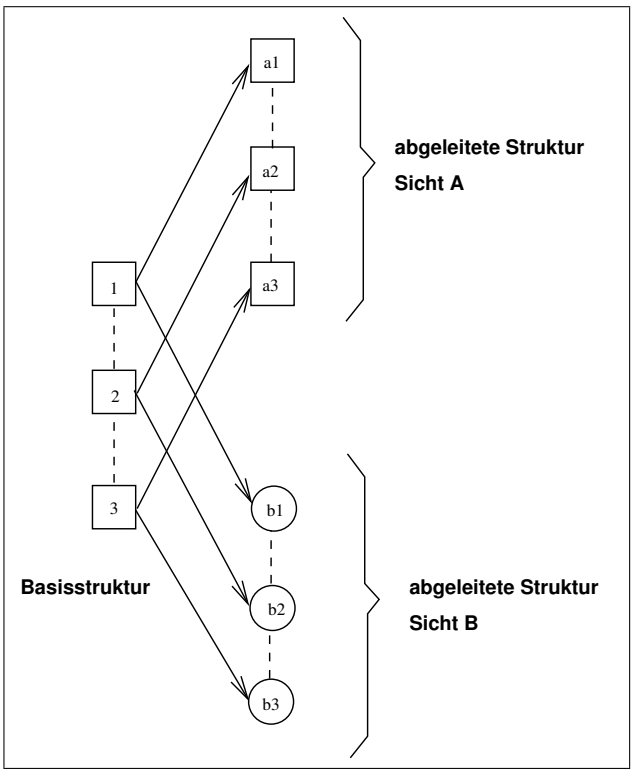

Abbildung 10: Abgeleitetes Modell

on aufgerufen, die zwei Listen übergeben bekommt und überprüft, ob für jedes Element der ersten Liste auch ein Element in der zweiten Liste existiert. Näheres zur Kopplung in Kapitel 5.1.

Codegenerierung DEViL bietet für jeden Editor die Möglichkeit, beliebig viele Codegenerierungmodule zu spezifizieren. Instanzen von Objekten im Strukturbaum können mit Textfragmenten kombiniert werden, die dann bei der Codegenerierung zusammengefügt werden. Für jedes Objekt im Strukturbaum kann in einer LIDO Datei eine attributierte Grammatik spezifiziert werden, die mit von PTG generierten Funktionen kombiniert werden. PTG ist ein Generator zur strukturierten Textausgabe. Die Übersetzung kann in jede beliebige Zielsprache erfolgen. Der Zieltext wird durch Text-Muster spezifiziert, die durch Funktionen in LIDO aufgerufen werden können. PTG fügt während der Codegenerierung Attribute und Textfragmente zusammen und gibt diese in eine Datei aus.

In Abbildung 11 ist ein Teil der Codegenerierungsspezifikation für Statechart Diagramme zu sehen. Dabei werden einem Strukturobjekt *StatechartDiagram* spezielle Attributberechnungen zugeordnet. Das Präfix *codeGen* gibt dabei den Namensraum des Codegenerierungsauswerters an. In Zeile eins und zwei werden zunächst zwei Attribute definiert. Das zweite ist vom Typ *PTGNode*. Ein *PTGNode* ist ein Typ, der beliebig viele Textfragmente enthalten kann. In Zeile vier wird die Berechnungsumgebung eingeleitet.

```
1. SYMBOL codeGen_StatechartDiagram: myVar: VLString;
2. SYMBOL codeGen_StatechartDiagram: code: PTGNode;
3. SYMBOL codeGen_StatechartDiagram
4. COMPUTE
5.  SYNT.code = PTGStatechartDiagram(THIS.pers_name,
6.                 CONSTITUENTS codeGen_State.code
7.                   SHIELD codeGen_XORState.code
8.              WITH (PTGNode, PTGNewLine, IDENTICAL, PTGNull));
9. END;
```

Abbildung 11: Spezifikation der Codegenerierung

```
StatechartDiagram:
"Name: " $1 string [IndentNewLine]
$2
```

Abbildung 12: Textmuster zur Codegenerierung

Für die lokale Variable *code* wird die PTG-Funktion “PTGStatechartDiagram“ mit zwei Attributen aufgerufen. Das erste *THIS.pers_name* ist ein persistentes Attribut des Strukturobjektes StatechartDiagram. Das zweite Attribut ist der PTGNode *codeGen_State.code*, also das lokale Attribut “code“ des Strukturobjekts “State“. Da es viele Unterknoten vom Typ “State“ geben kann, muss die *CONSTITUENTS* Klausel angegeben werden. Zeile acht besagt, dass “code“ vom Typ PTGNode ist und die verschiedenen “State“ Knoten mit einer neuen Zeile (PTGNewLine) verknüpft werden sollen. Eine Sequenz oder eine kommaseparierte Sequenz ist beispielsweise ebenso möglich. Die *SHIELD* Klausel besagt, dass nicht beliebig viele “State“ Knoten berücksichtigt werden sollen. So sollen die “State“ Knoten, die unterhalb des Strukturobjekts “XORState“ im Strukturbaum liegen, nicht aufgeführt werden. In Abbildung 12 ist das PTG Muster, das in Abbildung 11, in Zeile fünf, aufgerufen wurde, zu sehen. Die übergebenen Attribute werden durch PTG im Text übersetzt. Zugriff auf die Attribute erfolgt durch ein Dollarzeichen gefolgt von ihrer Position im Funktionsaufruf.

Attributzugriff im Strukturbaum In DEViL kann mit Hilfe einfacher Konstrukte auf Attribute im Strukturbaum zugegriffen werden. Dies ist nötig, um z.B. auf entfernte Attribute in Sichtberechnungen zugreifen oder in der Codegenerierung beliebige Objekte erreichen zu können. In einer Sichtberechnung eines Knotens kann es nötig sein, dass das Layout von Bedingungen eines Attributknotens abhängt, der in einem Schwesterknoten hängt. Ein Listenknoten könnte so zum Beispiel je nach Setzen eines boolschen Attributs ein- oder ausgeblendet werden. In den Dateien der Sicht- oder Codeberechnungen können Spezialkonstrukte eingesetzt werden:

Mittels LIDO CONSTITUENTS kann im Strukturbaum auf Attribute in Unterbäu-

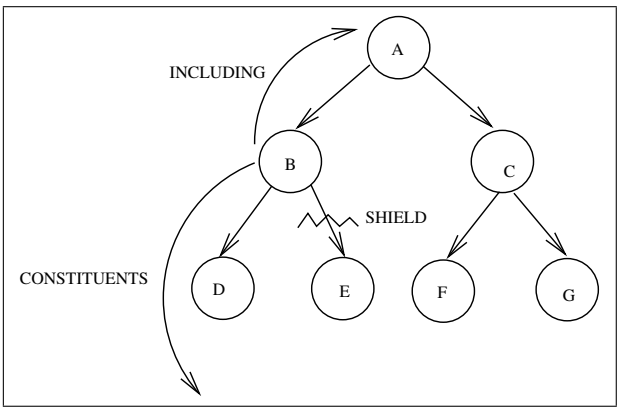

Abbildung 13: Attributzugriff im Strukturbaum

men und mittels LIDO INCLUDING auf Attribute zugegriffen werden, die höher im Strukturbaum liegen. Wie in Abbildung 13 zu sehen, kann vom Knoten B mit CONSTITUENTS zu allen Knoten unterhalb von B gelangt werden. Mittels der SHIELD Klausel können Teilbäume abgeschnitten werden. Diese Teilbäume werden bei der Codegenerierung dann nicht mehr berücksichtigt. Mit INCLUDING kann Knoten A erreicht werden. Soll in Schwesterbäume navigiert werden, z.B. Knoten C, F oder G, so muss am Knoten A eine sogenannte “Property“ gesetzt werden. In dieser Property wird dann der Wert des Schwesterknotens gespeichert und kann dann mit Hilfe eines Methodenaufrufs in Knoten B abgefragt werden. Diese beiden Konstrukte können mit Pfadausdrücken kombiniert werden, auf die später noch genau eingegangen werden soll. Dazu ein Beispiel:

```
stateChartView_Statechart_states INHERITS VPFormElement, VPSet
COMPUTE
 SYNT.formElementName = "statesContainer";
END;

stateChartView_State INHERITS VPForm, VPSetElement
COMPUTE
 SYNT.drawing = ADDROF("StateDrawing");
 SYNT.hide = INCLDUDING SystemProperties.hideList;
END;
```

In diesem Beispiel wird der Knoten *State* nur dargestellt, wenn das Kontrollattribut *SYNT.hide* der *VPForm* Rolle gesetzt ist. Die Berechnung hängt dabei von einem Attribut *hideList* ab, das sich in einer Klasse *SystemProperties* oberhalb der Klasse *State* befindet.

3.3 Systeme zur Spezifikation graphischer Strukturediteren

In diesem Abschnitt sollen einige existierende Generatoren für graphische Strukturediteren untersucht werden. Insbesondere soll darauf geachtet werden, ob Spezifikations-

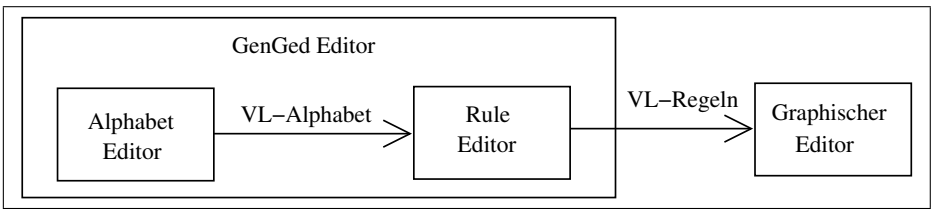

Abbildung 14: Die GenGed Umgebung (aus [15])

konzepte für diese Arbeit übernommen werden können.

3.3.1 GenGed

GenGed [15] ist ein graphischer Editor zur Spezifikation von visuellen Sprachen. Aus einem gegebenen Alphabet und Regeln einer spezifischen visuellen Sprache erzeugt GenGed einen graphischen Editor (siehe Abb. 14). GenGed basiert auf algebraischen Graph Grammatiken, wobei eine graphische Struktur durch eine logische Ebene (abstrakte Syntax) und eine visuelle Ebene (konkrete Syntax) definiert wird. Die Verbindung zwischen diesen beiden Ebenen erfolgt durch Layout Operationen.

Algebraische Spezifikationen werden benutzt, um graphische Symbole, Beziehungen und Layout-Bedingungen zu definieren. Die eigentliche Sprache, d. h. ihre grammatikalische Struktur, wird durch Graphtransformations-Regeln beschrieben. Um eine graphische Sprache zu spezifizieren, wird zunächst das sogenannte VL-Alphabet erstellt. Der *Alphabet* Editor wiederum besteht aus mehreren Komponenten wie in Abbildung 16 zu sehen ist. Das VL-Alphabet entspricht den Klassen, Attributen, Referenzen und der graphischen Repräsentation in DEViL. Im *GraphicalObjectEditor* werden das optische Layout und das Aussehen der Beziehungen zwischen den Objekten spezifiziert. Im *TypiEditor* werden die graphischen Elemente dann optisch in Baumknoten verwandelt. Hier findet dann auch die Zuweisung von Objekten an Knoten oder Kanten statt. Objektnamen werden abgekürzt dargestellt. Dies dient der Übersicht im *ConEditor*, in dem die Beziehungen zwischen Objekten definiert werden. Im Anschluss werden für jedes Objekt graphische Regeln generiert, die im *RuleEditor* verändert werden können. Bei diesen Regeln steht auf der linken Seite eine Vorbedingung, die gelten muss. Diese Vorbedingung wird auf alle Vorkommen im Strukturbaum angewendet, die diese Bedingung erfüllen. Wenn die Regel angewendet wird, werden alle Vorkommen der linken Seite im Strukturbaum durch die rechte Seite ersetzt. Im Beispiel aus Abbildung 15 wird in eine leere Umgebung ein State Objekt eingefügt. Der Strukturbaum repräsentiert in der Vorbedingung also eine leere Liste, in die dann State Objekte eingefügt werden können.

Fazit GenGed stellt mit dem *RuleEditor* ein intuitiv zu benutzendes Werkzeug zur Spezifikation von VL Regeln zur Verfügung. Die visuelle Modellierung von Grammatikableitungen wird unmittelbar verständlich. Hier wird jedoch ein Nachteil von GenGed deutlich: für jede Grammatikregel muss eine VL-Regel erstellt werden. Nicht erreichba-

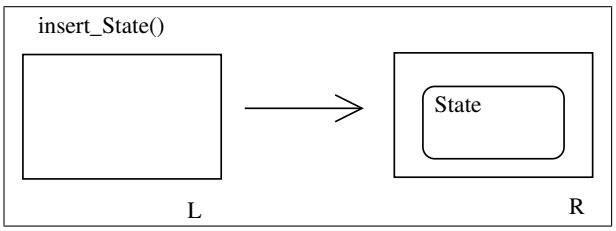

Abbildung 15: Eine VL-Regel: Die linke Seite ist leer (Vorbedingung), es kann also ein neues Objekt eingefügt werden.

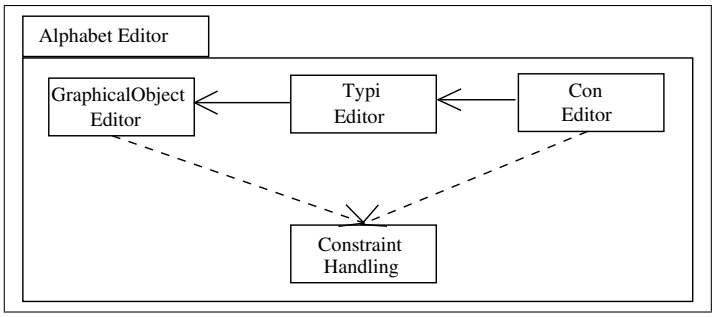

Abbildung 16: Komponenten des Alphabet Editors (aus [15])

re Regeln und immer wiederkehrende Operationen, wie das Einfügen von Objekten in eine Liste, müssen so immer wieder modelliert werden. Die Konzepte von GenGed und DEViL sind sehr verschieden: Die Berechnung der graphischen Repräsentation basiert bei GenGed auf Regeln, während bei DEViL Strukturobjekten Layoutrollen zugeordnet werden.

3.3.2 MetaEdit+

MetaEdit+ [16] ist ein kommerzielles Produkt der Firma MetaCase. Anhand eines Beispiels aus dem Tutorial von MetaEdit+, bei dem ein Familienstammbaum modelliert wird, soll die Anwendung von MetaEdit+ gezeigt werden. MetaEdit+ besteht aus fünf Werkzeugen mit denen Editoren spezifiziert werden. Mit dem *Object Editor* aus Abbildung 17 werden zunächst Objektschablonen angelegt, die den DEViL Klassen entsprechen. In einem Eigenschaften Dialog werden dann Attribute hinzugefügt. Im *Symbol Editor* werden den angelegten Objekten dann Zeichnungen (Symbole) zugewiesen. Dies erfolgt analog zu DEViL Formularen und generischen Zeichnungen, mit dem Unterschied, dass die Symbole keine Container mit Unterelementen enthalten können. Im *Relationship Tool* und dem *Role Tool* werden dann “Bindungen“ zwischen den Symbolen festgelegt. Auch dies ähnelt dem DEViL-Konzept. Im *Graph Tool* werden schließlich Referenzen zwischen Objekten hergestellt, wie in Abbildung 19 zu sehen. Ein Report Generator kann vordefinierte “Berichte“ zu Instanzen des Modells erstellen. Eine Codegenerierung ist hier nur sehr eingeschränkt möglich.

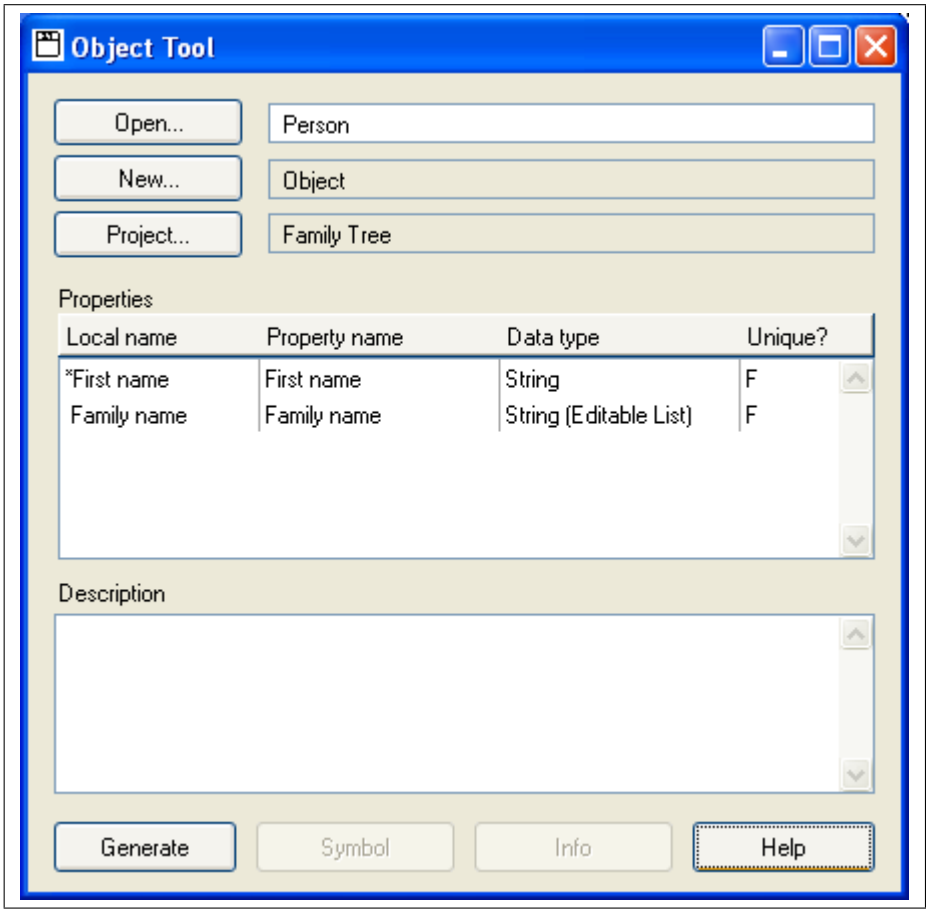

Abbildung 17: MetaEdit+ *Object Editor*: Erstellen eines Objekts “Person“ mit zwei Attributen: “Family Name“ und “First Name“

Fazit Mit MetaEdit+ können einfache Strukturediteren schnell spezifiziert werden. Allerdings können Unterelemente, wie in DEViL SUB Knoten, nicht modelliert werden. Das Anlegen von Objekten und die Zuweisung von Symbolen ist einfach und kann als Vorlage für das Modellieren von Klassen der abstrakten Struktur in dieser Arbeit benutzt werden. Das Relationship Tool sowie das Role Tool ist allerdings schwer zu benutzen. Hier würde eine textuelle Repräsentation wie in DEViL sogar leichter erlernbar sein.

3.3.3 DiaGen

DiaGen [17] ist ein an der Universität München entwickeltes System zur Generierung von Diagrammeditoren. Es basiert vollständig auf Java und besteht im Wesentlichen aus einem Generator und diversen Frameworkklassen zur Visualisierung von Diagrammkomponenten. Diese Frameworkklassen können vom Benutzer beliebig erweitert werden. An einem Diagrammeditor für Wurzelbäume (siehe Abbildung 20) soll hier nun der grundlegende Spezifikationsprozess erläutert werden. Als Eingabe bekommt der Generator zunächst eine Sprachdeklaration:

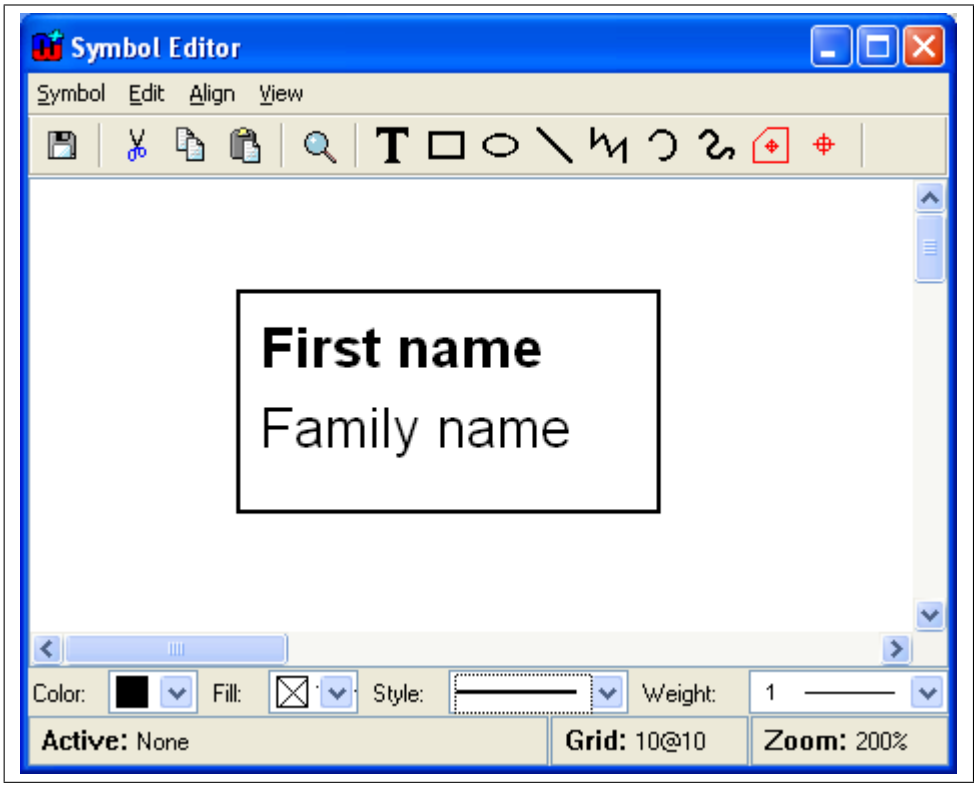

Abbildung 18: MetaEdit+ *Symbol Editor*: Zuordnen einer Zeichnung zu einem Objekt

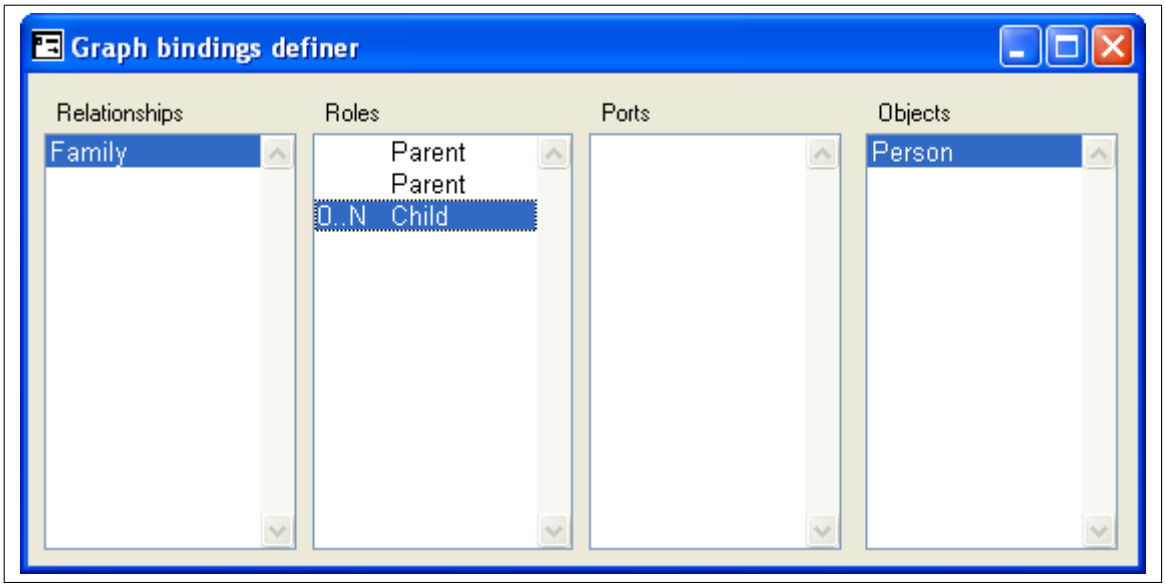

Abbildung 19: MetaEdit+ *Graph Editor*: Zuordnen von Bindungen und Kardinalitäten zwischen vorher definierten Rollen

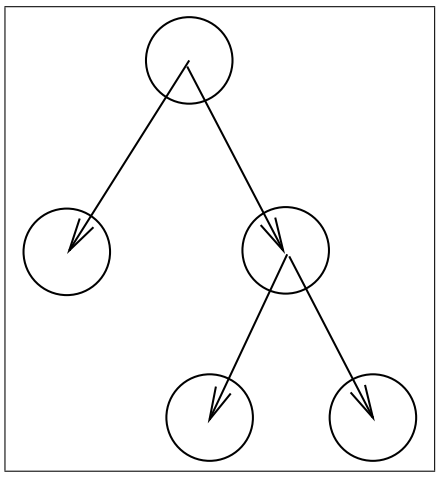

Abbildung 20: DiaGen Beispiel: Wurzelbaumeditor

```
package diagen.editor.sample.tree;
  component
    circle[1] { Circle[CircleArea] },
    arrow[2]  { Arrow[ArrowEnd, ArrowEnd] };
  relation
    inside[2] { ArrowInside[ArrowEnd, CircleArea] };
  terminal
    tNode[1]  { String name;
                Variable xpos, ypos, radius; },
    tChild[2];
  nonterminal
    Tree[1]   { Semantics.Node root;
                Variable top, left, right, x_root; },
    Subtrees[2] { java.util.List subtrees,
                  Variable top, left, right,
                    left_root, right_root; };
  constraintmanager
      diagen.editor.param.QocaLayoutConstraintMgr;
```

Zunächst wird das Java Paket deklariert, das den generischen Editor enthalten wird. Danach werden die so genannten Hyperkantentypen deklariert (DiaGen benutzt intern Hypergraphen). Circle und Arrow sind in diesem Fall die Komponenten, deren Visualisierung in automatisch generierten Frameworkklassen realisiert sind. Diese Frameworkklassen können, falls gewollt, durch eigene Implementierungen überschrieben werden. Arrow, also ein Pfeil, wird durch zwei Pfeilenden (ArrowEnd) spezifiziert. Eine Relation (inside) wird durch die beiden Endpunkte (ArrowEnd, CircleArea) spezifiziert. Es kann also nur eine Verbindung zwischen einem Kreis und einem Pfeilende geben. Diese Beschreibung von graphischen Relationen zwischen Objekten ist ein großer konzeptioneller Unterschied zu DEViL, wo diese Berechnungen nicht explizit ausgedrückt

werden müssen. Die Spezifikation der Terminale und Nicht-Terminale dient der späteren Verwendung in einer Grammatik. In geschweiften Klammern werden ihnen Attribute zugeordnet. Attribute vom Typ *Variable* sind Constraintvariablen. In der letzten Zeile wird der verwendete Constraintmanager spezifiziert, hier ist es QOCA [21], PARCON [20] kann ebenfalls benutzt werden.

Danach wird zur Laufzeit die lexikalische Analyse ausgeführt, die mittels Reduktionsregeln den internen Hypergraph in einen reduzierten Hypergraph (rHGM) verwandelt. Dieser rHGM beschreibt, wie Komponenten des Editors zusammenhängen:

```
reducer {
 ar:arrow(b,c) inside(b,a) inside(c,d) c1:circle(a)
    c2:circle(d)
    ==> tChild(a,d)
    {
      constraints: ar.xStart == c1.xAnchor; ar.yStart
                        == c1.yAnchor;
                   ar.xEnd == c2.Anchor; ar.yEnd
                        == c2.yAnchor;
                   ar.startRadius == c1.radius;
                   ar.endRadius == c2.radius;
    };
}
```

Der Ausschnitt aus den Reduktionsregeln weist den Editor an, zu erkennen, wann ein Pfeil zwei Kreise verbindet, wobei jede der Pfeilenden in einem der Kreise liegt. Dann wird eine *tChild*-Kante zwischen den entsprechenden Knoten gezogen. Die constraints berechnen die Pfeillänge. Diese ist nicht immer gleich, da die Größe der Kreise nicht vorbestimmt ist.

Schließlich wird die Grammatik spezifiziert um festzulegen, wie die Struktur zusammenhängt. Hier ist es zusätzlich möglich, den Grammatiksymbolen Attributauswerter und Layoutregeln hinzuzufügen.

Fazit Die Spezifikation eines Strukturediteors mit DiaGen ist schwierig, da sie rein textuell erfolgt. Während die Spezifikation der Sprachstruktur einem Kenner von Grammatiken noch recht einfach gelingt, sind die Reduktionsregeln und die Spezifikation von Layoutbedingungen sehr anspruchsvoll, da sie zum Teil mit mathematischen Ausdrücken modelliert werden müssen. Ein Vergleich zu DEViL ist hier nur schwer möglich. Sicherlich ist der Entwurfsaufwand mit DEViL im graphischen Teil wesentlich geringer, da das Layout nur konzeptionell beschrieben wird.

```
if ( num == 1 ) {
   double x = pos[0].getX();
   double y = pos[0].getY();
   double w = cur.getX()-x;
   double h = cur.getY()-y;
   java.awt.geom.Rectangle2D r =
      new java.awt.geom.Rectangle2D.Double(x,y,w,h);
   g2d.draw(r);
}
```

Abbildung 21: Ausschnitt aus der Spezifikation eines Statechartobjekts mit dynamischer Ausdehnung

3.3.4 DiaGen Designer

DiaGen Designer [18] ist die Weiterentwicklung von DiaGen. Der Designer erlaubt eine weitgehend visuelle Entwicklung von Strukturediteren. Sogenannte Hypergraph Grammatiken bilden die Sprache eines Strukturediteors. Sie sind mit den VL-Regeln von GenGed vergleichbar, da sie ebenfalls aus Vorbedingung und einem daraus folgenden Resultat bestehen. Der DiaGen Designer besteht dabei aus folgenden Teilen:

- Globale Definitionen: dient zur Spezifikation von globalen Konstanten wie Farben oder Schriftarten.
- Components: legt visuelle Komponenten wie z.B. Kreise oder Linien und deren “Attachmentareas“ fest. Attachmentareas kennzeichnen Bereiche von Komponenten, die eine Verbindung mit anderen Komponenten eingehen können.
- Relations: definiert Verknüpfungen zwischen Objekten: werden intern durch ein *Hypergraph Model (HGM)* dargestellt.
- Links: zusätzliche interne Hyperkanten im HGM, die es z.B. erlauben, zusätzliche Label an Objekte zu binden.
- Terminale u. Nicht-Terminale: Das HGM besteht aus Terminalen und Nicht-Terminalen, die hier mit zusätzlichen Attributen verbunden werden können.
- Reducer: dient zur Definition der bereits oben erwähnten Hypergraph Grammatiken.

Fazit Der DiaGen Designer bietet bei der Definition von Komponenten ein einfach zu bedienendes Frontend. Eigenschaften wie Komponententyp und die Definition von Attachmentareas sowie Grammatikabbildungen müssen immer noch von Hand spezifiziert werden. Generische Zeichnungen und insbesondere der dort implementierte Dehnungsmechanismus für Container müssen im DiaGen Designer vom Benutzer, wie in Abbildung 21, mathematisch beschrieben werden. Auch die große Anzahl von Werkzeugen zur

Spezifikation ist zunächst verwirrend. Insgesamt scheint der DiaGen Designer nur eine Oberfläche für die Spezifikationsprachen des ursprünglichen DiaGens zu sein.

4 Sprachentwurf

In diesem Kapitel sollen die Anforderungen der zu entwickelnden Umgebung herausgearbeitet werden. Dabei soll zunächst auf allgemeine Entwurfsprinzipien für visuelle Sprachen eingegangen werden, anschließend sollen konkrete Anforderungen an die Umgebung gestellt werden.

4.1 Entwurfsprinzipien

Das Anwenden einer textuellen oder einer visuellen Programmierumgebung bedeutet immer eine Abbildung von einer Problemwelt in eine Programmwelt. Ist der Abstand zwischen diesen beiden Ebenen klein, so ist der Aufwand für den Benutzer gering. In einer visuellen Programmiersprache muss also ein geeignetes Abstraktionsniveau gefunden werden. Dies kann manchmal recht schwierig sein, wie in Abbildung 22 zu sehen ist. Die

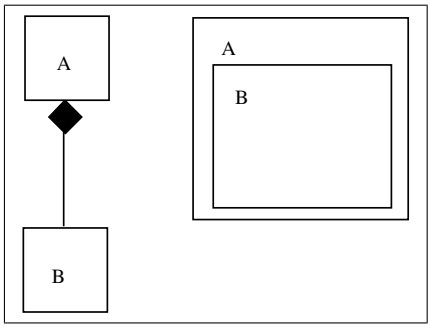

Abbildung 22: Unterschiedliche Abstraktionsniveaus

beiden Darstellungen sollen die Aggregation einer Klasse B in eine Klasse A zeigen. Dies ist sicherlich klarer zu erkennen als in der rein textuellen Darstellung:

```
CLASS KlasseA {
  bKlassen: SUB KlasseB*;
}

CLASS KlasseB {
}
```

Die rechte Darstellung in der Abbildung ist sicherlich die intuitivere, da eine Aggregation explizit ausgedrückt wird. Die Darstellung wird jedoch unpraktisch, wenn noch Beziehungen (verdeutlicht durch Linien) und weitere Schachtelungsebenen hinzukommen. Dann ist die linke Darstellung, trotz des Kompromisses, die bessere. Insbesondere kann eine Struktur der Form

```
CLASS KlasseA {
 bKlassen: SUB KlasseB*;
}
```

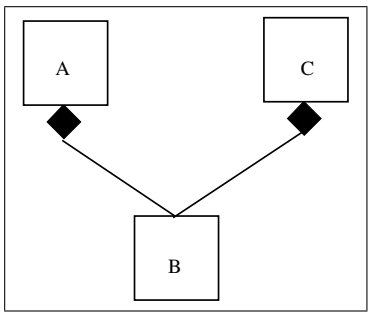

Abbildung 23: Darstellung bei Aggregation

```
CLASS KlasseC {
 bKlassen: SUB KlasseB*;
}

CLASS KlasseB {
}
```

nicht mehr durch graphische Schachtelung ausgedrückt werden, wenn Objekte nicht dupliziert werden sollen. Hier ist nur eine Darstellung, wie in Abbildung 23 zu sehen möglich.

Auch der Platzbedarf muss bedacht werden. Scrollen und das Benutzen von vielen Fenstern führen dazu, dass der Anwender Teile seines Entwurfs nicht mehr sieht und sich erinnern muss. Damit der Platz optimal ausgenutzt wird, ist es sinnvoll, Teile ausblenden zu können oder das Layout und die räumliche Struktur entsprechend anzupassen. Um dem Anwender unnötige Mühe zu ersparen, ist es von Vorteil, wenn die Anzahl der verwendeten Symbole möglichst gering ist ([1]). Ein Anwender kann sich in der Regel etwa sieben (± 2) Symbole merken ([1]). Dies ist somit schon eine gute obere Grenze für die Anzahl der Symbole, die in der zu entwickelnden Umgebung eingesetzt werden sollen. Auch sollten in einer visuellen Sprache nicht zu viele textuelle Bestandteile vorkommen ([1]). Eine Sequenz von Java Anweisungen, die mit Linien verbunden wird, ist sicherlich nicht wünschenswert. Textuelle Bestandteile sollten sich, wenn möglich, ausschließlich auf Namen von Objekten beschränken. Falls textuelle Bestandteile trotzdem benötigt werden, so ist es sinnvoll, diese Bestandteile in Kontext-Menüs zu integrieren, um die visuelle Darstellung nicht unnötig zu belasten.

Ein wichtiger Punkt beim Entwurf einer visuellen Sprache ist es, dem Anwender möglichst freie Hand bei der Arbeit zu lassen und den Arbeitsfluss nicht allzu restriktiv zu handhaben. Programmieren ist kein linearer Vorgang, sondern oft ein Springen im Code. Dabei kommt es häufig vor, dass Details und auch die Grobstruktur der Software ineinander verschlungen entwickelt werden. Die visuelle Sprache sollte hier also keine Vorschriften bei der Abarbeitung machen. Ein dezenter Hinweis auf den Arbeitsfluss ist

jedoch wünschenswert, gerade um Anfänger zu unterstützen.

Die Repräsentation von Daten ist eine wichtige Eigenschaft. Erst durch eine geeignete Darstellung werden Daten zu Informationen. Dazu kann Farbe eingesetzt werden, jedoch immer nur hinweisend, entweder um auf Fehler hinzuweisen oder um bestimmte Strukturen zu unterstützen. Eine “Wasserpipeline“ könnte so zum Beispiel auf einen Datenfluss hinweisen. Allerdings kann nicht alles gleichzeitig hervorgehoben werden, im Gegenteil: falls etwas in den Vordergrund rückt, so rückt eine andere Information in den Hintergrund ([4]). Wichtige Informationen sollten durchaus mehrmals kodiert werden, wie es Fitter et al in [3] durch den Begriff “Redundant Recoding“ beschreiben.

Die zu entwerfende Sprache sollte einen Geschwindigkeitsvorteil gegenüber der herkömmlichen Entwicklung bieten und den Anwender dabei unterstützen, den Umgang mit dem DEViL-Designer zu erlernen, um auch komplexere Aufgaben zu bewältigen.

Leider gibt es nicht die eine perfekte Notation, auch kann nicht grundsätzlich gesagt werden, dass eine Sprache gut oder schlecht ist. Sie ist entweder eher geeignet oder eben nicht. Somit kann auch Diagrammen nicht immer der Vorzug gegeben werden. Die Sprache muss den Ansprüchen des Anwenders genügen, leicht erlernbar, fehlertolerant, übersichtlich und flexibel hinsichtlich des Arbeitsflusses sein ([4]).

4.2 Anforderungen

Um die konkreten Anforderungen der hier zu entwickelnden Sprache zu definieren, ist es zunächst nötig zu wissen, wer später einmal mit dem DEViL-Designer arbeiten wird und was mit der Umgebung gemacht werden soll. Die Umgebung soll von Anfängern möglichst intuitiv eingesetzt werden können. Sollte sich herausstellen, dass die entworfene Sprache ausdrucksstark ist, so werden auch Fortgeschrittene die Umgebung weiterhin einsetzen.

Der Funktionsumfang wird nicht vollständig das bereitstellen, was DEViL alles an Funktionen zur Verfügung stellt. Der DEViL-Designer soll das Erstellen des Modells sowie die Spezifikation von textuellen Repäsentationen und der Sichten erlauben. Zusätzlich wird die visuelle Spezifikation der Codegenerierung ermöglicht. Nicht unterstützt wird die Möglichkeit separate Sichtmodelle zu definieren.

Neben den bereits erwähnten Anforderungen an die zu entwickelnde Umgebung sollten unvollständig angewendete Muster farblich gekennzeichnet werden und wenn möglich automatisch ergänzt werden. Wie in Kapitel 3 erwähnt, bestehen Muster immer aus mindestens zwei Rollen. Wenn der Benutzer also auf eine Liste von Knoten das Mengenmuster anwendet, ist es wünschenswert, dass die Mengenelementrolle an die Knoten der Liste automatisch ergänzt wird. Dies ist jedoch nicht immer gewünscht. Beim Tabellenmuster zum Beispiel gibt es die Rolle *VPTableCell*, die die Tabellenzellen darstellt. Die Rolle kann auf Attributknoten angewendet werden, das heißt, sowohl auf

Blattknoten als auch auf SUB Knoten. Der DEViL-Designer kann hier jedoch nicht automatisch entscheiden, an welchen Elementen die *VPTableCell* Rolle angewendet werden soll, da in der Regel nur eine Teilmenge der Attributknoten gewünscht ist. Ein Auswahldialog mit allen zur Verfügung stehenden Optionen ist hier also angebracht.

Da DEViL noch weiterentwickelt wird, werden in Zukunft wohl noch weitere visuelle Muster hinzukommen, wie zum Beispiel ein Muster, das eine baumartige Strukturansicht darstellt. Deshalb ist es nötig, dass der DEViL-Designer auch durch zukünftig entstehende Muster erweitert werden kann und eine geeignete Schnittstelle zur Definition von Mustern anbietet. Eine spezielle Sicht zur Spezifikation von Mustern ist also nötig. Muster, die durch Kontrollattribute stark vom Benutzer verändert werden können, werden fest implementiert, da so Vorschausichten implementiert werden können, die die Auswirkungen von gesetzten Kontrollattributen direkt sichtbar machen. Beim Formularmuster soll der Editor für generische Zeichnungen (GDED [13]) eingebunden werden. Beim Listenmuster soll es so zum Beispiel eine Sicht geben, in der die Eigenschaften der Liste editiert und die Auswirkungen direkt betrachtet werden können.

Es soll verhindert werden, dass der Anwender Inkonsistenzen entwirft. Inkonsistenzen können unter anderem entstehen, wenn der Anwender vergisst, bestimmte Muster oder Rollen anzuwenden. Es fehlen also bestimmte benötigte Interfaces, um die graphische Darstellung zu berechnen. Auch darf der Anwender bestimmte Rollen nicht an jeden Knoten anwenden: Die *VPSimpleList* Rolle darf nur auf einen SUB Knoten, also einem Knoten, der eine Folge von Elementen enthält, angewendet werden. Die Anwendung auf einen Objekt- oder Blattknoten muss in diesem Fall verhindert werden. Des Weiteren gibt es Konsistenzeigenschaften für die Spezifikation der abstrakten Struktur. Hier muss verhindert werden, dass von konkreten Klassen geerbt wird und dass bestimmte Bedingungen für Namen von Klassen oder Attributen verletzt werden, wie beispielsweise die Verwendung von Leer- oder Sonderzeichen.

Ein weiterer wichtiger Punkt einer graphischen Benutzeroberfläche ist es, den Arbeitsfluss des Anwenders nicht in eine bestimmte Richtung zu drängen. Der Arbeitsfluss des Anwenders sollte möglichst flexibel gestaltet werden. In der hier zu entwickelnden Umgebung sind dabei mehrere Varianten möglich. Der Anwender könnte so zunächst das Modell entwerfen und darauf aufbauend in einer zweiten Sicht die Muster anwenden. Diese zweite Sicht wird aufbauend auf der abstrakten Struktur automatisch generiert und auf die speziellen Bedürfnisse der visuellen Muster zugeschnitten. Eine zweite Variante wäre eine integrierte Sicht. Der Anwender erstellt Objekte, auf die er dann direkt visuelle Muster anwendet. Danach werden sowohl abstrakte Struktur als auch die Sichten automatisch generiert. Auch eine Kombination beider Varianten ist denkbar. In dieser Arbeit wird dabei zunächst die erste Variante verfolgt.

Spezielle Assistenten sollen integriert werden, die dem Benutzer beim Spezifikationsprozess helfen und ihm DEViL Konzepte auf visuellem Wege verdeutlichen.

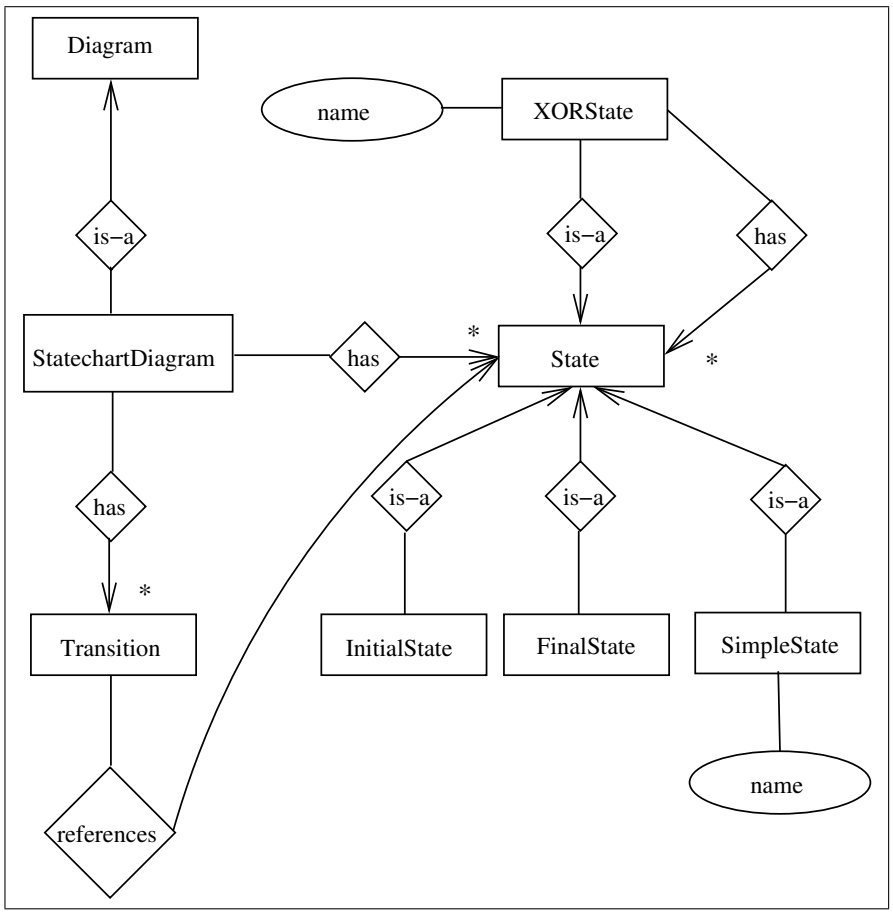

Abbildung 24: ER Diagramm

Die entworfene Struktur sollte zudem testbar sein, schon bevor die Modellierung vollständig fertig ist.

4.3 Sprachentwurf für das Modell

Im folgenden Abschnitt soll eine geeignete graphische Repräsentation für den Entwurf des Modells gefunden werden. Dazu werden zunächst diverse Repräsentationen des Modells anhand eines laufenden Beispiels untersucht.

4.3.1 ER-Diagramme

Eine Variante um das Modell zu entwerfen sind die Entity-Relationship Diagramme [11]. Dazu ist das Statechart Beispiel aus Kapitel 3 als Entity-Relationship Diagramm in Abbildung 24 spezifiziert. Entity-Relationship Diagramme sind nicht besonders gut geeignet, um das Modell zu spezifizieren. Rollen können nicht explizit ausgedrückt werden, d. h. in der Notation

```
subStates: SUB State*;
```

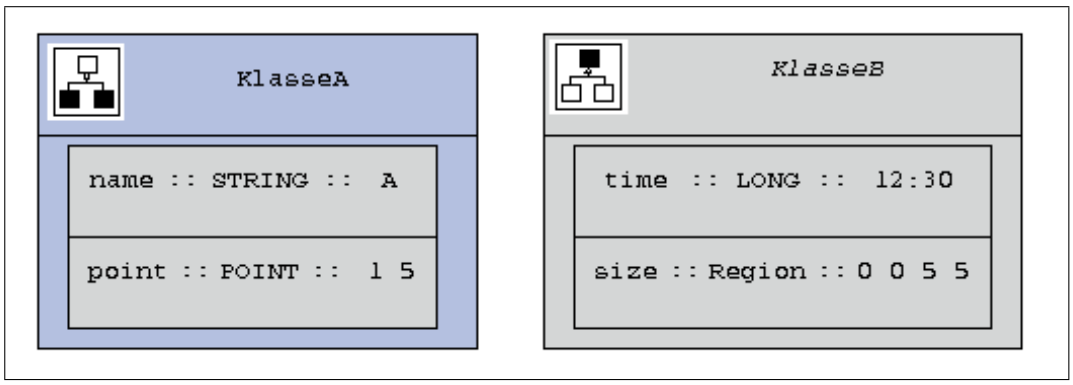

Abbildung 25: Darstellung einer Klasse (links eine konkrete, rechts eine abstrakte Klasse)

kann der Name der Aggregation *subStates* nicht ausgedrückt werden. Ebenfalls gibt es keine Notationen für abstrakte Klassen. Die Unterscheidung von Referenzen und Aggregationen kann nicht graphisch ausgedrückt werden, sondern nur textuell: “has“ bzw. “references“ Relationen. Hier wäre es also nötig, zusätzliche Notationen zu finden.

4.3.2 UML Klassendiagramme

UML Klassendiagramme [9] bieten die Möglichkeit, die Struktur des zu entwerfenden oder abzubildenden Systems darzustellen. Es können statische Eigenschaften und Beziehungen von Objekten untereinander abgebildet werden. In der UML Spezifikation enthalten Klassendiagramme folgende Elemente:

- Klassen
- Schnittstellen
- Attribute
- Operationen
- Assoziationen (Kompositionen und Aggregationen)
- Generalisierungsbeziehungen
- Abhängigkeitsbeziehungen

Für die Sprache, in der die abstrakte Struktur modelliert wird, werden Operationen und Schnittstellen nicht benötigt, da sie im Modell nicht existieren. Alle anderen Elemente existieren auch im Modell. UML Klassendiagramme sind somit eine gute Wahl zur Spezifikation einer Sicht für das Modell. Eine Klasse wird, wie in Abbildung 25 zu sehen dargestellt. Eine Klassendefinition besteht aus einem Rechteck mit zwei Kästen. Im oberen Kasten ist neben dem Klassennamen ein Icon zu sehen, das variiert, je nachdem ob es sich um eine abstrakte Klasse handelt oder nicht. Das hierfür eigentlich von UML benutzte kursiv Stereotyp des Klassennamens erschien zu ausdrucksschwach. Im

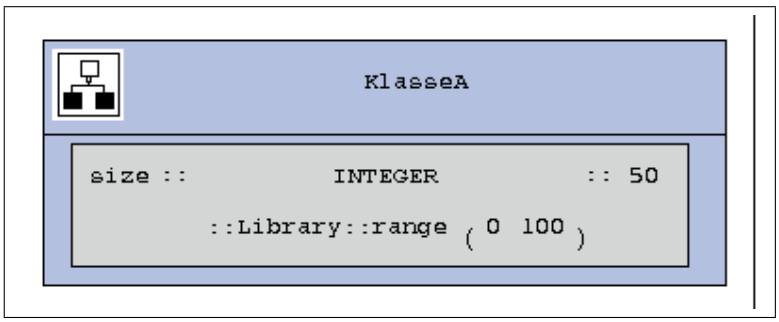

Abbildung 26: Konsistenzcheck für Attribute: *size* muss zwischen 0 und 100 groß sein

unteren Kasten sind die Attribute mit Namen und Typ aufgelistet. Zusätzlich besteht die Möglichkeit, Attributen Vorgabewerte zuzuweisen. Dies ist in Abbildung 25 beim Attribut *name* der KlasseA zu sehen. Der Vorgabewert ist der String "A". Außerdem können Attributen von Klassen Konsistenzbedingungen übergeben werden. In Abbildung 26 ist dies für das Attribut *size* zu sehen, dessen Wert zwischen 0 und 100 liegen muss. Die Funktionen für die Konsistenzbedingungen sind in der Standard-Bibliothek enthalten.

Einen Kasten für Operationen, wie er in der UML Spezifikation vorhanden ist, gibt es nicht. In einer Spezifikation für eine abstrakte Struktur gibt es keine Möglichkeit Operationen zu definieren. Neben der Definition von Klassen ist es nun noch nötig, die Beziehungen der Klassen untereinander zu definieren. Hier ist zunächst die Referenz zu nennen. Folgender Ausschnitt aus der abstrakten Struktur

```
CLASS KlasseA {
 myRef: REF KlasseB;
}
```

wird graphisch wie in Abbildung 27 durch einen Pfeil verdeutlicht. Vererbungsbeziehungen werden genauso wie in UML dargestellt, wie in Abbildung 28 zu sehen ist. Es kann nur von abstrakten Klassen geerbt werden. Mehrfachvererbung ist möglich. Eine Besonderheit und Abweichung vom UML Standard gibt es bei der Modellierung von Klassen, die andere Klassen als Unterelemente enthalten:

```
CLASS KlasseA {
 bKlassen: SUB KlasseB*;

 name: VAL VLString;
 position: VAL VLPoint;
}
```

Hier wird die Beziehung *bKlassen* nicht durch ein Attribut in der Klasse *KlasseA* modelliert. In Java etwa würde in der Klasse ein Attribut *Vector bKlassen* hinzufügt werden, das dann mit Klassen vom Typ KlasseB gefüllt würde. Im DEViL-Designer soll so eine Beziehung jedoch explizit dargestellt werden, um die Untermenge auszudrücken. Dazu wird

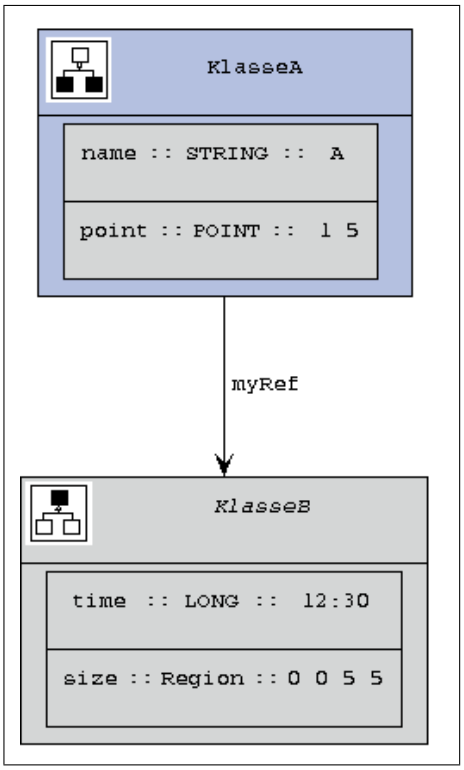

Abbildung 27: Referenz auf eine Klasse

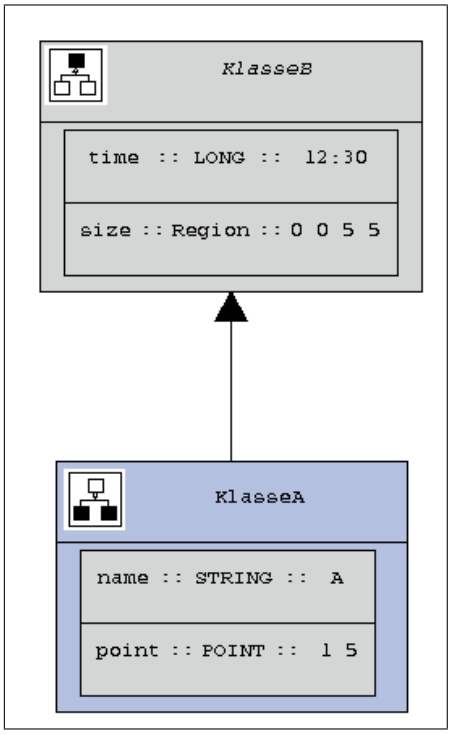

Abbildung 28: Vererbung

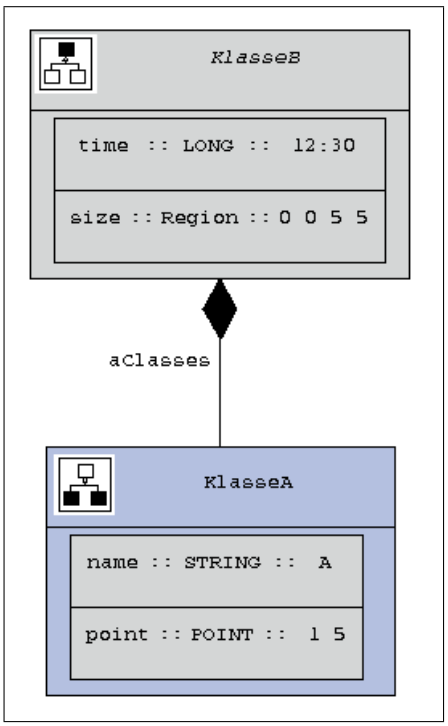

Abbildung 29: Aggregation

die UML Komposition verwendet. Die textuelle Darstellung oben sieht dann wie in Abbildung 29 aus. Die Komposition ist wie in UML auch hier eine Teil-Ganzes Beziehung. Die Aggregation besteht nur, wenn auch die Klasse *KlasseA* besteht.

Darstellung des erzeugten Codes DEViL bietet die Möglichkeit, die abstrakte Struktur mit einer speziellen Dokumentarsyntax zu versehen, aus der dann wiederum Dokumentationen beispielsweise in HTML generiert werden können. Der Code sieht wie folgt aus und wurde ebenfalls in die die Sicht für die Spezifikation des Modells integriert:

```
DOC StatechartDiagram  {
  [[ Modelliert StatechartDiagram Knoten ]]
  name:        [[ VAL Knoten, Name des Diagrams ]]
  states:      [[ SUB Knoten mit States  ]]
  connections: [[ Liste mit Verbindungen ]]
}
```

Über das Kontextmenü eines jeden Strukturobjekts kann ein Kommentar hinzugefügt werden.

Des Weiteren sollte der später erzeugte Code auch vom Benutzer lesbar sein, damit spätere Änderungen oder der Einbau von zusätzlicher Funktionalität von Hand ermöglicht wird. Dazu wurde der erzeugte Code mit Kommentaren und übersichtlichen Einrückungen ausgestattet.

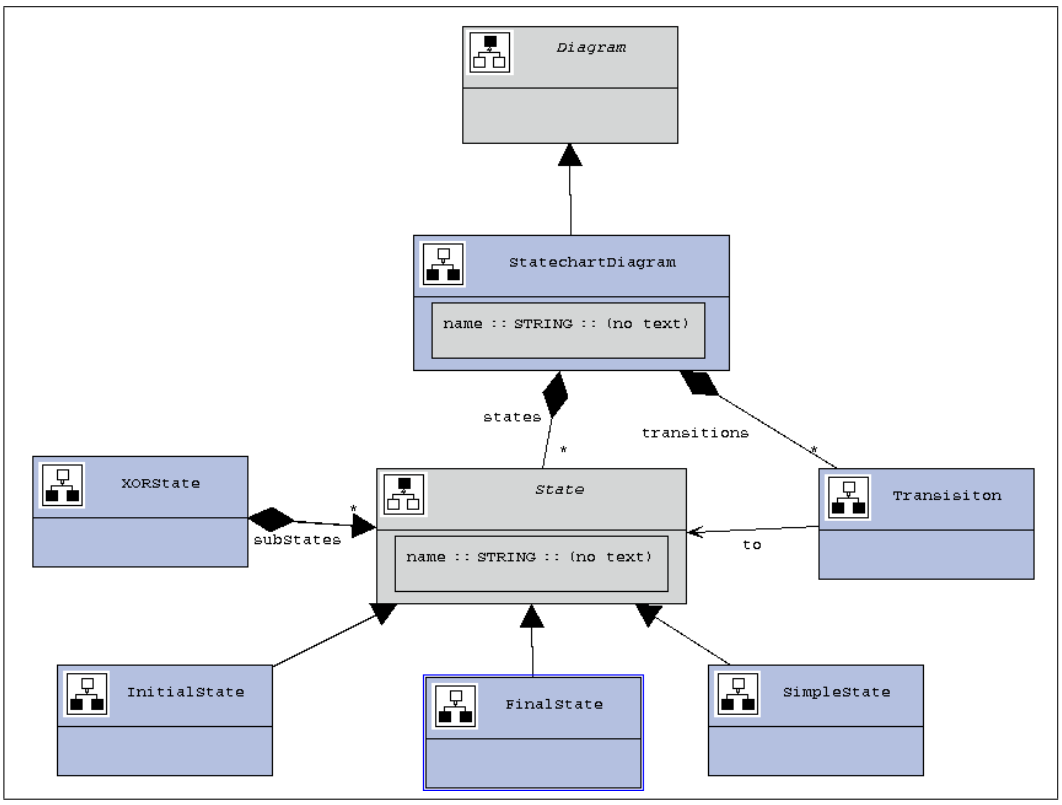

Abbildung 30: Statechart Beispiel

Zum Schluss soll nun das Statechart Diagramm als abstrakte Struktur in UML Klassendiagrammnotation dargestellt werden, wie in Abbildung 30 zu sehen ist.

4.4 Sprachentwurf für visuelle Muster

In Kapitel 3 wurde erläutert, wozu visuelle Muster verwendet, wie sie angewendet werden und wie die innere Struktur der Muster spezifiziert ist. In diesem Kapitel soll nun eine Sicht für das Anwenden der Muster entwickelt werden. Bevor diese Sicht entworfen wird, soll jedoch erst eine Umgebung zur generischen Spezifikation von Mustern und deren Rollen entwickelt werden, damit auch später entstehende Muster noch dem DEViL-Designer hinzugefügt werden können. Zum Schluss soll auf die einzelnen Sichten der Muster eingegangen werden, die es erlauben, die Auswirkungen der Modifikation von Kontrollattributen direkt zu sehen.

4.4.1 Spezifikation von Mustern

Visuelle Muster sollen in dem hier zu entwerfenden DEViL-Designer möglichst automatisch vervollständigt werden können. Fehler beim Entwurf durch den Benutzer sollen erkannt werden. Auch soll es möglich sein, später weitere Muster hinzuzufügen,

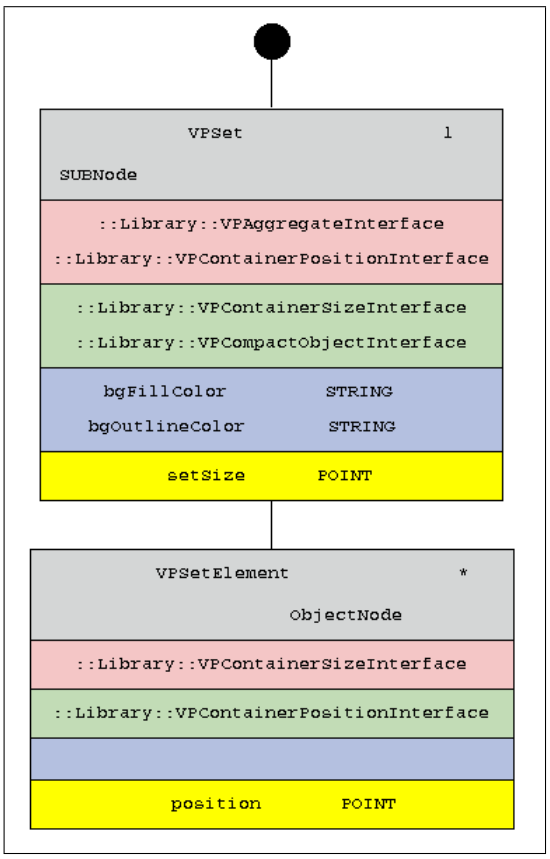

Abbildung 31: Rollendiagramm zur Spezifikation visueller Muster (hier: Mengenmuster *VPSet*) mit den zusätzlichen Kontrollattributen *bgFillColor* und *bgOutlineColor* und weiteren internen Kontrollattributen (gelb hinterlegt)

die sich jetzt noch in der Entwicklungsphase befinden. Dazu ist es nötig, die in Kapitel 3 beschriebene Struktur der visuellen Muster in der Umgebung beschreiben zu können.

Die Sicht, in der die visuellen Muster für den zu entwickelnden DEViL-Designer spezifiziert werden, hält sich sehr eng an die zuvor eingeführten *Rollendiagramme*. Die Rollen mit den benötigten und bereitgestellten Interfaces bilden eine Baumstruktur, die das interne Konzept der Muster abbildet. Zu den Interfaces, die Rollen bereitstellen bzw. benötigen, kommen noch Definitionen für Kontrollattribute hinzu, wie in Abbildung 31 zu sehen ist (blau hinterlegter Teil). Kontrollattribute bestehen aus einem Typ und einem Wert. Wird eine Rolle später in der Sicht zur Spezifikation von graphischen Strukturediteren angewendet, so werden die Kontrollattribute mit Hilfe des Kopplungsalgorithmus automatisch hinzugefügt. Die Kontrollattribute werden dann schließlich bei der Codegenerierung zur Modifikation der Rollen benutzt.

Einige Rollen visueller Muster benötigen zusätzliche Attribute in der abstrakten Struktur. So benötigt die *VPSetElement* Rolle das Attribut

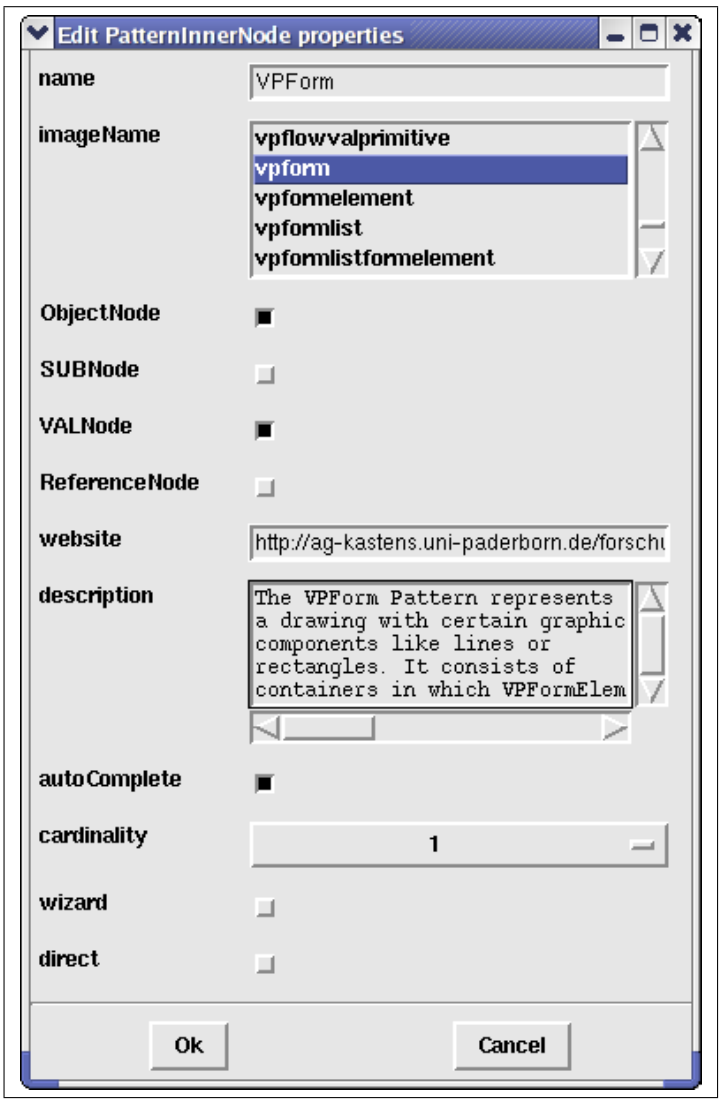

Abbildung 32: Eigenschaften der Spezifikation eines Musters

```
position: VAL VLPoint;
```

in der abstrakten Struktur, um einem Mengenelement eine Position in der Ebene zuzuordnen. Diese zusätzlich benötigten Attribute werden automatisch der abstrakten Struktur hinzugefügt. Der Anwender muss dies nicht explizit in der Sicht zur Spezifikation des Modells machen. In der Musterspezifikationssicht dient der gelb hinterlegte Teil der Spezifikation dieser internen Kontrollattribute.

Neben den Kontrollattributen kommen noch einige weitere Attribute hinzu, wie am Kontextmenü einer Rolle in Abbildung 32 zu sehen ist. Diese Attribute sollen im folgenden Abschnitt genauer erläutert werden.

In der gerade beschriebenen Sicht für die Definition von visuellen Mustern wird

die Struktur der Muster abgebildet. Neue Muster können einfach integriert werden. Der Zugriff auf die Musterstruktur erfolgt generisch und wird durch Setzen einer Referenz auf ein Muster in der Bibliothek realisiert. Dies ist umständlich und für die wichtigsten Muster wäre es nützlich, eigene Spezifikationen zu haben, damit ein Einfügen in die Sichten vereinfacht und die Modifikation von Kontrollattributen durch Vorschausichten direkt sichtbar wird. Hierzu wurden die viel verwendeten Muster *VPForm*, *VPSimpleList*, *VPSet*, *VPConnection*, *VPTable*, *VPFlowForm*, *VPFlowList* und *VPValTextPrimitive* als eigene Strukturobjekte realisiert. Die Struktur dieser Muster greift jedoch weiterhin auf die oben beschriebene Spezifikation zu, die persistent in einer Bibliothek gespeichert wird. Auf die speziellen Sichten der fest integrierten Muster wird in Abschnitt 4.4.2 eingegangen.

Konsistenz für Rollen Der Benutzer soll unterstützt werden, konsistente Sichtspezifikationen zu entwerfen. Dazu ist es nötig zu spezifizieren, an welchen Knotentypen welche Rollen angewendet werden dürfen. Über die Attribute *ObjectNode*, *SUBNode*, *VALNode* und *ReferenceNode* kann angegeben werden, an welchen Knotentyp die Rolle vererbt werden darf. In DEViL existieren drei verschiedene Arten von Knoten der abstrakten Struktur:

- Objektknoten: repräsentieren Instanzen von Klassen
- Listen Knoten (SUB): repräsentieren eine Folge von Objektknoten
- Blattknoten: repräsentieren Attribut Knoten (sowohl Attribute von Objekten als auch Referenzen (VAL oder REF))

Die *VPSimpleList* Rolle darf so zum Beispiel nur an einen SUB Knoten vererbt werden, also an eine Menge von Objekten. Für die Rollen gibt es neben den oben aufgezählten Knotenvarianten noch zwei weitere Kombinationen. Bestimmte Rollen dürfen sowohl an SUB Knoten als auch an Blattknoten vererbt werden. Die *VPFormElement* Rolle ist so ein Fall. Einige Endstücke wie die *VPIdTextPrimitive* Rolle können sogar an jeden beliebigen Knotentyp angewendet werden. *VPIdTextPrimitive* stellt den Knotennamen textuell dar. Da jeder Knotentyp einen Namen besitzt, kann *VPIdTextPrimitive* an jeden Knotentyp angewendet werden. Durch die vier Kontrollattribute namens “ObjectNode, SUBNode, VALNode“ sowie “ReferenceNode“ können alle möglichen Kombinationen von Rollenanwendungen an Knoten spezifiziert werden.

Das letzte Attribut *direct* dient dazu, zu spezifizieren, ob die Unterrolle an einem Knoten direkt unterhalb ihrer übergeordneten Rolle angewendet werden muss. Ist es aktiviert, muss sich die Rolle direkt in einem Knoten unterhalb ihrer übergeordneten Rolle befinden. Ist es nicht aktiviert, dürfen sich beliebig viele andere Knoten zwischen Rolle und übergeordneter Rolle befinden.

Automatische Vervollständigung Eine Anforderung an die Umgebung war es, dass der Benutzer auf dem Weg zu einer Sichtspezifikation möglichst unterstützt werden

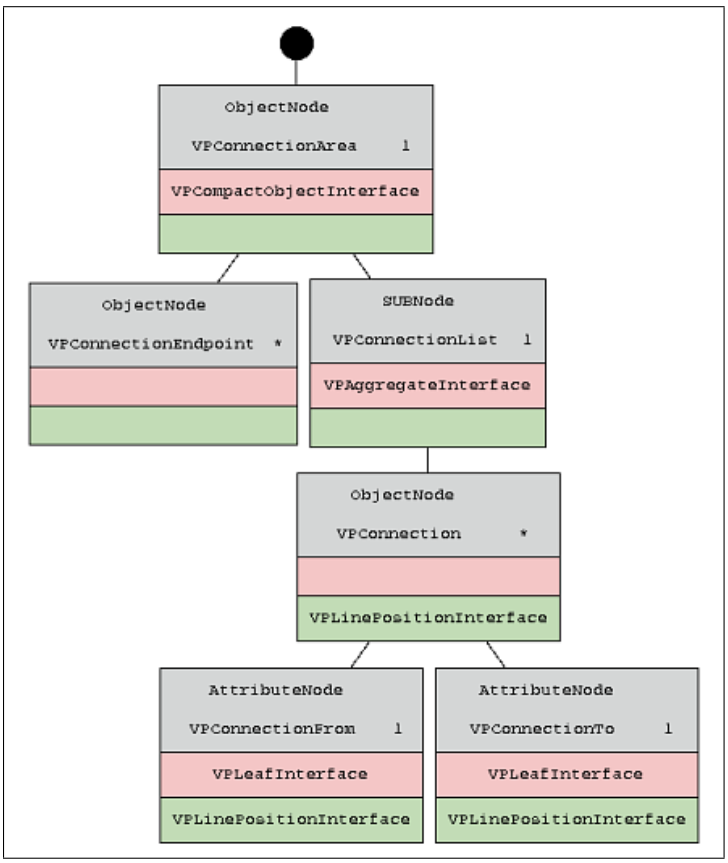

Abbildung 33: Das *VPConnection* Muster im Spezifikationseditor

soll und überflüssige Aktionen vermieden werden. Dazu sollen Unterrollen von Mustern möglichst automatisch an den entsprechenden Knoten angewendet werden. Das Attribut *autoComplete* in der Spezifikation der Muster gibt dazu an, ob diese Rolle automatisch an Unterknoten vererbt werden soll. Dies ist zum Beispiel beim *VPForm* Muster interessant. *VPFormElement* Rollen werden dann automatisch an allen Unterknoten erzeugt.

Die Kardinalität gibt an, ob genau ein Unterknoten dieser Rolle existieren muss oder ob beliebig viele Unterknoten existieren dürfen.

Das Attribut *wizard* gibt an, ob der “Vervollständigungsassistent“ eingesetzt werden soll. Dieser Assistent fragt den Benutzer dabei in einem modalen Dialog, an welche Unterknoten die entsprechende Rolle vererbt werden soll. Bei der *VPTableCell* Rolle kann der DEViL-Designer, wie bereits erwähnt, nicht automatisch entscheiden, welche Unterknoten eine Tabellenzelle darstellen sollen, da es sein kann, dass es in der übergeordneten Klasse mehrere Attribute und Referenzen gibt. Der Assistent wird ebenfalls automatisch gestartet, falls das Attribut *autoComplete* gesetzt ist und die Kardinalität eins ist. Auch hier kann der DEViL-Designer nicht autonom entscheiden.

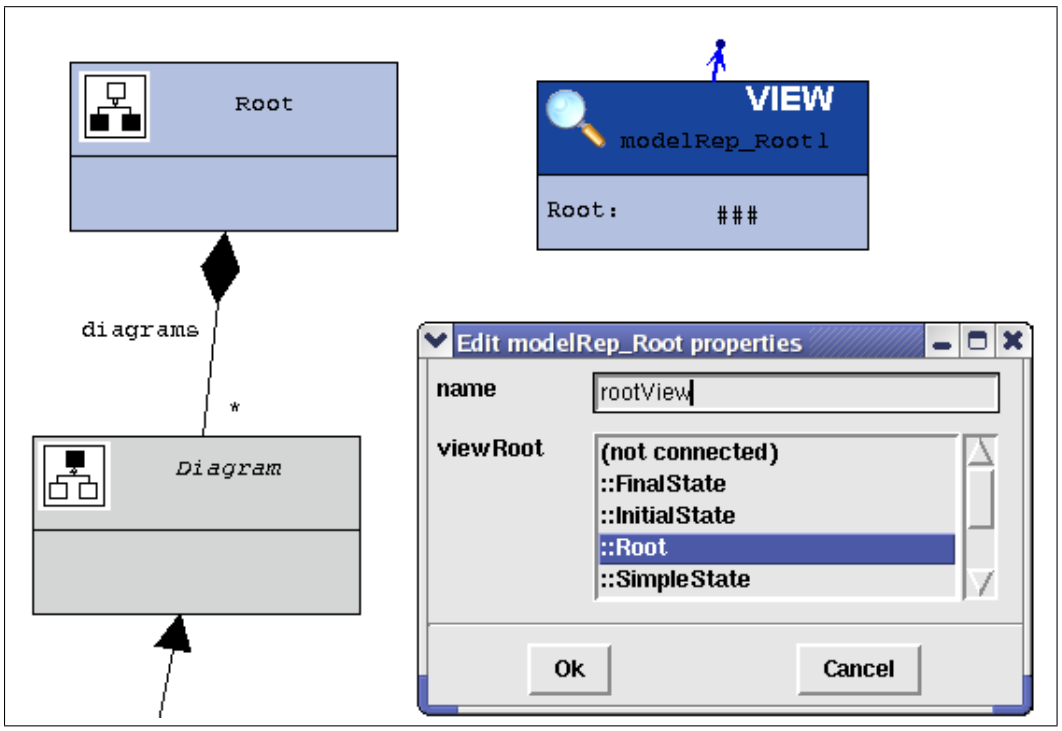

Abbildung 34: Definition einer Wurzel für die Sicht

4.4.2 Sicht für das Anwenden visueller Muster

In diesem Abschnitt soll nun die Sicht beschrieben werden, die für das Anwenden der visuellen Muster benutzt wird. Wie visuelle Muster angewendet werden, wurde in Kapitel 3 beschrieben. Ziel ist es nun, die dort beschriebene textuelle Spezifikationssprache durch eine graphische zu ersetzen.

In DEViL wird eine Sichtspezifikation begonnen, indem die Wurzel, also der Ausgangspunkt für die Spezifikation, festgelegt wird. Dazu wird in der Sicht für das Modell eine neue Sichtspezifikation eingefügt und, wie in Abbildung 34 zu sehen, eine Klasse ausgewählt, die die Wurzel repräsentiert. Durch Doppelklick auf die Sichtdefinition öffnet sich die Sicht, in der visuelle Muster auf Objekte der abstrakten Struktur angewendet werden können. Diese Sicht entspricht einer Baumstruktur und unterstützt dadurch den Arbeitsfluss des Benutzers. Außerdem können Muster, die ebenfalls in einer Baumstruktur vorliegen, leichter identifiziert werden. In der Regel ist es am einfachsten, wenn Rollen von “oben nach unten“ angewendet werden.

Eine alternative Variante für das Anwenden der visuellen Muster ist in Abbildung 35 zu sehen. Hier wurde die Sicht des Modells übernommen und zusätzlich für die visuellen Muster UML Stereotypen an die Klassen geschrieben. Da an DEViL SUB Knoten ebenfalls Rollen von Mustern vererbt werden können, wurden diese durch horizontale Linienverbindungen und einem Text dargestellt.

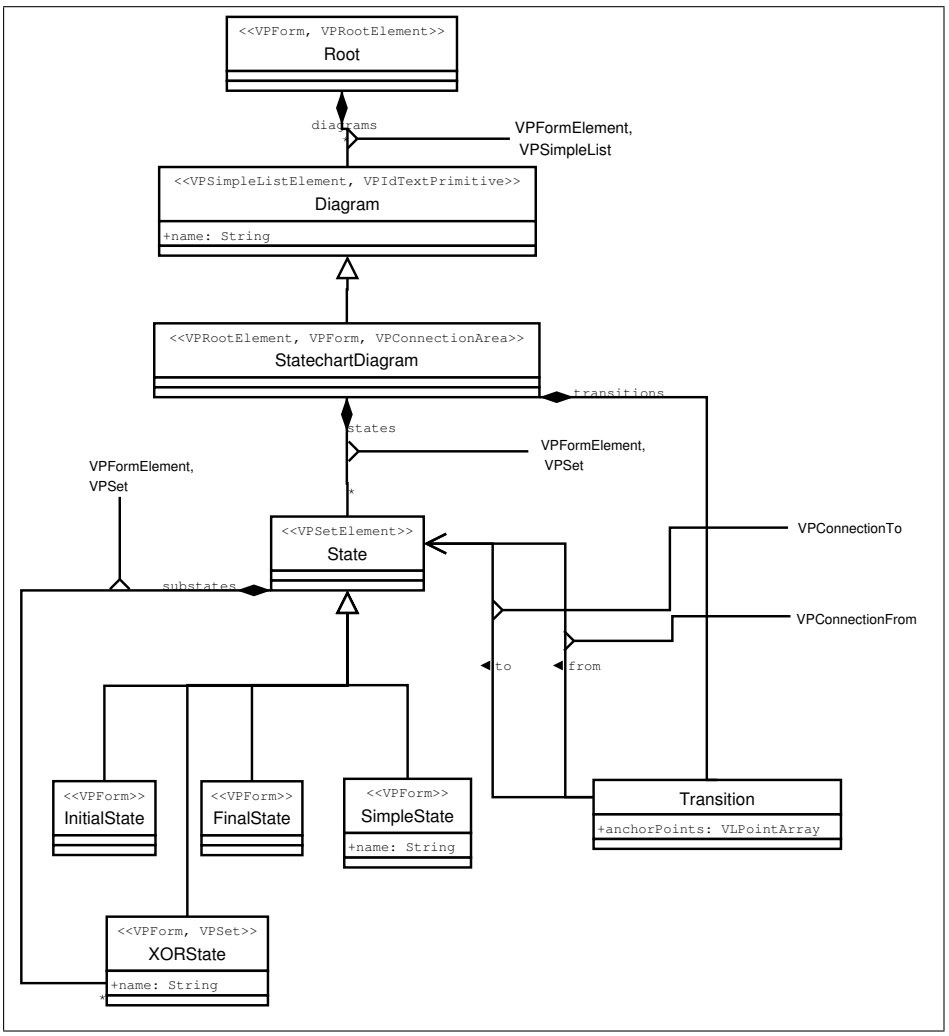

Abbildung 35: Eine zweite Variante für das Anwenden der visuellen Muster

Klassen, Knoten und Kardinalitäten Wie in der Abbildung zu sehen, ist diese Art der Darstellung sehr unübersichtlich, es gibt zu viele Linien und insbesondere können an Attribute keine Muster vererbt werden. Deshalb wurden gegenüber der Sicht für das Modell einige Modifikationen gemacht. So werden SUB Knoten als eigenständige Objekte dargestellt, wie in Abbildung 38 zu sehen. Falls es sich dabei um einen Knoten mit der Kardinalität “*“, also einer Liste von Knoten handelt, so wird die Verbindung zum SUB Attributknoten durch eine dreifach gestrichelte Linie gekennzeichnet. Ist die Kardinalität “!“, so wird eine durchgezogene Line gezeichnet. Bei der Kardinalität “%“ wird die Linie dick und bei der Kardinalitätsangabe “?“ wird eine einfach gestrichelte Linie gezeichnet. Diese Unterscheidungen erlauben es, einfach zu erkennen, welcher Knotentyp vorliegt und welches Muster geeignet ist.

Attribute und Referenzen werden, wie in Entity-Relationship Diagrammen, als abgerundete Rechtecke dargestellt (*name, bRef* in Abb. 39). Abstrakte Klassen bekommen

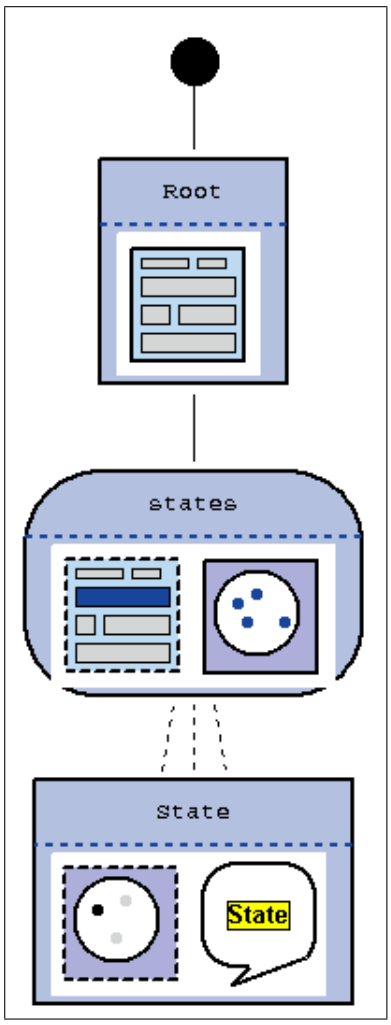

Abbildung 36: Eine Musterkombination: *VPForm* Rolle am Knoten Root, Rollen *VPFormElement*, *VPSet* am SUB Knoten states sowie die Rollen *VPSetElement* und *VPTextPrimitive* am Knoten State

ein UML Stereotyp (Klasse A).

Wie in Abbildung 39 zu sehen, sind alle Objekte (bis auf Stellvertreter Objekte, hier werden Rollen an die Ausgangsklassen angewendet) durch eine gestrichelte Linie zweigeteilt. Im oberen Teil ist der Name des Objekts zu sehen, und im unteren Teil können nun visuelle Muster eingefügt werden. Diese Zweiteilung ist von Vorteil, da Platz gespart wird und Muster unmittelbar mit Knoten verknüpft werden können. Die Zugehörigkeit zwischen Mustern und Knoten ist direkt sichtbar. In Abbildung 36 wurde das *VPForm* Muster angewendet. Das Piktogramm *VPForm* wurde an die Root Klasse vererbt, *VPFormElement* an den SUB Knoten states. Die wichtigsten Mustervarianten sind alle durch eigene Piktogramme dargestellt.

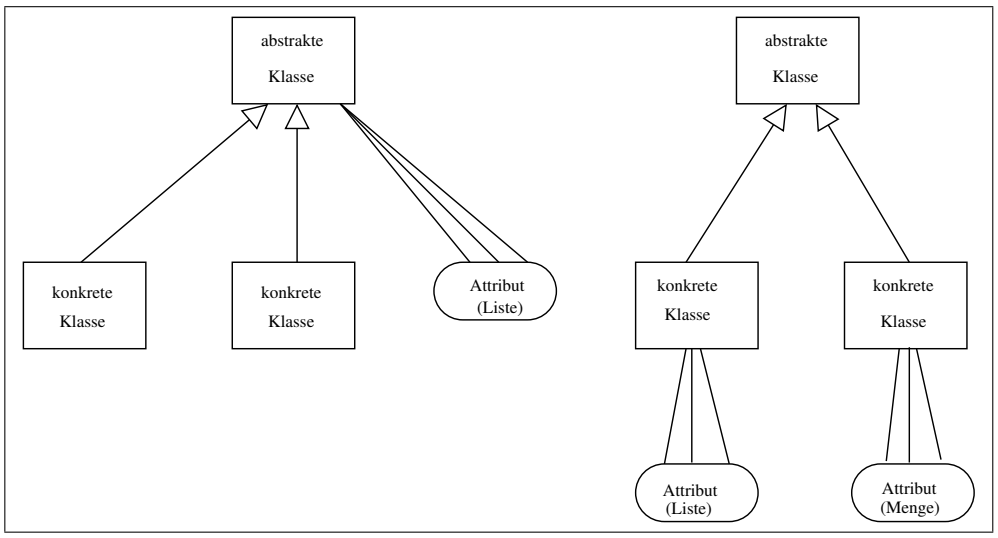

Abbildung 37: Zuordnung von Attributen: Das Attribut links wird an die Unterklassen zugewiesen, um unterschiedliche Muster anwenden zu können.

Zuordnung von Attributen In DEViL können abstrakten Klassen Attribute, also Referenzen, SUB Knoten und Werte zugeordnet werden. Diese Attribute sollten, falls die abstrakte Klasse Unterklassen besitzt, auch an die Unterklassen angewendet werden. Dies hat den Vorteil, dass Attribute in den verschiedenen Unterklassen mit unterschiedlichen Mustern dargestellt werden können, weil dann in der Sicht zur Musteranwendung an jeder konkreten Unterklasse ein Attributknoten gehängt wird.

Vorstellbar wäre es also, dass ein SUB Knoten einer abstrakten Klasse in einer konkreten Unterklasse als Menge und in einer zweiten Unterklasse als einfache Liste dargestellt wird, wie in Abbildung 37 rechts zu sehen ist. Zu beachten ist hierbei, dass unter einer abstrakten Klasse noch beliebig viele weitere abstrakte Klassen hängen können. Die Zuweisung der Attribute der abstrakten Oberklasse erfolgt dann an die erste konkrete Unterklasse. Des Weiteren ist es wünschenswert, dass Attribute auch ergänzt werden können. Ein Einfügen eines an eine Unterklasse zugewiesenen Attributes an eine Oberklasse soll weiterhin möglich sein. Die Rollen des Attributes der Oberklasse werden dann zu denen des Attributs der Unterklasse hinzugefügt.

Stellvertreterklassen Da in der Sicht zur Zuordnung von visuellen Mustern eine Baumstruktur, also ein gerichteter azyklischen Graph verwendet wird, die Sicht der abstrakten Struktur jedoch einen ungerichteten ggf. auch zyklischen Graphen darstellt, war es nötig, eine Umwandlung in diese Baumstruktur zu entwerfen. Dabei ist der Spezialfall zu betrachten, dass Klassen in der abstrakten Struktur auch in einem Zyklus enthalten sein können und eine Klasse auch in mehreren anderen Klassen aggregiert enthalten sein kann. Um diese Klassen im Baum darstellen zu können, war es nötig, sie durch Stellvertreter zu repräsentieren. Rollen können damit nur zentral an eine Klasse gebunden werden und nicht an alle möglichen Vorkommen im Baum. Dies dient der Übersichtlichkeit. Des

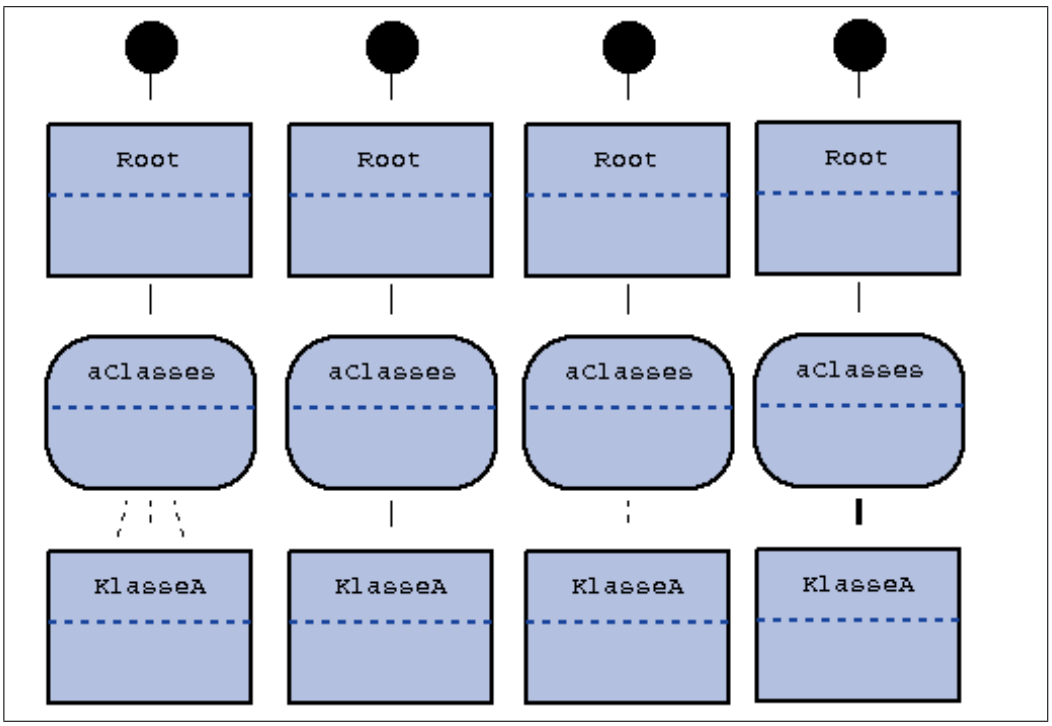

Abbildung 38: Ein SUB Knoten jeweils mit den Kardinalitäten *, !, ? und %

Weiteren wird dadurch eine Endlos-Rekursion bei der Kopplung vermieden, dazu in Kapitel 5.1 mehr. Die Klassen werden also einmal graphisch als normale Klasse dargestellt und sonst nur noch durch Stellvertreterobjekte, die durch einen UML Stereotyp Kreis gekennzeichnet sind (Klasse A in Abbildung 39). Damit Zyklen und wiederholt im Baum auftretende Klassen gefunden werden können, muss eine Suche im Baum gestartet werden, um herauszufinden, welche Kanten dargestellt werden. Dieser Mechanismus und die Kopplung mit dem Modell sind in Kapitel 5.1 erläutert.

Zwischenknoten Des Weiteren ist es möglich, Zwischenknoten einzufügen, die die Grammatikabbildung ändern. Zwischenknoten können, wie in Kapitel 3 erklärt, mehrere Knoten des Strukturbaums zusammenfassen und existieren nur in einer Sicht. Sie haben kein Pendant in der abstrakten Struktur, können aber genauso behandelt werden wie Objektknoten, d. h. es können Rollen angewendet werden. Nützlich sind diese Zwischenknoten zum Beispiel bei der Anwendung des Baummusters. Hier kann eine Wurzel als Zwischenknoten hinzugefügt werden, an die dann das Formularmuster vererbt werden kann.

Ausblenden von Teilen der Sicht der visuellen Muster Oft kommt es bei der Spezifikation von Strukturediteren vor, dass es eine Sicht gibt, in der nur eine Auflistung von Strukturobjekten existiert. Bei Doppelklick auf eines dieser Elemente erscheint eine Sicht, die das Strukturobjekt konkretisiert. Dies ist auch der Fall bei dem Strukturediteor für Statechart Diagramme. Hier gibt es eine Sicht, in der alle Statecharts nur nament-

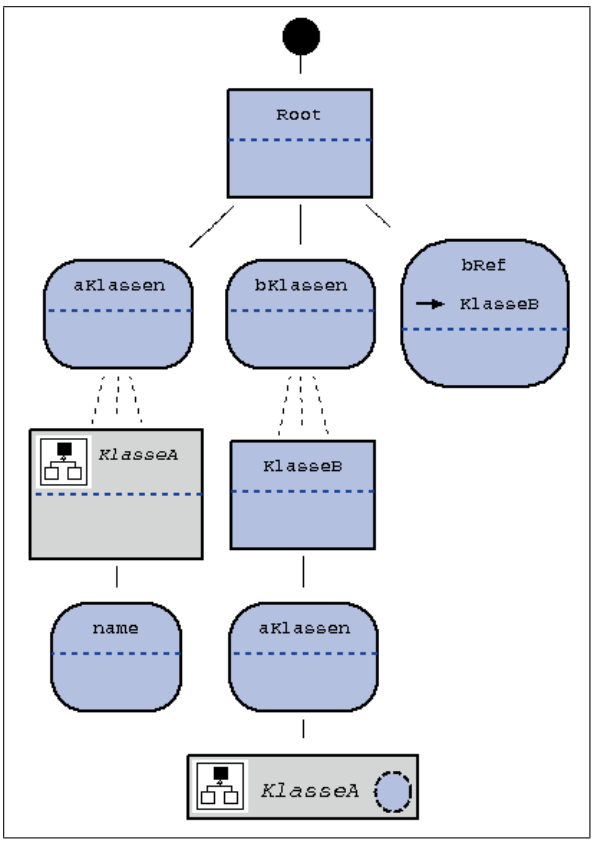

Abbildung 39: Sicht für das Anwenden der visuellen Muster

lich aufgeführt sind. Bei Doppelklick auf einen Namen erscheint die konkrete graphische Darstellung des Statecharts. Bei der Spezifikation des Statecharteditors gibt es also eine Sicht, bei der der Baum, der die abstrakte Struktur repräsentiert, nur relativ wenig Rollen enthält, da die Spezifikation zur Darstellung einer Liste mit Namen nur fünf Rollen benötigt. Zusätzlich gibt es eine Sicht mit vielen Rollenanwendungen über den ganzen Baum verteilt zur Spezifikation der konkreten Darstellung von Statecharts. In dieser ersten Sicht interessiert sich der Benutzer also nur für einen kleinen Teil des Baumes, der Rest kann somit ausgeblendet werden. Der Benutzer kann im Eigenschaftenmenü jedes SUB Knotens auswählen ob der Unterbaum angezeigt werden soll. Soll der Unterbaum nicht angezeigt werden, so wird er gelöscht. Dies hat nicht nur Vorteile bei der Übersicht, es ist auch schneller, da nur noch ein kleiner Teil der Sicht berechnet und dargestellt werden muss.

Automatische Vervollständigung von Rollen Wie bereits erwähnt, soll es die Möglichkeit geben, automatisch Rollen zu ergänzen. Falls die Umgebung jedoch nicht automatisch entscheiden kann, an welche Unterknoten Rollen angewendet werden, wird der in Abbildung 40 zu sehende Assistent gestartet, der den Benutzer die Alternativen zur Auswahl anzeigt. In diesem Beispiel wurde das *VPTable* Muster angewendet, der DEViL-Designer kann jedoch nicht automatisch entscheiden, an welche Knoten die Rolle

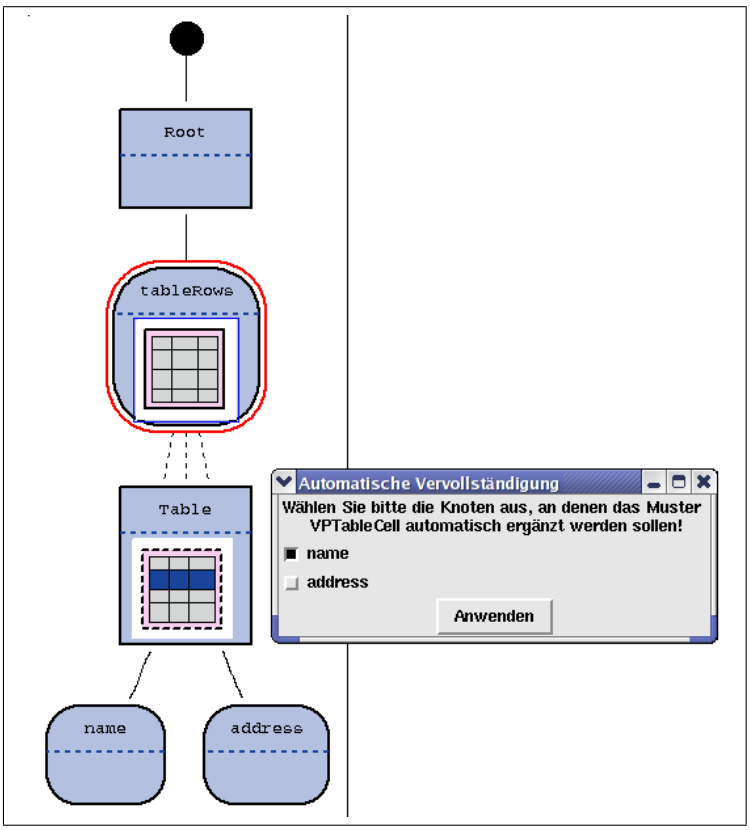

Abbildung 40: Der Assistent zur automatischen Zuweisung von Rollen

VPTableCell vererbt werden soll.

Ergänzungs-Assistent Für Anfänger im Umgang mit DEViL kann es schwierig sein, die richtigen Rollen zu finden, damit alle Interfaces vorhanden sind. Um dies zu erleichtern, unterstützt der DEViL-Designer einen "Ergänzungs-Assistenten", der für jeden Knoten, an dem Interfaces fehlen, Vorschläge unterbreitet. Dazu muss der Benutzer nur das Eigenschaftenmenü eines Knotens öffnen, und es erscheint der in Abbildung 41 dargestellte Assistent. Im oberen Teil zeigt der Assistent die bereitgestellten bzw. benötigten Interfaces an (grün bzw. rot hinterlegt). Im gelb hinterlegten Teil werden die noch fehlenden Interfaces angezeigt. Für jedes fehlende Interface macht der Assistent Vorschläge für Rollen, die das entsprechende Interface bereitstellen. Wählt der Benutzer Rollen aus, so werden sie direkt an dem Knoten angewendet. Reichen dem Benutzer diese Informationen nicht aus, so kann er sich Informationen zu den Mustern aus der Dokumentation holen. Im Kontextmenü jeder Rolle ist dazu ein Verweis auf eine Internetseite integriert.

Muster-Assistent Mit dem "Muster-Assistent" kann ein gesamtes Muster Schritt für Schritt angewendet werden. Dabei werden alle Rollen eines Musters erläutert und nacheinander an Knoten angewendet. Die Knoten, an die eine spezielle Rolle angewendet werden darf, werden grün umrandet, wie in Abbildung 42 zu sehen ist. Das Rollendia-

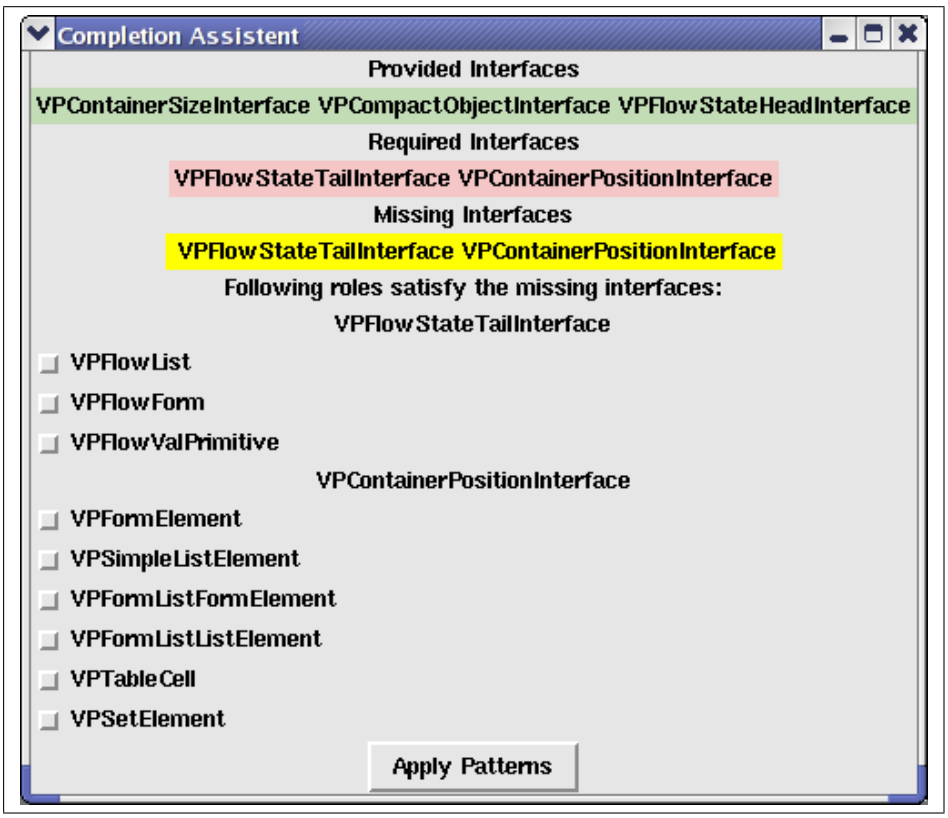

Abbildung 41: Ergänzungs-Assistent

gramm des zugehörigen Musters wird als Grafik dargestellt. Zusätzlich kann die DEViL Dokumentation für die visuellen Muster aufgerufen werden. Mit dem Muster-Assistenten sollen Anfänger auch komplexe Muster mit einer großen Anzahl an Rollen anwenden können, ohne Rollen zu übersehen, was in der Evaluation häufiger passierte (siehe Kapitel 6 auf Seite 72). Der Muster-Assistent ist im Kontextmenü der Sicht zur Anwendung visueller Muster integriert.

Vom Modell zur Baumsicht Da die Sicht zur Spezifikation visueller Muster einem Baum entspricht und zusätzliche Elemente beispielsweise für SUB Knoten eingeführt wurden, die im Modell nur als Namen existieren, war es nötig, einen Mechanismus, der in Kapitel 5.1 genauer erläutert wird, zu entwerfen, der diese zusätzlichen Strukturobjekte einfügt. Bei diesem Mechanismus, auch Kopplung genannt, werden zwei Teile eines Modells aneinander gekoppelt. Dabei wird zu einer gegebenen Ausgangsliste eine zweite Version erstellt, in der die Objekte von einem anderen Typ als in der Ausgangsliste sind. Diese Objekte haben eine Referenz auf ihr Pendant in der Ausgangsliste. Wird ein Objekt in der Ausgangsliste gelöscht, so wird auch das Replikat gelöscht.

Automatische Generierung der Grammatikabbildung Bei der Codegenerierung war zu berücksichtigen, dass einige Teile der Grammatik in der Sicht nicht visuali-

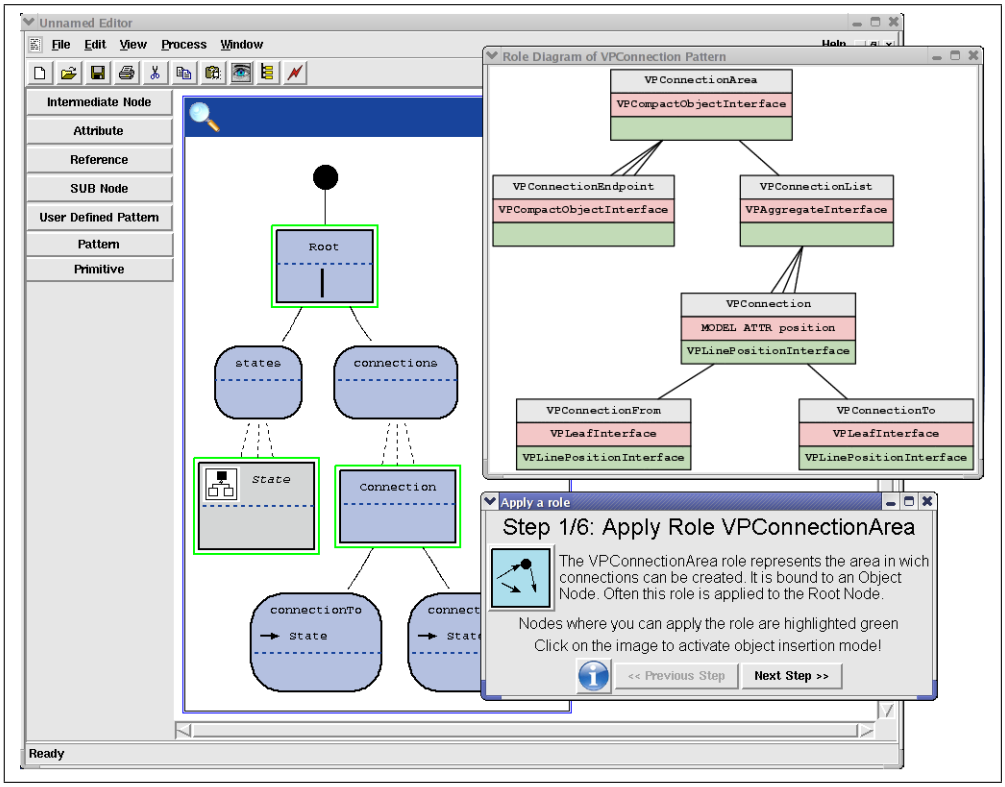

Abbildung 42: Muster-Assistent bei Auswahl des *VPConnection*-Musters. Für die Rolle gültige Knoten sind grün umrandet.

siert sind, da keine Rollen angewendet wurden. Für diese Teile gilt, dass eine Aufnahme in die Grammatikabbildung nur erfolgt, falls mindestens eine Rolle angewendet wurde. Dazu ein Beispiel:

```
XORState (subStates, name);
```

Dies bedeutet, dass der Knoten XORState zwei Attribute *substates* und *name* besitzt, die selbst in der Sicht über Rollen verfügen. Hat ein Attribut keine Rollen in der Sicht, so wird es in der Grammatikabbildung weggelassen.

4.4.3 Spezialrepräsentation der Muster

Die wichtigsten visuellen Muster sind fest in die Umgebung integriert. Dies erlaubt es, dass die Auswirkungen von Kontrollattributen in Vorschausichten betrachtet werden können. Diese Vorschausichten sollen nun genauer erläutert werden.

Formular Muster Das Formular Muster (*VPForm*) spezifiziert eine generische Zeichnung, in die beliebig viele Container eingebettet werden können, in die dann Unterelemente geschachtelt werden können. Zusätzliche graphische Elemente wie Rechtecke,

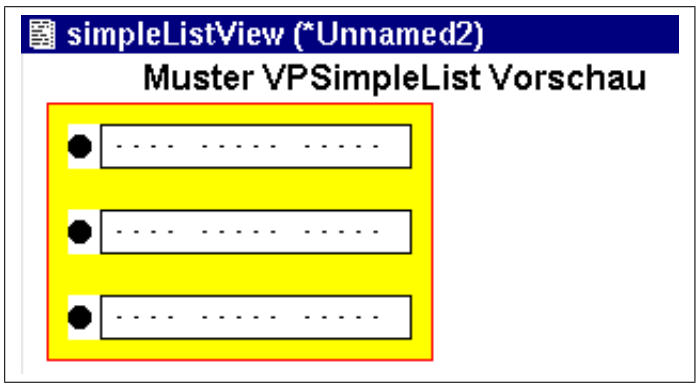

Abbildung 43: Die Vorschau für das *VPSimpleList* Muster

Kreise, Linien und Bilder können in generischen Zeichnungen ebenfalls spezifiziert werden. Zur Spezifikation dieser generischen Zeichnungen gibt es den GDED (Generic Drawings Editor) [13], der als eigenständiger Editor in DEViL integriert ist. Die Funktionalität des GDED wird ebenfalls in den DEViL-Designer integriert (Abb. 8). Für jede Unterrolle *VPFormElement* wird ein Container automatisch in die generische Zeichnung eingefügt.

Listen Muster Das Listen Muster (*VPSimpleList*) wird häufig im Zusammenhang mit SUB Knoten verwendet. Es stellt eine Reihe von Objekt- oder Attributknoten in einer Liste dar. Es hat die beiden Rollen *VPSimpleList*, die an SUB Knoten und *VPSimpleListElement*, die an Objekt- oder Attributknoten vererbt werden. Kontrollattribute können Ausrichtung, Abstand der Listenelemente sowie Farben modifizieren. In Abbildung 43 ist die Vorschau für das *VPSimpleList* Muster zu sehen. Es wurde eine vertikale Ausrichtung der Elemente gewählt, horizontal ist ebenso möglich. Zusätzlich wurde die Hintergrundfarbe und der Elementabstand modifiziert.

Mengen Muster Das Mengen Muster (*VPSet*) stellt eine Liste von Objekten als Menge dar. Die Objektknoten können dabei auf einer Zeichenfläche beliebig im Raum angeordnet werden. Kontrollattribute können Vorder- und Hintergrundfarbe ändern. Die Vorschau für das *VPSet* Muster zeigt, wie beim *VPSimpleList* Muster, einige Anwendungen bestimmter Layoutkontrollattribute.

VPTextPrimitive Das *VPTextPrimitive* Muster kann auf Attributknoten angewendet werden und stellt beliebigen Text dar, der in Schriftart und Farbe geändert werden kann. Anders als beim *VPForm* und *VPSimpleList* Muster wurde keine zusätzliche Sicht definiert. Die Kontrollattribute verändern den Beispieltext im Piktogramm des Musters direkt, wie in Abbildung 44 zu sehen.

Tabellen Muster Das Tabellen Muster (*VPTable*) stellt Tabellen dar und besteht aus den Rollen *VPTable*, *VPTableRow* und *VPTableCell*. *VPTable* repräsentiert die Tabelle

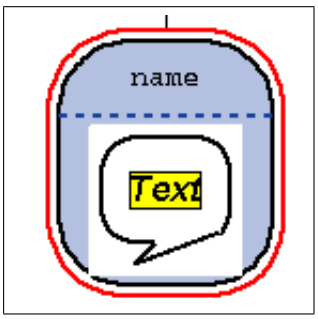

Abbildung 44: Vorschau für die *VPValTextPrimitive* Rolle

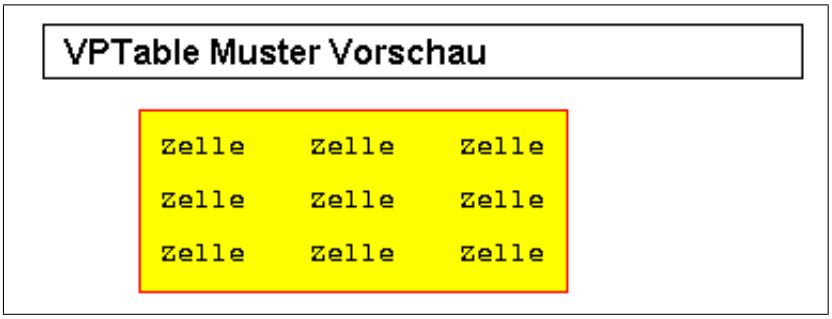

Abbildung 45: Die Vorschau für das *VPTable* Muster

und wird an einen Objektknoten angewendet, *VPTableRow* stellt ein Zeile der Tabelle dar und muss auf einen SUB Knoten angewendet werden. Die Vorschau ist genauso wie beim *VPSimpleList* Muster realisiert und in Abbildung 45 zu sehen. Kontrollattribute können hier neben den Farben auch Abstände zwischen Zellenelementen modifizieren.

VPFormList Das *VPFormList* Muster stellt geschachtelte Listen in einem Formular dar. Es ist eine spezielle Variante des *VPForm* Musters und benötigt statt einer generischen Zeichnung drei. Die Vorschausicht entspricht somit der des *VPForm* Musters mit dem Unterschied, dass drei generische Zeichnungen editiert werden können.

Generisch definierte Muster Selbstdefinierte Muster werden durch das in Abbildung 46 zu sehende Icon dargestellt. Das Objekt selbst hat eine Referenz auf eine Musterdefinition in der Bibliothek. Kontrollattribute werden hinzukopiert und können in einer eigenen Sicht editiert werden (siehe Abbildung 47).

4.4.4 Spezifikation von textuellen Bestandteilen

In Sicht-Definitionen kann spezifiziert werden, wie bestimmte textuelle Bestandteile von Sichten aussehen. Natürlich könnte dieser Teil auch im Rahmen der restlichen Sicht-Spezifikation durch attributierte Grammatiken ausgedrückt werden. Da textuelle Repräsentationen jedoch ein “einfacher“ Spezialfall visueller Darstellungen sind, sollte eine Spezifikation besonders bequem modellierbar sein.

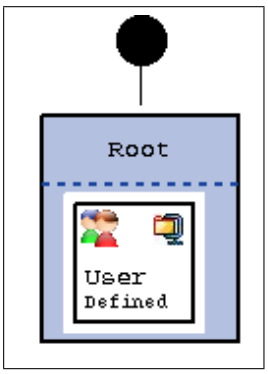

Abbildung 46: Das Piktogramm für benutzerdefinierte Muster

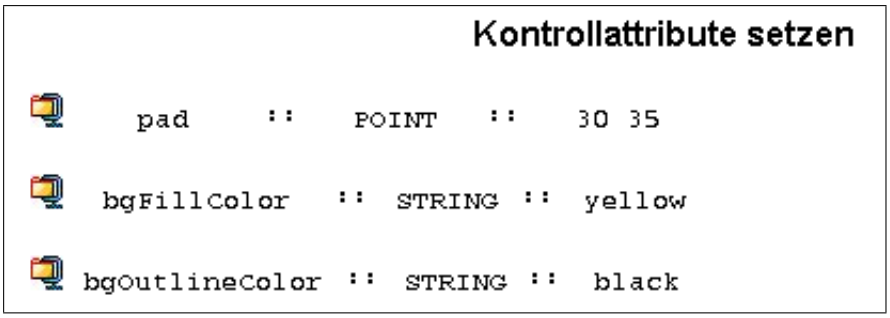

Abbildung 47: Sicht zum Editieren von Kontrollattributen

```
IFStmt: "IF  (" expr ") THEN " stmts "ELSE" stmts.
```

In diesem Beispiel soll der Knoten *IFStmt* textuell dargestellt werden. Die Zeichen in Anführungszeichen sind Terminale. Die restlichen Teile stellen Attribute des Objektknotens dar.

Um diese textuellen Bestandteile in der Umgebung auch graphisch spezifizieren zu können, wurde das *VPFlowForm* Muster verwendet. Es wird auf einen Objektknoten angewendet und bedeutet, dass Attribute des Knotens textuell dargestellt werden sollen. Zwischen die Attribute können dann, wie auch bei der Spezifikation textueller Bestandteile, Textknoten eingefügt werden. Beim Start der Codegenerierung wird der Strukturbaum dann der Reihenfolge von links nach rechts durchlaufen, wie in Abbildung 48 zu sehen ist. Um auch Listen von Objekten textuell darstellen zu können, gibt es das *VPFlowList* Muster. In Abbildung 49 ist eine Implementierung einer Fallunterscheidung zu sehen. Die Anweisungen im “then“-Teil sind eine Liste von Statements. An diese Liste wird die *VPFlowList* Rolle gebunden, an die Unterknoten die *VPFlowListElement* Rolle. Die Liste wird also textuell dargestellt. Um die Reihenfolge der Knoten ändern zu können, ist es wichtig, dass sowohl Attribute als auch SUB Knoten gleichwertig in die Baumstruktur eingegliedert sind.

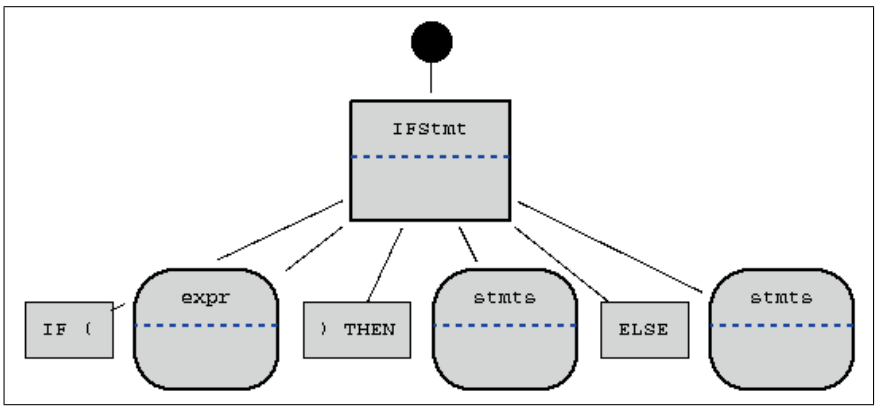

Abbildung 48: Spezifikation textueller Bestandteile

4.4.5 Konsistenzbedingungen für die Sicht der visuellen Muster

Wie bereits im Abschnitt zu den Musterüberdeckungen erwähnt, gibt es einige Bedingungen, die gelten müssen, damit eine Musterkombination korrekt ist. Zum einen dürfen bestimmte Rollen nur an bestimmte Knotentypen vererbt werden. Zur Spezifikation dieser Bedingung kann durch die Attribute *ObjectNode*, *SUBNode*, *VALNode* und *ReferenceNode* in der Sicht zur Spezifikation visueller Muster ausgewählt werden, an welchen Knotentyp die Rolle angewendet werden darf (siehe Abbildung 32 auf Seite 38). Wurde die Rolle in der Sicht nicht an den korrekten Knotentyp vererbt, so wird die Rolle rot umrandet wie in Abbildung 50 zu sehen. Hier wurde die *VPForm* Rolle nicht wie vorgeschrieben an einem Objektknoten angewendet, sondern an einem SUB Knoten.

Eine weitere Konsistenzeigenschaft, die erfüllt sein muss ist, dass alle Interfaces, die die Rolle benötigt, auch am selben Knoten verfügbar sind. Falls Interfaces fehlen, so wird der gesamte Knoten rot umrandet. Wie in Abbildung 50 zu sehen, ist nicht nur der Knotentyp falsch, sondern dem *VPForm* Muster fehlt auch das Interface *VPContainer-PositionInterface*, was sich ebenfalls in einer DEViL Fehlermeldung äußert. Um diesen Fehler zu umgehen, ist es nötig, dass die Rolle mit weiteren Rollen kombiniert wird, die die entsprechenden Interfaces bereitstellen.

Die dritte Konsistenzeigenschaft, die für Muster gelten muss, ist die Positionierung der einzelnen Rollen eines Musters in der Sicht. Die *VPForm* Rolle benötigt so zum Beispiel *VPFormElement* Rollen, die direkt unter der *VPForm* Rolle stehen. Es gibt auch Rollen, bei denen die Unterrollen beliebig tief unterhalb hängen dürfen. Diese Eigenschaft wird in der Sicht zur Spezifikation der Muster durch das boolsche Attribut *direct* der Musterrolle angegeben. Ist *direct* aktiviert, so muss die zugehörige Rolle an einem Knoten direkt unterhalb der höheren Rolle liegen.

Hier ist es nötig, Vererbungshierarchien mit abstrakten Klassen zu berücksichtigen, wie in Abbildung 51 zu sehen ist. Die konkreten Klassen im unteren Teil der Abbildung bekommen durch die Vererbungshierarchie die Rollen der übergeordneten Klassen mit.

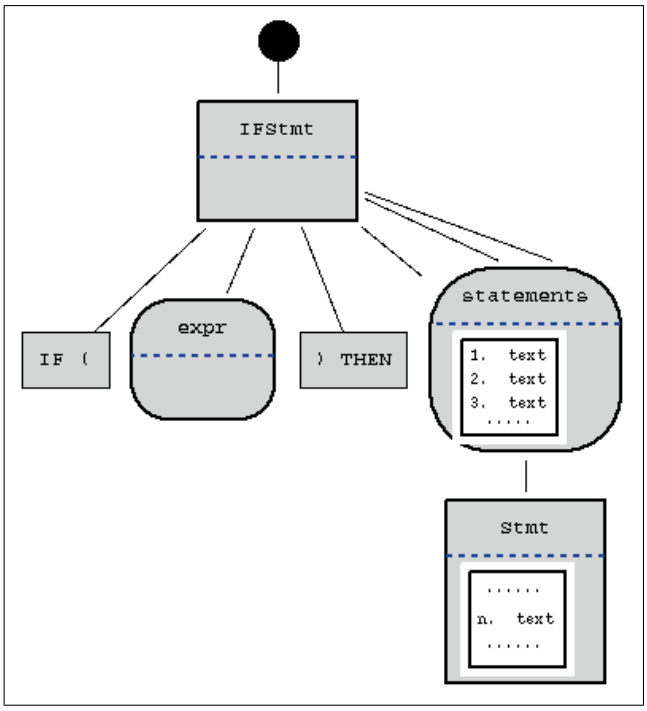

Abbildung 49: Spezifikation textueller Listen

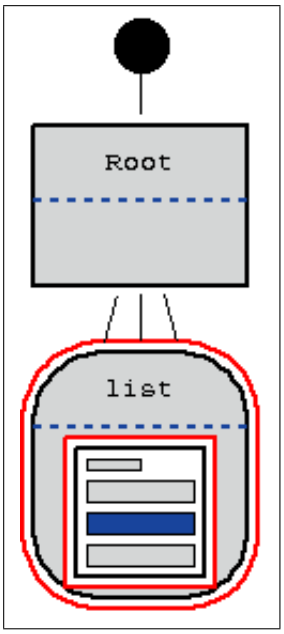

Abbildung 50: Ein falsch angewendetes Muster

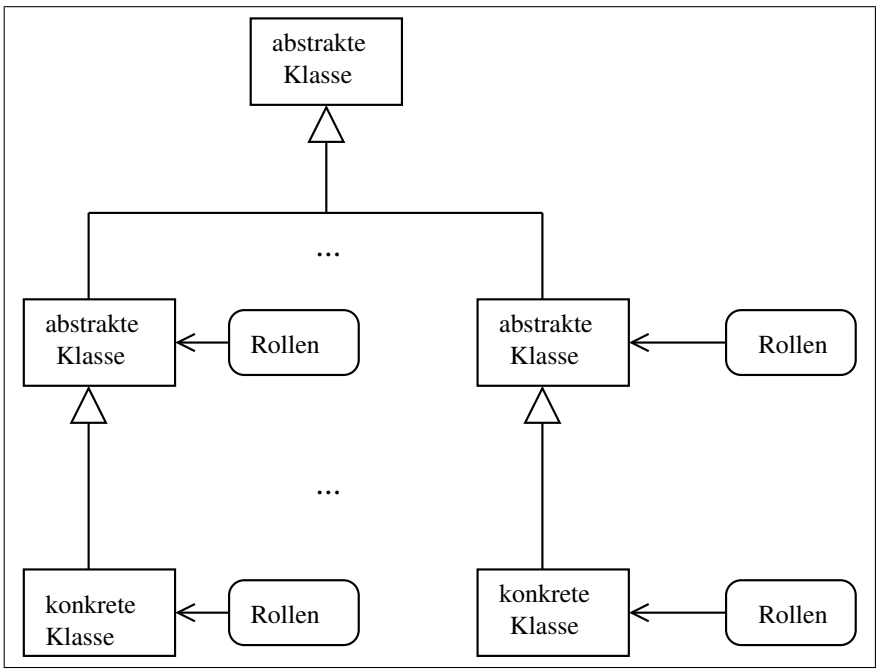

Abbildung 51: Sammeln von Rollen bei Vererbungshierarchien

Wird ein Rollenteil eines Musters an die oberste abstrakte Klasse gebunden, so kann die untergeordnete Rolle des Musters weiter unten an den Baum gebunden werden, ohne einen Fehler zu produzieren. Der untere Rollenteil kann an Vorkommen einer konkreten Klasse angewendet werden. Die *VPForm* Rolle könnte so zum Beispiel an die oberste abstrakte Klasse gebunden werden und die *VPFormElement* Rolle an die erste konkrete Klasse unterhalb der abstrakten Klasse, ohne eine Fehlermeldung zu produzieren. Zu berücksichtigen ist, dass die Vererbungstiefe beliebig groß sein kann. Diese Konsistenzeigenschaft wird von DEViL in den Musterdefinitionen zur Übersetzungszeit überprüft und entspricht in der *VPForm* Variante folgendem LIDO Code:

```
VPForm CONSTITUENTS VPFormElement
          SHIELD (RelationObjectClass)
```

Der umgekehrte Konsistenzcheck überprüft, ob einer Unterrolle die entsprechende Rolle übergeordnet ist. Auch hier müssen Vererbungshierarchien mit ihren Musteranwendungen überprüft werden. Diese Konsistenzeigenschaft wird genauso wie die anderen für jede Rolle überprüft. Gibt es hier eine Fehlermeldung, so steht diese in einer Dialogbox, die aktiviert wird, falls der Benutzer auf den DEViL Konsistenzcheck-Knopf klickt.

Die letzte Konsistenzeigenschaft, die erfüllt sein muss, ist, dass an einem Knoten nicht dieselben Interfaces von unterschiedlichen Rollen bereitgestellt werden. Dies würde einen Laufzeitfehler des Strukturediteors zur Folge haben.

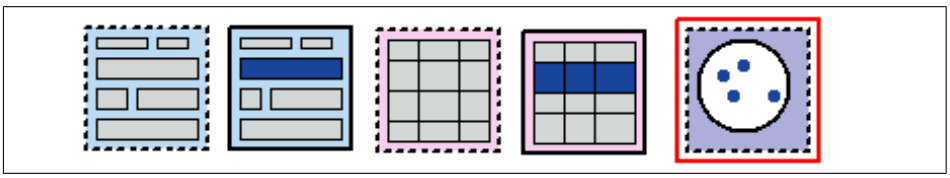

Abbildung 52: Die Rollen Piktogramme: *VPForm*, *VPFormElement*, *VPTable*, *VPTableRow*, *VPSet* mit Fehlermeldung (von links nach rechts)

4.4.6 Design der Rollen-Piktogramme

Das Design der Rollen-Piktogramme sollte insgesamt einheitlich und dezent wirken. Wie in Abbildung 52 zu sehen, werden Rollen, die zu einem Muster gehören, mit einem gleichfarbigen Hintergrund ausgestattet. Die Zusammengehörigkeit mehrerer Rollen kann so in der Sicht auch bei komplexeren Mustern gewährleistet werden. Rollen, die zum Layout beitragen, werden durch einen schwarz gepunkteten Rand dargestellt, in der Abbildung z.B. die *VPFormRolle*. Handelt es sich bei der Rolle um ein Element aus einer Menge, so wurde versucht, dies farblich dunkelblau abzuheben, z.B. *VPFormElement*. Eine komplexere Rollenkombination ist in Abbildung 36 auf Seite 43 zu sehen.

4.5 Sprachentwurf für die Codegenerierung

In Abschnitt 3.2 auf Seite 16 wurde die Codegenerierung erläutert. Strukturediteren können so den aktuellen Strukturbaum, je nach Spezifikation, textuell ausgeben. So könnte zum Beispiel der Editor für UML Statechart Diagramme um ein Codegenerierungsmodul erweitert werden, das es erlaubt, Instanzen textuell als Webseiten darzustellen. Ein Strukturediteor für UML Klassendiagramme könnte als Ausgabe Java Code erzeugen. Die Spezifikation von Codegenerierungsmodulen soll nun genauer betrachtet werden, um eine geeignete visuelle Sprache zu finden.

Struktur der textuellen Codegenerierung Bei der Codegenerierung werden an Klassen des abstrakten Strukturbaums Attributberechnungen gebunden. Diese Attributberechnungen werden dann an ein PTG Muster übergeben, das quasi Textlücken durch die berechneten Attribute ersetzt und so den Code generiert. In Abbildung 53 ist dies an den beiden Klassen *StatechartDiagram* und *State* zu sehen. Die Spezifikation dieser Attributberechnungen wird durch den Namen des Codegenerierungsmoduls als Präfix (hier *codeGen*) gefolgt vom Namen der Strukturklasse eingeleitet.
Die eigentlichen Berechnungen finden dann in einem Block, der durch *COMPUTE* eingeleitet und mit *END;* beendet wird, statt (Zeilen 4 - 10 bzw. 13 - 17). Diese Umgebung wird im Folgenden *Attributberechnungsblock* genannt. Wie in Abbildung 53 zu sehen ist, gibt es neben dem Attributberechnungsblock noch einen Block zur Deklaration von Variablen (Zeile 1 und 2). Diese Variablen dürfen dann im Attributberechnungsblock benutzt werden. Der Zugriff erfolgt mit dem *THIS*-Operator (*THIS.code* in Zeile 9).

```
1.  SYMBOL codeGen_StatechartDiagram: myVar: VLString;
2.  SYMBOL codeGen_StatechartDiagram: code: PTGNode;
3.  SYMBOL codeGen_StatechartDiagram
4.  COMPUTE
5.   SYNT.code = PTGStatechartDiagram(THIS.pers_name,
6.                 CONSTITUENTS codeGen_State.code
7.                   SHIELD codeGen_XORState.code
8.              WITH (PTGNode, PTGNewLine, IDENTICAL, PTGNull));
9.   PTGOutWindow("code.txt", THIS.code);
10.  END;
11.
12. SYMBOL codeGen_State
13. COMPUTE
14.  SYNT.code = PTGState(THIS.pers_name,
15.                 CONSTITUENTS codeGen_Attribute.code
16.               WITH (PTGNode, PTGNewLine, IDENTICAL, PTGNull));
17. END;
```

Abbildung 53: Struktur der Codegenerierung

```
1.  StatechartDiagram:
2.  "Diagram: " $1 string [IndentNewLine]
3.  $2
4.
5.  State:
6.  "State: " $1 string [IndentNewLine]
7.  $2
```

Abbildung 54: PTG Fragment

Die Ausgabe der Codegenerierung findet in Zeile 9 statt. Hier wird spezifiziert, dass der Inhalt der Variable *code* in einem Fenster mit Namen “code.txt“ ausgegeben werden soll. Eine Ausgabe in eine Datei ist ebenfalls möglich und erfolgt analog mit der Funktion *PTGOutFile*. Die Variable *code* ist dabei eine besondere Variable: sie ist vom Typ “PTGNode“. Ein PTGNode ist ein Knoten, der Text enthält. Der Text stammt dabei sowohl von Variablenwerten als auch von Werten weiterer PTGNodes. In Zeile 5 wird dem PTGNode *code* eine PTG Funktion *PTGStatechartDiagram* zugeordnet, in dem die übergebenen Variablen *THIS.pers_name* und *codeGen_State.code* durch Platzhalter ersetzt werden. In Abbildung 54 ist die Ersetzung in Zeile 2 zu sehen. In Zeile 6 aus Abbildung 53 wird der PTG Funktion eine weitere Variable übergeben. Es handelt sich dabei um einen PTGNode. Die Anweisung bedeutet, dass alle PTGNodes zusammengesammelt werden sollen, die im Attributberechnungsblock des Strukturobjekts *State* deklariert wurden. Diese Knoten sollen durch eine neue Zeile miteinander verknüpft werden. Die Variable in Zeile 3, aus Abbildung 54, wird somit durch so viele

Textfragmente, wie sie in Zeile 5, aus Abbildung 54, spezifiziert sind, ersetzt, wie es Objekte vom Typ *State* gibt.

Entwurf der visuellen Sprache zur Codegenerierung Der Entwurf einer visuellen Codegenerierung für den DEViL-Designer, wie im vorherigen Abschnitt erläutert, entspricht einer Baumstruktur. Codefragmente werden dabei Strukturobjekten zugeordnet und mit anderen Codefragmenten entsprechend ihrer Tiefe im abstrakten Strukturbaum miteinander verknüpft. Aus diesem Grund sollte die visuelle Spezifikation einer Codegenerierung ebenfalls einem Baum entsprechen. Außerdem ist dem Anwender die Baumstruktur schon aus der Spezifikation der visuellen Muster bekannt.

Als Baumknoten werden die Klassen der abstrakten Struktur verwendet. Die Baumknoten werden entsprechend ihrer Position in der abstrakten Struktur dargestellt. Ist eine Klasse A in einer Klasse B aggregiert enthalten, so hängt A im Baum unterhalb von B. In jedem Knoten existieren Einfügestellen für PTGNodes und für selbst zu definierende Variablen. Diese Einfügestellen entsprechen dem Attributberechnungsblock in LIDO. Als Vorbelegung existiert hier immer ein PTGNode mit Namen “code“. In diesem PTGNode werden alle Attribute und Referenzen hinzukopiert, die an der zugeordneten Klasse existieren. Zusätzlich werden Strukturobjekte eingefügt, die eine Klasse mit einer aggregiert enthaltenen Klasse verknüpfen können. Diese Objekte entsprechen dem CONSTITUENTS Konstrukt in LIDO. Um die strukturierte Textausgabe miteinzubinden, können vom Benutzer PTG Primitive benutzt werden, die in der Reihenfolge beliebig innerhalb eines PTGNodes platziert werden können. PTG Primitive können einfache Textausgaben sein oder Elemente, die den Text strukturieren, wie ein Zeilenumbruch oder Einrückungen. Alle Sprachelemente der Codegenerierung können vom Benutzer deaktiviert werden, falls sie in der Codegenerierung nicht berücksichtigt werden sollen.

In Abbildung 55 ist ein Knoten zu einer Klasse mit Namen “Diagram“ zu sehen. In dem eingeschachtelten PTGNode “code“ sind die Attribute und Referenzen der Klasse “Diagram“ automatisch hinzugefügt worden. Der PTGNode “code“ stellt seinen Inhalt in einem Ausgabefenster dar, was an dem Piktogramm neben dem Namen zu erkennen ist. Eine Ausgabe in eine Datei ist ebenfalls möglich. Zusätzlich ist ein Sprachelement mit Namen “State.code“ automatisch eingefügt worden, das dem erwähnten CONSTITUENTS Konstrukt entspricht. Es referenziert auf den PTGNode “code“ der Klasse “State“ unterhalb der Klasse “Diagram“. Durch interne Kontrollattribute kann angegeben werden, wie die Objekte miteinander verknüpft werden sollen. Zur Auswahl stehen: *PTGNewLine* (Verknüpfung durch Zeilenumbruch), *PTGCommaSeq* (Verknüpfung durch Kommata) sowie *PTGSeq* (Verknüpfung durch Leerzeichen), in Abbildung 55 wurde die Kommaseparierung ausgewählt. Außerdem kann eine Liste mit SHIELD Klauseln angegeben werden (in Abbildung 55 verdeutlicht durch das Ritter-Piktogramm).

Die Darstellung der Sprachelemente innerhalb der PTGNodes erfolgt mit einer erweiterten Variante des *VPFlowList* Musters. Das *VPFlowList* Muster stellte in der

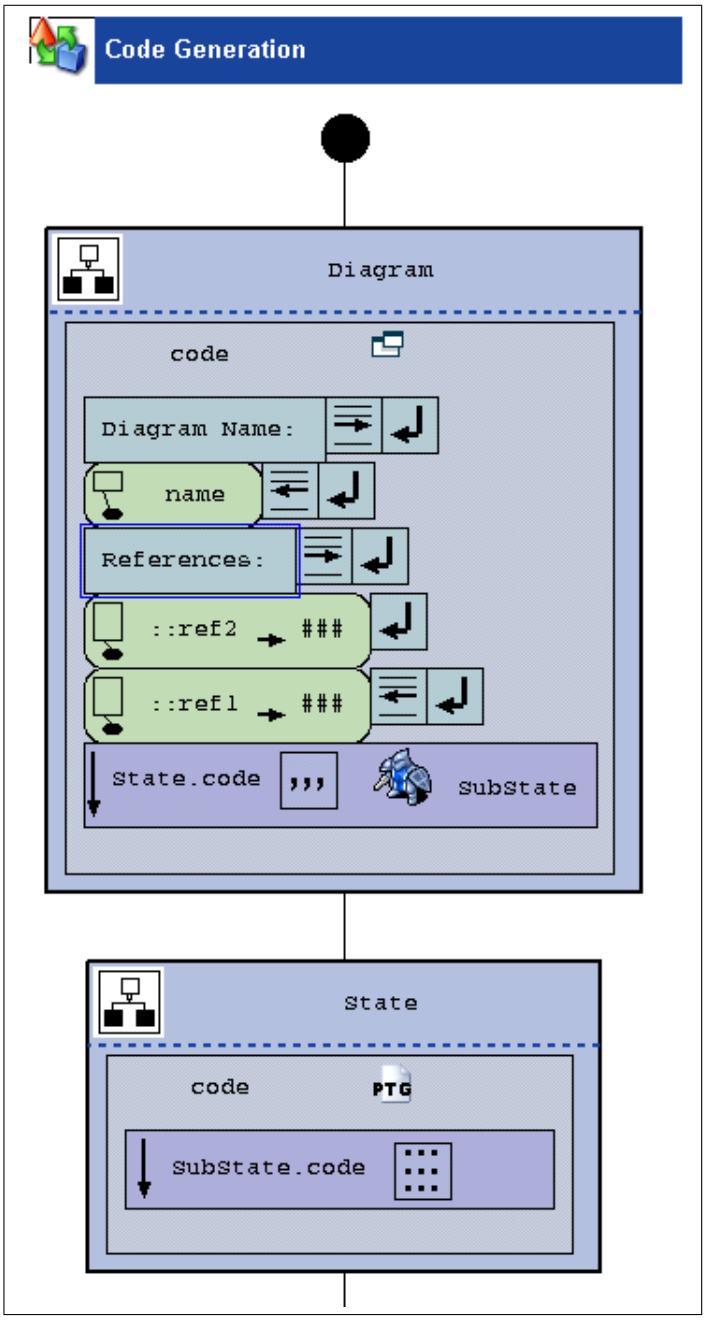

Abbildung 55: Visuelle Sprache für die Codegenerierung (PTG-Primitive sind in türkis dargestellt)

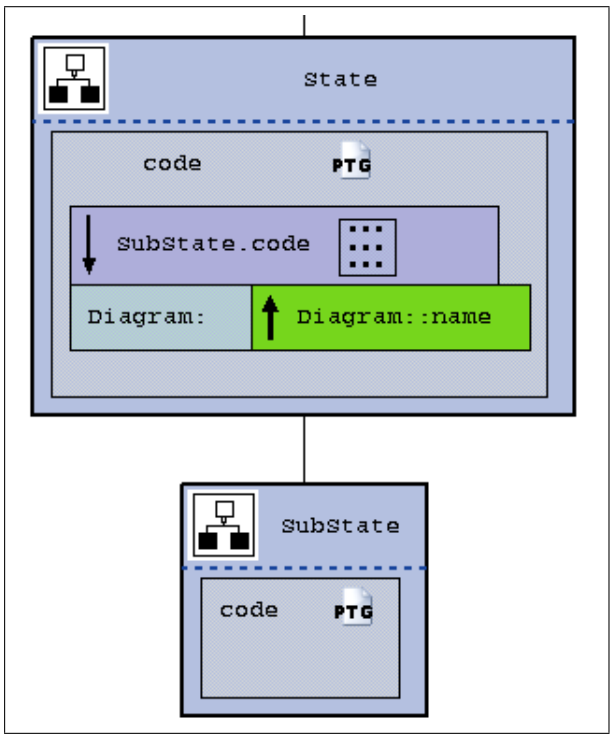

Abbildung 56: Visuelle Sprache für die Codegenerierung: entfernter Attributzugriff mit dem INCLUDING Konstrukt

ursprünglichen Version nur Textfragmente in einer fließenden Anordnung dar. Es wurde im Rahmen dieser Arbeit so erweitert, dass es beliebige Objekte, insbesondere generische Zeichnungen, fließend darstellt. Der Benutzer ist so in der Lage, Attribute und PTG Primitive beliebig horizontal oder vertikal anzuordnen. Eine zunächst entwickelte Implementierung mit Listen von Listen erwies sich für den Benutzer zu schwer zu benutzen und wurde verworfen.

Manchmal kann es wünschenswert sein, dass es innerhalb eines Attributberechnungsblocks mehrere PTGNodes gibt, deshalb können beliebig viele zusätzliche PTGNodes eingefügt werden. Die Attribute der umgebenden Klasse werden dann ebenfalls automatisch hinzugefügt.

Um auch einen entfernten Attributzugriff zu ermöglichen, wurde das Sprachelement “IncludingNode“ eingeführt. Es kann beliebig innerhalb eines PTGNodes angewendet werden und es besitzt eine Referenz auf ein Attribut einer Klasse. In Abbildung 56 wird auf das Attribut “name“ der übergeordneten Klasse “Diagram“ zugegriffen (grüner Knoten mit Pfeil nach oben). Die Menge der Attribute ist dabei jedoch eingeschränkt auf Attribute, die sich im Strukturbaum oberhalb des “IncludingNode“ befinden. Damit auf Attribute in einem Schwesterbaum zugegriffen werden kann, gibt es die Möglichkeit sogenannte “Properties“ zu definieren. Dabei wird ein Sprachelement “ResetPropertyNode“ mit einem Attribut als Schlüssel und einem zugehörigen Wert initialisiert. Dies ist ver-

gleichbar mit der Benutzung einer Hashtable. In einem Schwesterbaum wird dann das Sprachelement “GetPropertyNode“ angewendet, das ebenfalls ein Attribut übergeben bekommt und als Schlüssel für den Wert dient.

Damit auch komplexere Anweisungen wie beispielsweise

```
SYNT.code = IF(VLBoolean(SELECT(vlList("isRootNode",THIS.objId),
                                        eval())),
               PTGRootNode(THIS.pers_name),
                PTGNotRootNode(THIS.pers_name));
```

spezifiziert werden können, wurde ein Sprachelement “Variable“ eingeführt, das einen beliebigen Typ und einen beliebigen Wert annehmen kann. Das Sprachelement “VariableReference“ kann dann eine Variable referenzieren und an beliebigen Stellen innerhalb eines PTGNodes benutzt werden.

Generierung der Baumstruktur Die Darstellung zur visuellen Spezifikation der Codegenerierung entspricht einem Baum mit Klassen der abstrakten Struktur als Knoten. Die Position der Knoten im Baum wird durch eine Breitensuche vom Wurzelknoten der Codegenerierung ausgehend berechnet, bei der Vererbungen und Aggregationen berücksichtigt werden.

Hier kann die Breitensuche zur Erstellung des Baums für die Anwendung der visuellen Muster übernommen werden. Zur Kopplung der abstrakten Struktur mit der Sicht zur Spezifikation der Codegenerierung wird der bereits bekannte Kopplungsmechanismus benutzt. Hier ist es jedoch nicht erforderlich, zusätzliche Strukturelemente einzufügen, wie es bei SUB Knoten in der Sicht zur Anwendung visueller Muster nötig war. Für jede Klasse in der abstrakten Struktur existiert genau ein Objekt in der Sicht zur Spezifikation der Codegenerierung. Zyklische Abhängigkeiten müssen aber auch hier wieder beachtet werden. Die Breitensuche liefert auch hier Baumkanten, die benutzt werden und Querkanten zu Objekten, die nicht in die Sicht aufgenommen werden.

Attribute und Referenzen der abstrakten Struktur werden in jedem Baumknoten automatisch hinzukopiert.

Konsistenzbedingungen für die visuelle Codegenerierung Wie bei der Spezifikation der Sichten, gibt es auch bei der visuellen Spezifikation der Codegenerierung einige Konsistenzbedingungen zu beachten. Im Wesentlichen kann der Benutzer hier jedoch weit weniger falsch spezifizieren als beim Anwenden von Mustern. Insbesondere liegt bereits eine konsistente Codegenerierung vor, wenn der Benutzer ein Codegenerierungsmodul neu anlegt und die Wurzel wählt. Es würde sich dabei um eine rudimentäre Codegenerierung handeln, die ohne vom Benutzer definierte Textfragmente auskommt.

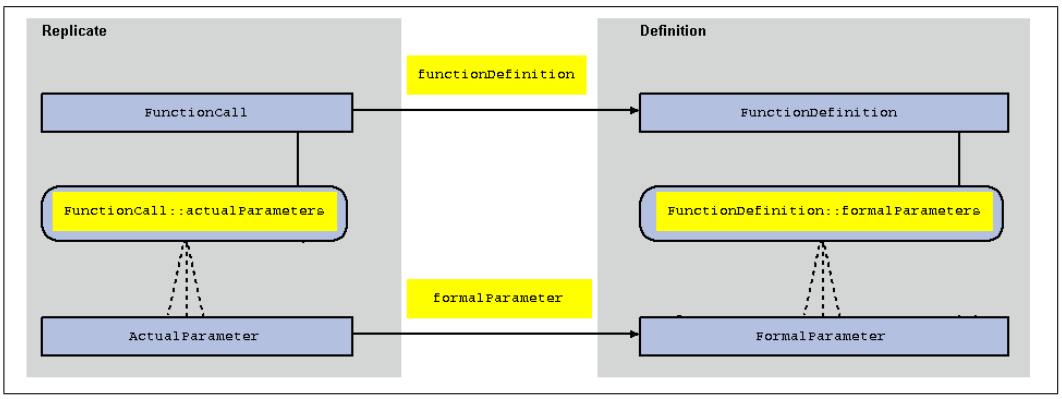

Abbildung 57: Visuelle Spezifikation der Kopplung

Eine Konsistenzbedingung ist, dass unterhalb eines *PTGOutWindow* oder *PTGOutFile* Funktionsaufrufs keine weiteren *PTGOutWindow* bzw. *PTGOutFile* Aufrufe existieren dürfen. Diese Konsistenzprüfung wird durch lokale Korrektheit überprüft, d. h. für alle Knoten gilt, dass einem *PTGOutFile* oder einem *PTGOutWindow* Knoten nur noch *PTGNodes* im Baum übergeordnet sein dürfen.

Weitere Konsistenzbedingungen schränken die Auswahl von Referenzen ein. Ein *IncludingNode* darf nur Attribute referenzieren, die oberhalb im Baum hängen und nicht in Schwesterbäumen. Ähnlich verhält es sich mit *SHIELD* Ausdrücken, die nur Strukturobjekte vom Typ *ConcreteClass* oder *AbstractClass* referenzieren dürfen, die im Baum unterhalb hängen. Diese Referenzbedingungen werden direkt beim Editieren der Referenzen überprüft. Der Benutzer kann somit keine strukturell falschen Spezifikationen erstellen.

4.6 Visuelle Spezifikation des Kopplungsmechanismus

Ein häufig angewandtes Prinzip bei der Spezifikation von Strukturediteren ist das der Kopplung. Dabei werden zwei Teile der Sprachstruktur aneinander gekoppelt. Auch bei der Entwicklung dieser Umgebung wurde Kopplung benutzt, wie in Abschnitt 5.1 beschrieben ist. Die Kopplung ist beispielsweise bei einem Strukturediteur von Vorteil, der Funktionsdefinitionen und Funktionsaufrufe realisiert. Dabei hat eine Funktionsdefinition formale Parameter. Wird diese Funktion aufgerufen, so werden beim Funktionsaufruf die aktuellen Parameter mit den formalen gekoppelt. Dabei können sowohl die formalen als auch die aktuellen Parameter als Liste aufgefasst werden, wobei die aktuellen Parameter eine Referenz auf ihren formalen Parameter haben. In dieser Umgebung ist die visuelle Spezifikation dabei so realisiert, dass visuell zwei SUB Knoten ausgewählt werden können. Die Referenzen der Objekte im ersten SUB Knoten, der Ausgangsliste, werden dann mit den Objekten, die der zweite SUB Knoten aggregiert, gekoppelt. In Abbildung 57 ist dies zu sehen. Auf der linken Seite wird dazu der SUB Knoten ausgewählt, in der aggregierte Objekte automatisch erstellt, modifiziert oder gelöscht werden

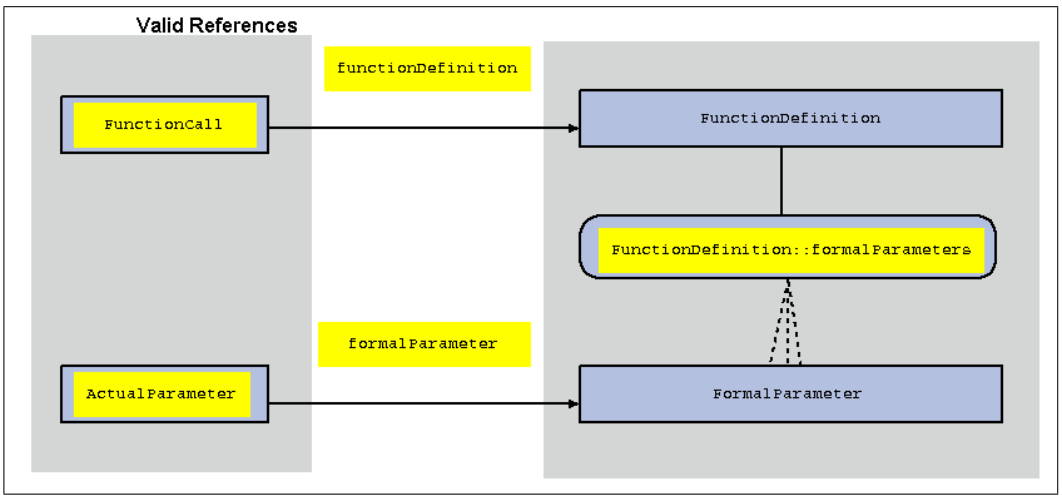

Abbildung 58: Visuelle Spezifikation von gültigen Referenzen

sollen. Bei dem Strukturedítor für Funktionsaufrufe würde es sich hier um eine Liste mit aktuellen Parametern handeln. Die rechte Seite in der Abbildung stellt den SUB Knoten mit der Ausgangsliste dar, im Beispiel wäre das die Liste mit den formalen Parametern. Die Referenz *functionDefiniton* stellt die Verbindung zur Ursprungsliste dar. Jeder aktuelle Parameter hat eine Referenz (*formalParameter*) auf das entsprechende Objekt in der Ursprungsliste. Alle anderen Objekte, wie FunctionCall, ActualParameter, FunctionDefinition und FormalParameter werden automatisch ermittelt. Bei der Codegenerierung des Strukturedítors wird schließlich eine zusätzliche Tcl Datei generiert, die für die Kopplung sorgt.

4.7 Visuelle Spezifikation von gültigen Referenzen

In Strukturedítoren kommt es häufig vor, dass eine Referenz auf eine Teilmenge eingeschränkt werden soll. Ein Beispiel wäre eine Spezifikation von Funktionsdeklarationen mit formalen Parametern und Funktionsaufrufen mit aktuellen Parametern. Referenziert nun ein Funktionsaufruf eine Funktionsdeklaration, so sollen die aktuellen Parameter des Funktionsaufrufs nur auf die formalen Parameter der zugewiesenen Funktionsdeklaration referenzieren. Die Menge der formalen Parameter wird also eingeschränkt auf die Teilmenge der formalen Parameter der Funktionsdeklaration. Dieser Mechanismus ist nun ebenfalls visuell zu spezifizieren, wie in Abbildung 58 zu sehen ist. Dabei muss lediglich auf der linken Seite in der Abbildung der Funktionsaufruf ausgewählt werden und auf der rechten Seite in der Abbildung die Funktionsdeklaration. Bei Aufruf der Codegenerierung werden automatisch zusätzliche Dateien mit den entsprechenden Konsistenzbedingungen generiert.

5 Realisierung

In diesem Kapitel sollen einige interessante Implementierungsaspekte diskutiert werden. Durch die aufwändige Kopplung der Sichten entstehen leider auch Geschwindigkeitsverluste bei den Reaktionen des Editors. Sie werden im letzten Abschnitt dieses Kapitels erläutert.

5.1 Kopplung der Sichten

DEViL bietet eine Funktionalität an, die es ermöglicht, Teile des Modells aneinander zu koppeln. Dadurch können unterschiedliche Sichten zueinander konsistent gehalten werden. Der gesamte Kopplungs-Mechanismus ist in der Skriptsprache Tcl [14] realisiert. Mittels spezieller Tcl Funktionen kann durch Pfadausdrücke innerhalb des Strukturbaums navigiert werden. Ebenfalls durch Tcl Funktionen kann der Strukturbaum verändert werden.

Die Kopplung ist nötig, um für die Sicht zur Spezifikation der visuellen Muster eine Darstellung zu erreichen, die mit der Sicht zur Spezifikation des Modells konsistent ist. Der Zweck, eine spezielle Sicht für das Anwenden der visuellen Muster zu entwickeln, wurde in Kapitel 4.4.2 erläutert. In diesem Kapitel soll nun der technische Ansatz zur Realisierung dargestellt werden.

Prinzipiell wird immer versucht, eine Ausgangsliste von Objekten mit einer Liste in einem anderen Kontext zu koppeln. Die Kopplung entsteht durch Referenzen der Objekte in der Ausgangsliste auf Objekte in der gekoppelten Liste. Es gilt immer die Konsistenzbedingung: existiert ein Objekt in der Ausgangsliste, so existiert auch ein Objekt in der zweiten Liste, das eine Referenz auf dieses erste Objekt hat. Wird das Objekt in der Ausgangsliste gelöscht, dann wird auch das Objekt in der gekoppelten Liste entfernt. Es handelt sich um eine gerichtete Beziehung. Ein Löschen eines Objekts in der gekoppelten Liste hat kein Löschen des Ausgangsobjekts zur Folge.

Die Realisierung erfolgt mittels der Tcl Funktion

```
syncutil::createMissing0
  { defList repList refAttr repCreationProc }
```

defList entspricht der Ausgangsliste, *repList* ist die gekoppelte Liste. *refAttr* ist der Name der Modell-Referenz in der gekoppelten Liste. *repCreationProc* wird eine Funktion übergeben, die das gewünschte Strukturobjekt erzeugt.

Die Standard Implementierung des Kopplungsalgorithmus erzeugt eine symmetrische Darstellung, wie in Abbildung 59 zu sehen. Die Struktur der Objekte ist gleich. Die Referenz *modelRef* zeigt auf ein strukturell gleiches Objekt im Basismodell. Der Kopplungsalgorithmus versucht für jede Klasse im abgeleiteten Modell die lokale Korrektheit herzustellen. Es wird also zunächst in der Klasse A des abgeleiteten Modells über das

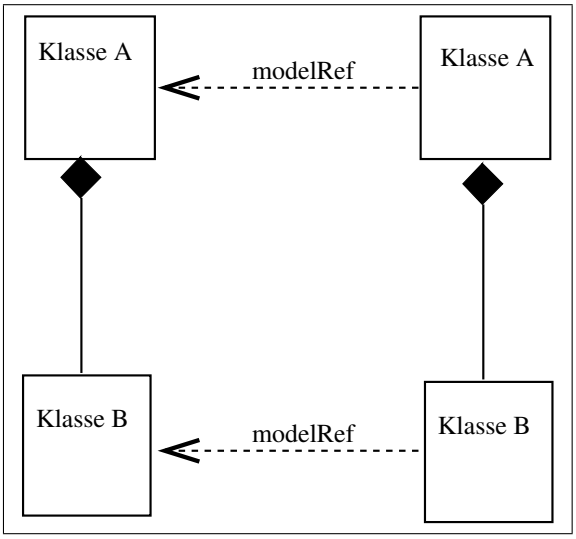

Abbildung 59: Standard Kopplung

modelRef Attribut zur Basismodellklasse A gegangen, von dort zur Basismodellklasse B. Stimmt nun die modelRef Referenz der Klasse B des abgeleiteten Modells mit der Basismodellklasse B überein, dann ist die abgeleitete Struktur konsistent zum Modell. Die Vorgehensweise des Kopplungsalgorithmus ist immer dieselbe: zuerst werden Klassen und Attribute im abgeleiteten Modell gelöscht, die im Basismodell nicht mehr existieren. Danach werden Objekte verschoben, die im Basismodell in andere Klassen umgehängt wurden. Zum Schluss werden im abgeleiteten Modell nicht existierende Objekte und Attribute neu erzeugt.

Wie bereits erwähnt, erzeugt der Standardkopplungsalgorithmus von DEViL nur eine 1:1 Abbildung. Dies ist jedoch in diesem Fall nicht erwünscht, da eine Modellierung eines SUB Knotens der abstrakten Struktur, wie in Abbildung 59 zu sehen, neben der Kardinalität noch zwei weitere Informationen beinhaltet: nämlich den Namen des SUB Knotens und den Typ des SUB Knotens, zum Beispiel:

```
CLASS KlasseA {
 classes: SUB KlasseB*;
}
```

Aus diesem Grund muss ein Attribut *classes* zwischen *KlasseA* und *KlasseB* geschaltet werden (Abbildung 60). Aus dem SUB Attribut *classes* wird ein neues eigenständiges Attribut in der Sicht für das Anwenden visueller Muster (Ellipse). Dies ist nötig, da an SUB Attribute ebenfalls Muster gebunden werden können. In diesem Beispiel könnte an den SUB Knoten *classes* eine *VPSimpleList* Rolle angewendet sein und an den Objektknoten *KlasseB* die Rolle *VPSimpleListElement*. Der entwickelte Kopplungsalgorithmus versucht nun die lokale Korrektheit herzustellen. Ein SUB Attribut im Modell entspricht einer eigenen Klasse, die zwei Referenzen besitzt, eine Referenz zur Klasse in die es eingeschachtelt ist (*refTo* in Abb. 61) und eine Referenz zur Klasse, die dort eingeschachtelt

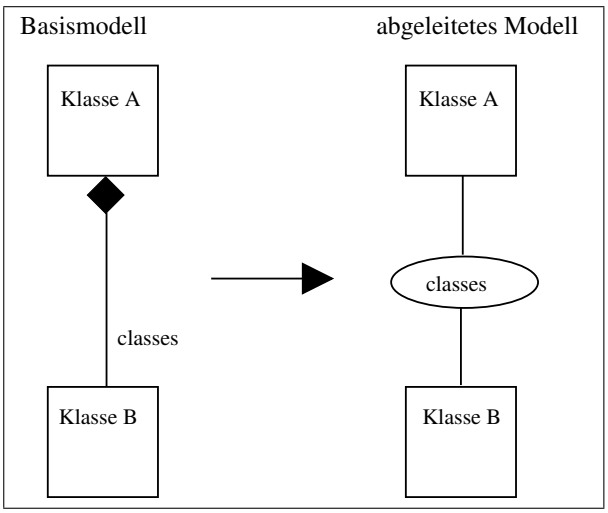

Abbildung 60: Kopplung eines SUB Attributs

wird (*refFrom* in Abb. 61). Zeigen die Referenzen der Klassen im abgeleiteten Modell auf die richtigen Klassen im Basismodell, dann ist die Kopplung lokal korrekt. Stimmt diese Bedingung für alle SUB Attribute, dann ist die Kopplung auch global korrekt. Diese Bedingungen lassen sich in DEViL durch Pfadausdrücke darstellen:

- *.Attributname* liefert die Unterknoten der Eingabeknoten, die das angegebene Attribut repräsentieren.
- *.VALUE* liefert die Werte aller VAL- und REF-Eingabeknoten.
- *.IVALUE* liefert die Menge aller REF-Knoten, die die aktuellen Knoten referenzieren.
- *.CHILDREN* liefert alle Unterknoten der Eingabeknoten.
- *.PARENT* liefert Vaterknoten der Eingabeknoten.
- *.INCLUDING Klassenname* liefert die nächstgelegenen übergeordneten Knoten aller Eingabeknoten, die der angegebenen Klasse angehören.
- *[Klassenname]* liefert alle Knoten der Eingabe, die der angegebenen Klasse angehören.

Die Konsistenzeigenschaft lautet also:

```
c::getList {
 $classA.PARENT.IVALUE[SubAttribute.referenceTo].PARENT
}
```

In diesem Fall befinden wir uns in einer SUB Attribut Liste der Klasse *classA*. Mit PARENT befinden wir uns dann direkt in *classA* und holen dann alle Klassen vom Typ *SubAttribute*, deren *referenceTo* Zeiger auf *classA* zeigt. Für das Strukturobjekt *SubAttribute*

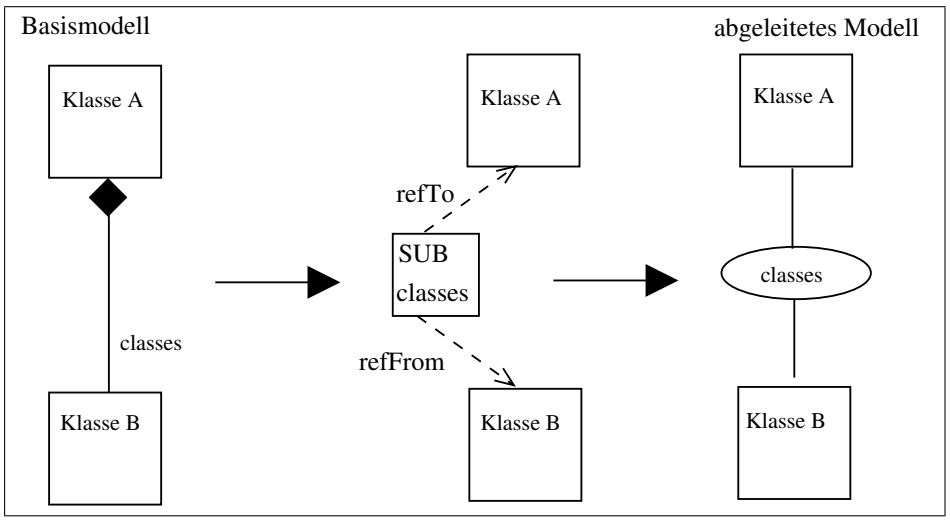

Abbildung 61: Kopplung eines SUB Attributs

im Basismodell wird dann im abgeleiteten Modell ein neues Strukturobjekt eingefügt, das die Anwendung von Rollen erlaubt. Unterhalb dieses neuen Strukturobjekts hängt dann die im Basismodell aggregierte Klasse. Im Folgenden muss nun noch überprüft werden, ob die Zeiger von *SubAttribute* auch im Modell der Sichten korrekt gesetzt sind also ob eine Situation wie in Abbildung 61 vorliegt.

5.1.1 Zyklen

Ein weiteres Problem der Kopplung ist das Erkennen und richtige Darstellen von Zyklen. Ein Zyklus in der Modellstruktur ist in Abbildung 62 links dargestellt. Die Klasse C enthält als aggregiertes Element wieder Klasse A und somit rekursiv den gesamten Unterbaum. Rechts in der Abbildung ist ein Zyklus mit SUB Attributen und Vererbungen zu sehen. Da in der Sicht zum Anwenden der Muster ein Baum generiert werden soll und kein Graph, würde der Kopplungsalgorithmus hier in einer Endlosschleife neue Elemente einfügen. Deshalb ist es nötig herauszufinden, welche Kanten dem Baum hinzugefügt werden müssen. In einem zweiten Schritt müssen Zyklen kenntlich gemacht werden und dafür sogenannte Stellvertreter Klassen eingefügt werden.
Um Baumkanten und Querkanten zu finden, wird die Breitensuche verwendet. Knoten sind die Klassen und als Kanten gelten die *referenceFrom* bzw. *referenceTo* Elemente der Aggregationen und Vererbungsbeziehungen. Referenzen werden übrigens nicht berücksichtigt, da sie eine spezielle Art von Attributen darstellen und immer in den Baum aufgenommen werden. Die Breitensuche wird verwendet, da sie Klassen, die in einem Zyklus vorkommen, möglichst tief in den Baum hängt. Dies bedeutet, dass der entstehende Graph immer relativ ähnlich strukturiert ist, egal mit welchem Knoten in der Nachfolgerliste begonnen wird. Der in Abbildung 63 zu sehende Ausgangsgraph wird durch Tiefensuche entweder in den Graph links aus Abbildung 64 umgewandelt, wenn mit Knoten “2“ begonnen oder in den rechten Graph, wenn mit Knoten “3“ begonnen wird. Die Breitensuche hingegen liefert immer denselben Graph, egal mit welchem Knoten begon-

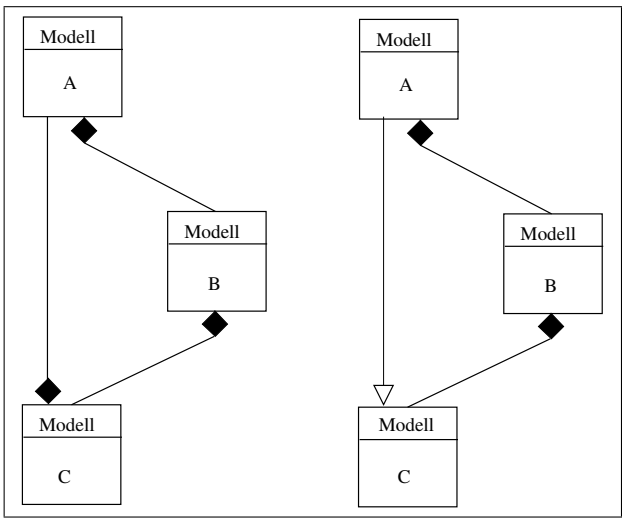

Abbildung 62: Ein Zyklus

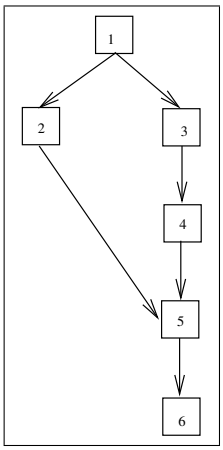

Abbildung 63: Der Ausgangsgraph

nen wird. Die gestrichelten Kanten im Graph sind Querkanten. Sie zeigen an, dass eine Klasse bereits besucht wurde. Dies deutet entweder auf einen Zyklus hin oder auf eine Vererbungsbeziehung zu einer Klasse, die bereits durch SUB Attribute in den Baum eingeschachtelt ist. Diese bereits besuchten Klassen werden in der Sicht zum Anwenden visueller Muster erneut dargestellt und durch ein UML Stereotyp besonders gekennzeichnet, wie in Abbildung 65 zu sehen. Diese Klassen sind jedoch besondere Stellvertreter, da sie keine Subbäume mehr enthalten, andernfalls würde der Kopplungsalgorithmus erneut durchlaufen und sich in einer Endlosschleife befinden.

5.2 Konsistenzbedingungen

Die Realisierung der Konsistenzbedingungen, die überprüft, ob Rollen an den korrekten Knotentyp angewendet wurden, gestaltet sich unproblematisch. Hier wurde der Knoten-

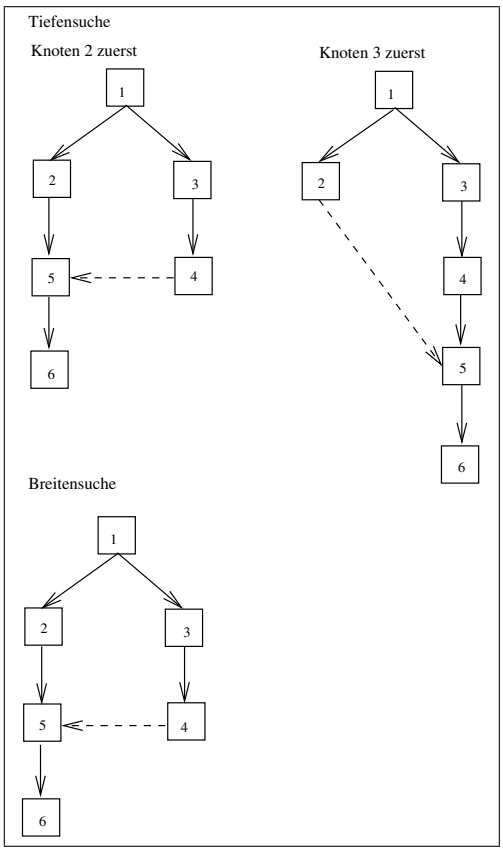

Abbildung 64: Breiten- und Tiefensuche für den Ausgangsgraph aus Abb. 63

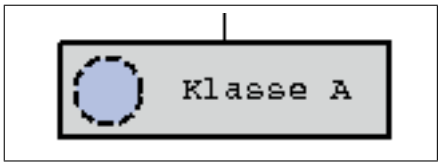

Abbildung 65: Eine Klasse, die bereits im Baum vorkommt

typ anhand der umgebenden Klasse ermittelt und mit dem in der Musterspezifikation abgeglichen.

Die Konsistenzbedingung, die sicherstellen, dass alle benötigten Interfaces auch an einem Knoten vorhanden sind, wurde mittels eines assoziativen Arrays realisiert, in die alle bereitgestellten Interfaces eingefügt werden.

Schwieriger zu realisieren waren die Konsistenzeigenschaften, die überprüft werden müssen, falls eine Rolle an einem Strukturobjekt hängen muss, das sich direkt unterhalb des Strukturobjekts ihrer übergeordneten Rolle befindet. Es handelt sich hier also um die bereits beschriebene CONSTITUENTS (nach unten im Strukturbaum) bzw. INCLUDING (nach oben im Strukturbaum) Konsistenzeigenschaft. Hier muss nämlich

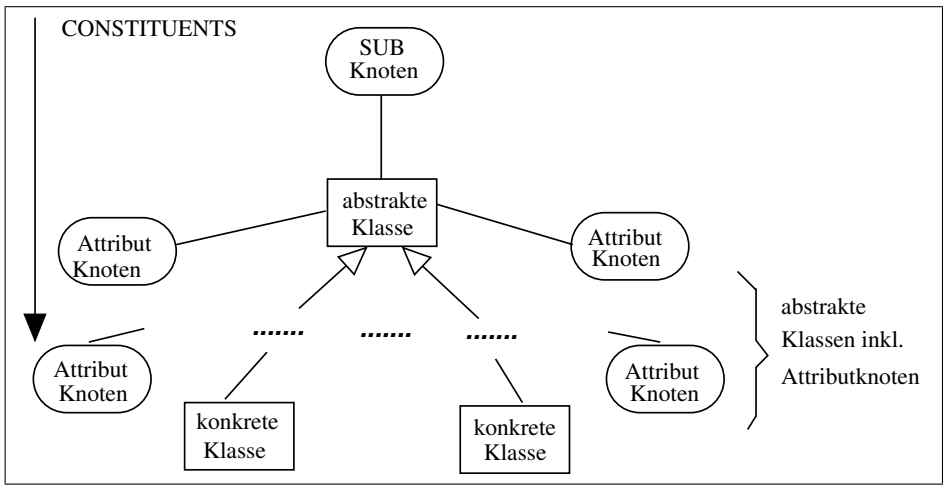

Abbildung 66: Sammeln von Rollen im CONSITUENTS Fall

berücksichtigt werden, dass es Vererbungsbeziehungen gibt, also dass zwischen Rolle und Unterrolle durchaus beliebig viele abstrakte Klassen hängen dürfen. In Abbildung 66 wurde eine Rolle an einen SUB Knoten (oberster Knoten in der Abbildung) gebunden. Nun werden alle Rollen unterhalb dieses Knotens gesammelt. Dabei kann es sein, dass direkt unterhalb eine Reihe von abstrakten Klassen hängt. Diese werden rekursiv durchlaufen, und alle Rollen von Attributen dieser Klassen werden gesammelt. Sobald eine konkrete Klasse erreicht wird, wird die Rekursion gestoppt. Hierbei muss die Vererbungshierarchie berücksichtigt werden.

Der INCLUDING Fall ist etwas einfacher. In Abbildung 67 ist die Suche von Rollen ausgehend von einer konkreten Klasse (unten in der Abbildung) dargestellt. Hier müssen nur die Rollen von abstrakten Klassen, die oberhalb der konkreten Klasse hängen, berücksichtigt werden. Die Rollen von Attributen dieser Klassen werden nicht berücksichtigt.

5.3 Geschwindigkeit

Die Geschwindigkeit der hier ursprünglich entwickelten Implementierung des DEViL-Designers lässt bei dem Statechart Beispiel sehr zu wünschen übrig. In der Tabelle aus Abbildung 68 sind die Lastanforderungen zur Kopplung inklusive Breitensuche und die Konsistenzüberprüfungen in Millisekunden angegeben. Der Gesamtaufwand entspricht etwa 11 Sekunden beim Modifizieren von Teilen der Umgebung, z.B. beim Einfügen eines Zeichenelements in eine generische Zeichnung. Dies ist problematisch und soll durch folgende Maßnahmen umgangen werden:

- Die Sichten sollen nur neu gekoppelt werden, wenn sich Objekte im Basismodell ändern. Wenn sich Objekte im abgeleiteten Modell ändern, soll eine Neukopplung nicht stattfinden.

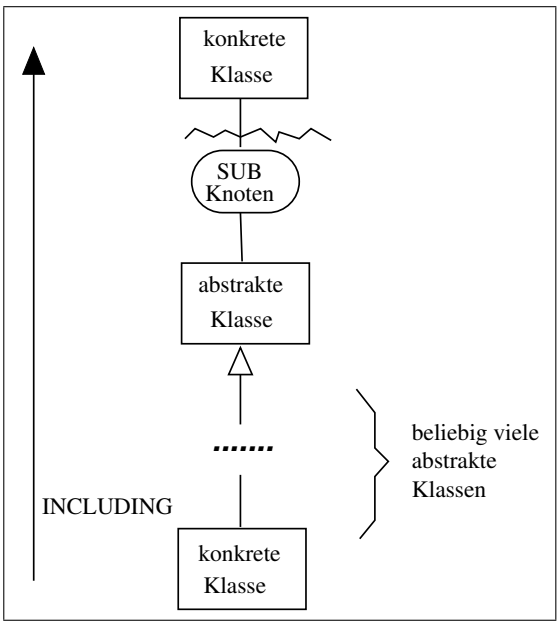

Abbildung 67: Sammeln von Rollen im INCLUDING Fall

	Millisekunden	Prozent
Kopplung	3640	32,9%
Konsistenzchecks	262 (43 Knoten)	2,3%
Breitensuche	78	0,7%
Erneuerung der Sichten	7074	63,9%

Abbildung 68: Anteile der Geschwindigkeitsverluste

- Die Sichten sollen nur neu gekoppelt werden, wenn sie auch wirklich sichtbar sind.
- Piktogramme für die Rollen sollten als Bilder und nicht als generische Zeichnungen vorliegen.

Wie in Abbildung 69 zu sehen ist, fallen die Kopplungsanteile heraus, da die Sichten nur noch neu gekoppelt werden, falls sich Objekte in der Basisstruktur ändern. Die Erneuerung der Sichten benötigt nun nur noch fast halb so viel Zeit, was vor allem daran liegt, dass die generischen Zeichnungen nun als Bilder vorliegen.

	Millisekunden	Prozent
Kopplung	0	0%
Konsistenzchecks	365 (43 Knoten)	10,5%
Breitensuche	83	2,4%
Erneuerung der Sichten	3018	87,1%

Abbildung 69: Anteile der Geschwindigkeitsverluste nach der Optimierung

6 Evaluation

In diesem Teil der Arbeit soll der hier entwickelte DEViL-Designer evaluiert werden. Dazu soll zunächst eine Evaluierung durch ein formales Modell, das in [2] entwickelt wurde, stattfinden. Anschließend soll die Umgebung durch mehrere Probanden mit unterschiedlicher Erfahrung im Bereich des visuellen Sprachentwurfs in der Praxis getestet werden.

6.1 Evaluation nach Green und Petre

Green und Petre stellen in [2] ein Modell zur Untersuchung visueller Programmiersprachen vor. Dieses Modell soll dabei keine Richtlinien an einen Designer oder Programmierer zur Erstellung von Editoren für visuelle Programmiersprachen vorgeben. Es soll lediglich eine gegebene Programmierumgebung hinsichtlich der Nutzbarkeit durch den Anwender untersucht werden. Anhand dieser Richtlinien soll nun die hier entwickelte Umgebung untersucht werden:

Abstraction Gradient Abstraction Gradient bedeutet, dass eine Gruppe von Elementen als eine Einheit behandelt werden kann, sei es zur Bequemlichkeit oder um die konzeptionelle Struktur zu ändern. Abstraktion kann bei Nicht-Experten zu Unverständnis führen, insgesamt kann Abstraktion eine Programmstruktur jedoch vereinfachen.

Im hier entwickelten DEViL-Designer tritt Abstraktion an mehreren Stellen auf. So können in der Sicht zur Spezifikation des Modells Klassen genauso wie Attribute behandelt werden. Beide Strukturen können einfach visuell eingefügt werden, ohne die konkrete Syntax auf Strukturebene kennen zu müssen. In der Sicht für das Anwenden der visuellen Muster werden alle Rollen gleich behandelt. Egal um welches Muster es sich handelt, der Benutzer muss keine Implementierungsdetails wissen, um sie anzuwenden. Generell gibt es im DEViL-Designer nur drei Abstraktionsniveaus: die Sicht zur Spezifikation der abstrakten Struktur, die Sicht zur Spezifikation der visuellen Muster und die Sicht zur Modellierung der Codegenerierung.

Closeness of Mapping Closeness of Mapping beschreibt die Abbildungsdistanz bei einer Abbildung einer Problemwelt in eine Programmwelt. Je näher diese beiden Welten beieinander liegen, umso einfacher ist die Problemlösung zu realisieren. Ideal ist es, wenn sich das Problem des Entwicklers direkt in die Programmbausteine übertragen lassen würde, ohne dass zusätzlicher konzeptioneller Aufwand betrieben werden muss. Ein existierendes Konzept in der Problemwelt ist direkt in der Programmwelt erkennbar.

Die Abbildungsdistanz im DEViL-Designer ist in der Sicht für die Spezifikation der abstrakten Struktur durch den Einsatz von UML recht gering. Die Konzepte von UML und der hier entwickelten Sprache sind nahezu identisch. Kennt der Benutzer bereits UML, so kann er auch Strukturmodelle in DEViL visuell spezifizieren. Der Benutzer braucht dabei nicht die zugrunde liegende Syntax kennen. Auch Elemente

wie Konsistenzchecks und Initialisierungen können einfach visuell eingefügt werden. In der Sicht zur Spezifikation der visuellen Muster können Rollen direkt an Knoten gebunden werden. Die Rollendiagramme, die in einer Baumstruktur vorliegen, können direkt auf die Baumstruktur der Sicht angewendet werden. Dem Anwender wird durch die Baumstruktur bei der Anwendung visueller Muster geholfen, da Rollen in der Regel vom Wurzelknoten ausgehend zu den Blättern angewendet werden. Ein Top-Down Entwurf hat sich in der Praxis als sinnvoll erwiesen. Die textuelle Codegenerierung entspricht einem Durchlauf von der Wurzel ausgehend durch den abstrakten Strukturbaum. Dies wurde in der Sicht für die visuelle Spezifikation der Codegenerierung ebenfalls ausgenutzt und durch Benutzen eines Baumes als visuelles Muster abgebildet.

Consistency Die Konsistenz beim Design visueller Sprachen beschreibt die Harmonie im Programmentwurf. Dabei gilt: “Gleiches soll gleich behandelt werden“. Ein Programm wäre beispielsweise inkonsistent, wenn es die Ausgabe von Variablen des Typs Integer und Real erlaubt, jedoch nicht die Ausgabe von Variablen des Typs Boolean. Konsistenz zeigt sich auch darin, dass unbekannte Teile der Programmstruktur aus anderen bekannten Teilen “erraten“ werden können.

Konsistenz zeigt sich vor allem in der Sicht für das Anwenden der visuellen Muster. Jede Rolle wird hier gleich behandelt, egal wie die interne Implementierung aussieht. Die Sicht zur Spezifikation des Modells ist mit UML Klassendiagrammen vergleichbar. Ein Benutzer, der nur die UML Variante kennt, sollte sich ohne Probleme auch in der Umgebung zurechtfinden. Die Sichten für das Anwenden der visuellen Muster und die Codegenerierungssicht benutzen beide eine Baumstruktur.

Diffuseness/Terseness Diffuseness/Terseness beschreibt die Weitläufigkeit bzw. die Knappheit einer Sprache. Es bezieht sich dabei auf die Anzahl der benutzten Piktogramme bzw. Codezeilen. Es ist sehr schwer anzugeben, welcher Umfang der Richtige ist. Sprachen können konzeptionell Gleiches sehr unterschiedlich ausdrücken. Das in dieser Arbeit verwendete Statechart-Beispiel hat insgesamt 659 Codezeilen (davon 158 Zeilen für die abstrakte Struktur, 218 Zeilen für die Sicht), dasselbe Programm wird in der Umgebung durch 9 Icons und 9 Pfeile/Linien für die abstrakte Struktur sowie 16 Icons für die Strukturobjekte in den Sichten und 22 Icons für die Rollen dargestellt. Der DEViL-Designer ist also besonders beim Entwurf der abstrakten Struktur sehr knapp gehalten, was der Übersicht sehr zu Gute kommt. Der Umfang der Spezifikation der Sichten ist besonders hinsichtlich des Platzbedarfs etwas problematisch. Die Darstellung des Baumes benötigt recht viel Platz. Deshalb wurde das Feature zum Ausblenden von Baumteilen hinzugefügt, außerdem kann der Benutzer die DEViL Zoom Funktion benutzen, um sich einen Überblick zu verschaffen.

Error-Proneness Error-Proneness bezeichnet die Fehleranfälligkeit eines Systems. Dabei wird zwischen zwei Arten von Fehlern unterschieden: Fehler, die quasi als Ausrutscher zu betrachten sind, z.B. weil ein Syntaxfehler gemacht wurde und Fehler,

die konzeptionell bedingt sind, da sie auf ein Missverstehen des Systems hindeuten.

Die erste Variante von Fehlern wird im DEViL-Designer zwar nicht vollständig umgangen, jedoch zeigt die Umgebung dem Anwender Fehler bei Bedarf an. Das DEViL System überprüft vor der Codegenerierung alle syntaktischen Fehlerquellen, z.B. das Weglassen eines Attributtyps. Fehler wie die falsche Kombination von Rollen oder das Anwenden einer Rolle an einen falschen Knotentyp werden ebenfalls erkannt. Die Umgebung zeigt dem Benutzer diese Fehler umgehend an. Nicht erkannt werden können Fehler, die das Design eines Strukturediters angehen. Unlogische Spezifikationen können jedoch nicht erkannt werden, wenn sie syntaktisch in Ordnung sind. Ein Strukturediter programmiert sich leider noch nicht von selbst.

Hard Mental Operations Hard Mental Operations beschreibt die Gedächtnisleistung, die ein Anwender bei der Benutzung einer visuellen Umgebung aufwenden muss. Dabei geht es darum, wie kompliziert Konstrukte ausgedrückt werden müssen. In textuellen Programmiersprachen sind komplexe boolsche Ausdrücke oder verschachtelte Fallunterscheidungen häufig ein Problem.

Im DEViL-Designer kann die Spezifikation der abstrakten Struktur bei großen Projekten zu Problemen führen, da Teile eventuell nicht auf dem Bildschirm angezeigt werden können und der Benutzer scrollen muss. Da die Umgebung allerdings nicht für derartig große Projekte konzipiert wurde, stellt dies kein größeres Problem dar. In einer neueren Version des DEViL Generators stehen Strukturediteren eine Zoom Funktion zur Verfügung, die es erlaubt, auch bei größeren Sichten den Überblick zu behalten.
Selbst komplexe abstrakte Strukturen werden durch die Transformation in den Baum übersichtlich aufgelöst. Das Anwenden der visuellen Muster stellt so wenig Probleme dar.

Hidden Dependencies Mit Hidden Dependencies sind versteckte Abhängigkeiten gemeint, die zwischen zwei Objekten bestehen, jedoch nicht explizit ausgedrückt werden. In textuellen Programmiersprachen werden Seiteneffekte als versteckte Abhängigkeiten bezeichnet.

In der hier entwickelten Umgebung konnten keine versteckten Abhängigkeiten identifiziert werden. Eine Änderung der Sichten oder der Codegenerierung erfolgt losgelöst von der abstrakten Struktur.

Progressive Evaluation Häufig ist es nötig, Programme zu testen, noch lange bevor sie endgültig fertig sind. Gerade für Anfänger ist es wichtig zu sehen, was bestimmte Konstrukte bewirken.

Auch im DEViL-Designer sollte das Testen des Strukturediters schon früh möglich

sein. Ein Test ist immer möglich, so lange alle Konsistenzbedingungen erfüllt sind. Die abstrakte Struktur kann so bereits getestet werden, bevor Muster angewendet wurden.

Role-Expressiveness Ausdrucksfähigkeit besagt, wie dem Benutzer Semantik von Programmkonstrukten vermittelt wird. Dabei wird vor allem auf die in [3] beschriebene "Redundant Recoding" Methode verwiesen, die wichtige Konstrukte durch mehrfache Notation ausdrückt.

Fehlermeldungen wie das falsche Anwenden von Rollen und das Fehlen von Interfaces an Knoten werden sowohl visuell (durch einen roten Rahmen) als auch textuell ausgedrückt. Konkrete und abstrakte Klassen unterscheiden sich sowohl durch ihr Piktogramm als auch durch eine Kursivschrift bei abstrakten Klassen. Es wurde eine textuelle Sicht für die abstrakte Struktur implementiert. Versierte Benutzer können so ebenfalls mit einer textuellen Repräsentation arbeiten. Der Muster-Assistent zeigt neben einer Erläuterung zur aktuellen Rolle auch eine Grafik des Muster-Rollendiagramms an. Der Benutzer kann somit sowohl auf die Struktur des Musters als auch auf die textuelle Beschreibung zurückgreifen

Secondary Notation and Escape from Formalism In herkömmlichen Programmiersprachen drückt der Programmcode oft mehr aus als der implementierte Algorithmus eigentlich verlangt. Einrückungen und Namen von Variablen sowie Kommentare vermitteln dem Leser des Codes schnell einen groben Eindruck.

Im DEViL-Designer wurde durch die Kommentarfunktion Gebrauch einer zweiten Notation gemacht. Anwender können bei der Spezifikation der abstrakten Struktur "Notizzettel" mit Kommentaren einfügen. Einen Einfluss auf die Struktur haben diese Notizzettel nicht. Außerdem wurde die DEViL-Dokumentationsfunktion implementiert, die es erlaubt, die abstrakte Struktur mit Kommentaren zu versehen. DEViL kann dann aus der abstrakten Struktur eine umfangreiche Dokumentation z.B. in HTML generieren. Durch die Wahl des Mengenmusters für die Spezifikation der abstrakten Struktur können inhaltlich zusammenhängende Klassen nah zusammenhängend auf der Zeichenfläche angeordnet werden. Auch die Wahl von Namen der Strukturobjekte wird nicht eingeschränkt.

Viscosity Viscosity beschreibt die Formbarkeit und gibt an, wie schwer es ist, ein existierendes Programm anzupassen. Dieser Punkt ist sehr wichtig, da für das Refactoring in der Praxis viel Zeit anfällt.

Die Umgebung arbeitet auf einem hohen Abstraktionsniveau, was Änderungen prinzipiell erleichtert, da Codestellen nicht aufwändig lokalisiert und geändert werden müssen. Die beiden Sichten zur Spezifikation sind gekoppelt. Eine Änderung der einen Sicht bewirkt sofort die Änderung der zweiten. Die Formbarkeit der Umgebung wird im nächsten Abschnitt noch eingehender untersucht.

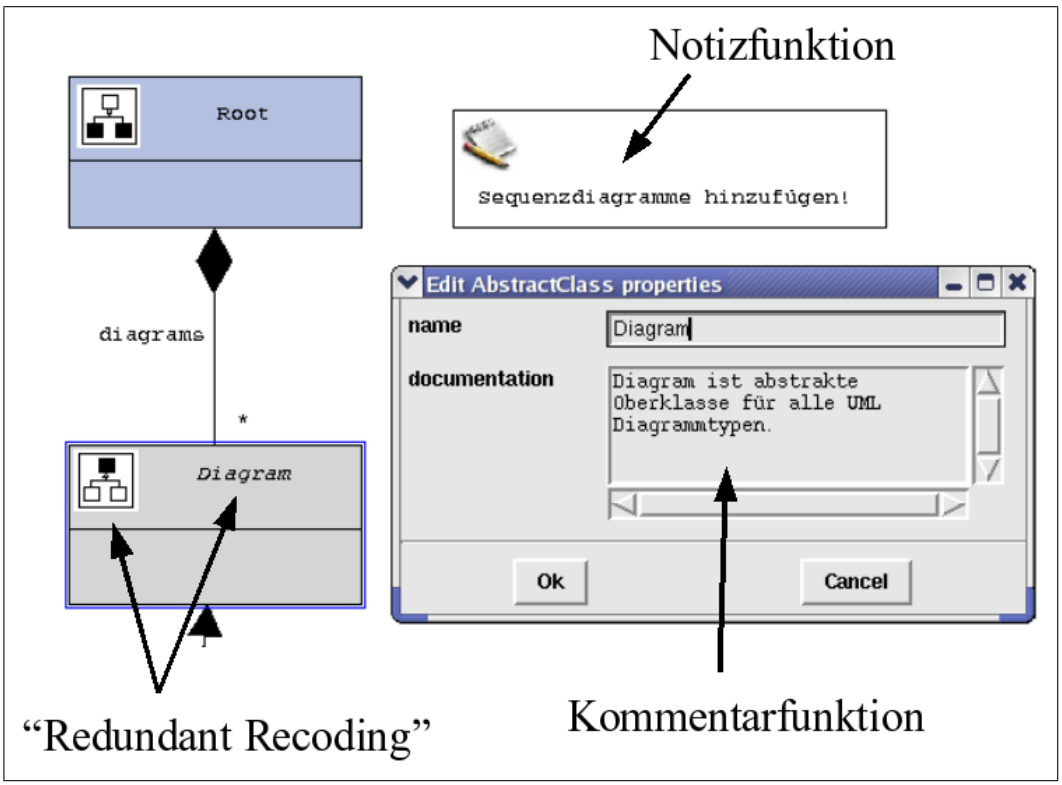

Abbildung 70: Verbesserungen im Dokumentar- und Notizbereich

Fazit Die Evaluierung nach Green und Petre hat gezeigt, dass der DEViL-Designer in weiten Teilen den Anforderungen gewachsen ist. Besonders die Auflösung der abstrakten Struktur in eine Baumstruktur zum Anwenden der visuellen Muster hilft dem Benutzer die Übersicht zu behalten und leitet ihn an, einen bewährten Arbeitsfluss zu benutzen. Inkonsistenzen bei der Spezifikation graphischer Strukturediteren werden unmittelbar durch Fehlermeldungen sichtbar. Schwächen im Bereich der Kommentierung konnten durch neu eingeführte Sprachkonstrukte, wie Notizen und Dokumentation für die abstrakte Struktur behoben werden. Alternative Darstellungen, wie die textuelle Sicht der abstrakten Struktur und die redundante Darstellung von Objekteigenschaften, wurden ebenfalls nachträglich implementiert und zeigen dem Benutzer nun wichtige Informationen mehrfach an (siehe Abb. 70).

6.2 Evaluation durch Fallstudie

In diesem Kapitel sollen die Ergebnisse einer praktischen Fallstudie zum Einsatz des hier entwickelten DEViL-Designers aufgezeigt werden. Die Umgebung wurde dabei von Probanden mit unterschiedlichen Vorkenntnissen in der Entwicklung graphischer Strukturediteren getestet.

Zunächst sollen nun die Ziele der Evaluation und danach die konkreten Ergebnisse

der Evaluation gezeigt werden.

Usability Die hier entwickelte Umgebung soll hinsichtlich ihrer Usability für den Anwender untersucht werden. Usability ist definiert als die Effektivität, Effizienz und Zufriedenheit, mit der der Benutzer eine gegebene Aufgabe erledigt.

Im diesem konkreten Fall bedeutet das, dass untersucht werden soll, wie komfortabel der Benutzer einfache graphische Struktureditoren erstellen kann. Die Zielfragestellungen lauten deshalb:

- Wie einfach lassen sich Editoren für überschaubare Sprachen spezifizieren? Dies ist die wichtigste Fragestellung, die es zu untersuchen gilt. Dabei sollen vor allem kleine Sprachen spezifiziert werden, weil im Rahmen des Entwurfs einer visuellen Sprache häufig kleine Prototypen entwickelt werden. Außerdem wendet sich diese Umgebung eher an einen Anfänger im Umgang mit DEViL.
- Wie einfach lassen sich bestehende graphische Spezifikationen ändern? Die Änderung bestehender Spezifikationen kommt in der Praxis häufig vor, gerade in frühen Phasen der Entwicklung, in der noch viele Sprachvarianten zur Diskussion stehen.
- Wie einfach ist der Entwurf der abstrakten Struktur?
- Wie einfach lassen sich visuelle Muster anwenden?
- Wie schnell lassen sich Spezifikationen erstellen?
- Macht der Benutzer bei der Spezifikation viele Fehler?
- Werden Konzepte zur Softwarestrukturierung benutzt? Dies betrifft vor allem die Spezifikation der abstrakten Struktur. Hier ist der Einsatz von abstrakten Klassen eine Möglichkeit, die Spezifikationen zu strukturieren.
- Traut der Benutzer es sich zu, Struktureditoren auch textuell zu spezifizieren? Dieser Punkt ist vor allem für Anfänger im Umgang mit Generatoren für visuelle Sprachen wichtig. Er soll aufzeigen, ob der Benutzer bereits einschätzen kann, dass er die Konzepte des Generators verstanden hat.
- Wo liegen Schwächen der Umgebung? Gibt es Teile der Umgebung, die unverständlich sind oder gar fehlen?
- Sind Fehlermeldungen verständlich?

Kontrolliertes Experiment Um die oben genannten Fragestellungen hinreichend beantworten zu können, wurde der DEViL-Designer in einem kontrollierten Experiment getestet. Dabei wurden Probanden Aufgaben gestellt, die allein und nur mit wenig Hilfestellung bearbeitet werden sollten. Dieses Experiment sollte zeigen, wie einfach Spezifikationen für Struktureditoren erstellt und abgeändert werden können. Zusätzlich konnte dabei

beobachtet werden, wie das Ziel erreicht wird und wo die Probanden Fehler machten bzw. Hilfe benötigten.

Die Probanden Da die Usability der Umgebung stark von den Vorkenntnissen der Benutzer abhängig ist, war es zunächst wichtig, die Probanden klassifizieren zu können. Dabei ergaben sich zwei Gruppen:

1. Anfänger im Bereich der Spezifikation von graphischen Strukturediteren, die noch nie einen graphischen Strukturediteor spezifiziert und auch keine Vorkenntnisse im Sprachentwurf haben.

2. Fortgeschrittene haben bereits Erfahrungen im Entwurf von Sprachen aber wenig bis keine Erfahrung bei der Anwendung visueller Muster

Diese beiden Gruppen werden bei der Evaluation getrennt untersucht, da sich gezeigt hat, dass es zu unterschiedlichen Problemen kam und eine Gleichbehandlung die Ergebnisse dieser Evaluation verfälschen würde. Die Anfängergruppe bestand aus drei, die Fortgeschrittenengruppe aus zwei Personen. Damit alle Probanden einen Überblick zu visuellen Sprachen, DEViL und der visuellen Spezifikationssprache bekommen, gab es eine kurze Einführung von ca. 30 Minuten. Die Themen dabei waren:

- Visuelle Sprachen: Entwurf auf einem hohen Abstraktionsniveau, Vorteile, Nachteile

- Der Generator DEViL: Konzepte, Spezifikation der abstrakten Struktur, visuelle Muster und ihr Einsatz, Spezifikation von Sichten, generische Zeichnungen, Konsistenzbedingungen beim Einsatz visueller Muster, die Muster *VPForm* und *VP-Connection* als Beispiele

- Einführung in den DEViL-Designer: Spezifikation der abstrakten Struktur, Spezifikation der visuellen Muster

Anschließend mussten die Probanden ihre Kenntnisse einschätzen.

Die Aufgaben Die Probanden mussten drei Aufgaben erledigen: Bei der ersten Aufgabe handelte es sich um die Spezifikation eines einfachen Petri-Netz Strukturediteors, der das Einfügen und Verbinden von beliebig vielen Transitionen und Stellen erlaubt. Auf einer Stelle soll dabei genau eine Marke liegen. Konsistenzbedingungen für den Petri-Netz Editor gab es keine. Der Petri-Netz Editor realisiert eine sehr einfache Sprache, was sich in den Codezeilen der Originalspezifikation widerspiegelt: 28 Zeilen für die abstrakte Struktur und 36 Zeilen für die visuellen Muster. Bei den Codezeilen werden Leerzeilen nicht berücksichtigt. Die abstrakte Struktur liegt dabei in der Form

```
1. CLASS Root {
2.  name: VAL VLString;
3.  classes: SUB Class*;
4. }
5. ...
```

vor. Spezifikationen der Sicht liegen im folgenden Format vor:

```
1. SYMBOL rootView_Root INHERITS VPForm, VPRootElement
2. COMPUTE
3.  SYNT.drawing = ADDROF(RootDrawing);
4. END;
5. ...
```

Bei der Spezifikation der abstrakten Struktur gibt es einen interessanten Aspekt: die Marke kann entweder zeichnerisch in die Stelle eingefügt sein oder als eigenständiges Objekt realisiert werden, was dazu führt, dass beliebig viele Marken eingefügt werden können. Den Probanden wurde es selbst überlassen, welche Variante sie modellieren wollten.

Die zweite Aufgabe realisiert einen einfachen UML Klassendiagramm Strukturediteor, der das Einfügen von Interfaces, konkreten und abstrakten Klassen ermöglicht. Klassen und Interfaces haben dabei zusätzlich Operationen, die durch ihren Namen spezifiziert werden sollen. Klassen haben Attribute, die einen Namen und einen Wert haben sollen. Zwischen allen Strukturobjekten kann es Verbindungen geben. Aggregationen und Kompositionen, wie es sie in UML gibt, sollen nicht berücksichtigt werden. Der Klassendiagramm Strukturediteor besitzt mit 47 Codezeilen eine etwas komplexere abstrakte Struktur, die Sichtspezifikation ist mit 158 Zeilen deshalb auch etwas aufwändiger. Eine Besonderheit ist hier ebenfalls die Spezifikation der abstrakten Struktur. Hier kann durch geschickten Einsatz von abstrakten Klassen die UML Attribute bzw. UML Operationen des UML Editors an Oberklassen delegiert werden.

Die dritte Aufgabe soll zeigen, wie einfach es ist, eine bestehende Spezifikation zu ändern. Dabei sollte der in der ersten Aufgabe entwickelte Petri-Netz Editor so abgeändert werden, dass er auch Stellen mit zwei oder drei Marken unterstützt. Hier konnten die Probanden entweder einfach zwei zusätzliche Klassen, die als Stellen mit zwei bzw. drei Marken dargestellt werden spezifizieren, oder die Marken als eigene Strukturobjekte einführen.

Bei den Aufgaben sollte zunächst die abstrakte Struktur spezifiziert werden. Dabei sollte insbesondere auf die Verwendung von abstrakten Klassen geachtet werden. Danach sollte die Sichtspezifikation entworfen werden. Zum Schluss sollte der Strukturediteor generiert und getestet werden.

Die Ergebnisse Vor Bearbeitung der Aufgaben wurde den Probanden ein Fragebogen gegeben, in dem sie ihre DEViL Vorkenntnisse einschätzen mussten. Die Fragen konnten durch Ankreuzen auf einer 5-stufigen Likert-Skala beantwortet werden, bei der sich zwei gegensätzliche Extremaussagen gegenüberstehen. Die linke Seite wird mit einem Punkt, die rechte Seite mit 5 Punkten bewertet. Die Fragen zielen auf Konzepte des DEViL Generators, es soll nicht darum gehen, wer eine Spezifikation syntaktisch korrekt

	ØAnfänger	ØFortgeschrittene
1. Ich verstehe die Sprache zur Spezifikation der abstrakten Struktur zu ...(1) 0% ...(5) 100%	2,7	4,5
2. Ich verstehe das Konzept der generischen Zeichnungen zu ...(1) 0% ...(5) 100%	2,7	3,5
3. Ich verstehe das Konzept der visuellen Muster zu ...(1) 0% ...(5) 100%	2,3	3,5
4. Ich verstehe das Konzept der Grammatikabbildungen zu ...(1) 0% ...(5) 100%	1,0	3,0
5. Ich verstehe das Konzept der Sichtspezifikation zu ...(1) 0% ...(5) 100%	2,3	3,0
6. Ich verstehe DEViL insgesamt zu ...(1) 0% ...(5) 100%	2,7	4,0
7. Bezüglich der Spezifikation von graphischen Strukturediteren bin ich ...(1) Anfänger ...(5) Experte	1,0	3,5

Abbildung 71: Einschätzung der DEViL Vorkenntnisse

	ØAnfänger	ØFortgeschrittene
Petri-Netz Editor: abstrakte Struktur	7,7	8
Petri-Netz Editor: visuelle Muster	14,3	9,5
UML: abstrakte Struktur	11,7	13,0
UML: visuelle Muster	18,0	13,5
Spezifikationsänderung: abstrakte Struktur	2,0	2,0
Spezifikationsänderung: visuelle Muster	4,0	3,0

Abbildung 72: Zeitverbrauch in Minuten bei der Bearbeitung der Aufgaben

aufschreiben kann.

An der Tabelle aus Abbildung 71 zeigt sich, dass die beiden Gruppen vor allem bei der Spezifikation der abstrakten Struktur und der Grammatikabbildung auseinander liegen. Die Fortgeschrittenengruppe schätzt sich hier deutlich stärker ein, da bereits Erfahrungen im Sprachentwurf vorliegen. Die Werte bei Frage vier sind zwischen den Gruppen nicht vergleichbar. Der Anfängergruppe wurde die Grammatikabbildung nicht erklärt, da der DEViL-Designer diese Arbeit automatisch übernimmt. Interessant ist die recht große Differenz zwischen den beiden Gruppen bei Frage sechs und sieben. Die Anfänger schätzen sich hier sehr schlecht ein, wahrscheinlich weil ihnen der Gesamtüberblick zu DEViL fehlt. Die Tabelle aus Abbildung 72 zeigt, dass die Fortgeschrittenengruppe sich etwas länger mit der Modellierung der abstrakten Struktur aufhält. Allerdings konnte diese Gruppe abstrakte Klassen sofort identifizieren, während die Anfängergruppe recht chaotisch an die Modellierung heranging und besonders bei der Bearbeitung der ersten Aufgabe einige Hilfestellungen benötigte. Der Anfängergruppe

war zunächst nicht klar, wozu die Root Klasse benötigt wird und dass von konkreten Klassen nicht geerbt werden darf. Die Spezifikation der abstrakten Struktur bei der zweiten Aufgabe gelang dann allen Probanden bereits eigenständig. Auch konnte ein Proband der Fortgeschrittenengruppe die Marken des Petri-Netz Editors als eigenständige Strukturobjekte identifizieren. Die letzte Aufgabe entfiel somit für ihn. Die Fortgeschrittenen konnten sofort identifizieren, an welche abstrakten Oberklassen Attribute und Operationen des UML Klassendiagrameditors gebunden werden mussten. Hier zeigte sich, dass die Fortgeschrittenengruppe mit Erfahrungen im Sprachentwurf besonnener an die Arbeit heranging. Interessant war auch die Herangehensweise bei der Spezifikation der abstrakten Struktur: so modellierte ein Proband zunächst alle Strukturobjekte und erst danach die Beziehungen dieser untereinander. Die restlichen Probanden spezifizierten die Struktur von einem Startobjekt (in der Regel die Rootklasse) ausgehend. Ein eindeutiger Vor- oder Nachteil dieser beiden Herangehensweisen kann nicht aufgezeigt werden, es ist jedoch von Vorteil, dass der entwickelte DEViL-Designer beide Verfahren gleich gut unterstützt.

Das Anwenden der visuellen Muster auf die Strukturobjekte fiel den Anfängern deutlich schwerer. Vor allem komplexe Muster wie das *VPConnection* Muster konnten nicht sofort angewendet werden, es wurde beklagt, dass ein Überblick zu allen vorhandenen Mustern und was diese leisten fehlt. Auch wurden Rollen bei komplexen Mustern vergessen anzuwenden. Dies führte dazu, dass in die Umgebung nachträglich ein Muster-Assistent (siehe Abschnitt 4.4.2 auf Seite 47) integriert wurde, der die Rollen eines Musters Schritt für Schritt anwendet. Das Verständnis des Muster-Konzepts an sich scheint eher ein konzeptionelles Problem zu sein, das mit einer ausführlicheren Einführung und einer Liste mit Mustern und deren Aufgaben hätte gelöst werden können. Die Tabelle aus Abbildung 73 zeigt, dass alle Probanden die Aufgaben verstanden hatten. Die Anfängergruppe schätzt die ersten beiden Spezifikationsaufgaben jedoch schwerer ein. Erwähnenswert ist, dass die zweite Aufgabe um fast einen Punkt leichter eingestuft wird, obwohl die Aufgabe vom Spezifikationsaufwand und bei der Identifikation von abstrakten Klassen eindeutig komplexer ist. Dies zeigt, dass sich die Probanden aus der Anfängergruppe bereits nach einer kleinen Aufgabe gut in das System eingearbeitet haben.

Die Änderung von bestehenden Spezifikationen wird durchweg als einfach eingestuft, jedoch mit der Einschränkung, dass es sich um geringe Änderungen bei verhältnismäßig kleinen Strukturediteren handelt. In der Tabelle aus Abbildung 74 ist zu sehen, dass die Spezifikation der abstrakten Struktur insgesamt als leicht eingestuft wird. Ebenso einfach werden die Benutzung von generischen Zeichnungen und das Verständnis zu Inkonsistenzen eingestuft. Eine Diskrepanz gibt es bei der Einschätzung der Sichtspezifikationen. Die Anfänger stufen diese als deutlich schwerer ein, was wohl daran liegt, dass die Fortgeschrittenengruppe sich mehr im Klaren über den Nutzen und das Anwenden von visuellen Mustern ist. Dies wurde auch bei den Gesprächen mit den Probanden während der Evaluation deutlich. Positiv zu bewerten ist, dass die Geschwindigkeit der Umgebung als durchaus akzeptabel eingestuft wird. Die Geschwindigkeitsoptimierungen haben hier

	ØAnfänger	ØFortgeschrittene
Petri-Netz Editor: Die Aufgabenstellung war ...(1) leicht ...(5) schwer verständlich	1,3	1,0
Petri-Netz Editor: Die Aufgabe war ...(1) einfach ...(5) schwer	3,3	1,5
Petri-Netz Editor: Die Bearbeitung der Aufgabe empfand ich als ...(1) angenehm ...(5) unangenehm	2,3	1,5
UML Editor: Die Aufgabenstellung war ...(1) leicht ...(5) schwer verständlich	2,0	1,0
UML Editor: Die Aufgabe war ...(1) einfach ...(5) schwer	2,6	1,5
UML Editor: Die Bearbeitung der Aufgabe empfand ich als ...(1) angenehm ...(5) unangenehm	2,3	1,5
Spezifikationsänderung: Die Aufgabenstellung war ...(1) leicht ...(5) schwer verständlich	1,6	2,0
Spezifikationsänderung: Die Aufgabe war ...(1) einfach ...(5) schwer	1,3	2,0
Spezifikationsänderung: Die Bearbeitung der Aufgabe empfand ich als ...(1) angenehm ...(5) unangenehm	2,3	2,0
Spezifikationsänderung: Bestehende Spezifikationen sind ...(1) leicht ...(5) schwer zu ändern	2,0	2,0

Abbildung 73: Einschätzung der Aufgaben

	ØAnfänger	ØFortgeschrittene
Die Sicht zur Spezifikation der abstrakten Struktur empfinde ich als ...(1) leicht ...(5) schwer verständlich	1,6	1,0
Die Sicht zur Spezifikation der visuellen Muster empfinde ich als ...(1) leicht ...(5) schwer verständlich	3,3	1,5
Kontrollattribute wie generische Zeichnungen sind ...(1) einfach ...(5) schwer anzuwenden	1,3	1,0
Meldungen zu Inkonsistenzen sind ...(1) leicht ...(5) schwer verständlich	2,6	2,0
Die Geschwindigkeit der Umgebung empfinde ich als ...(1) sehr schnell ...(5) sehr langsam	1,6	2,5

Abbildung 74: Einschätzung der Bedienung der Umgebung

	ØAnfänger	ØFortgeschrittene
Die Spezifikation der abstrakten Struktur auf textueller Ebene würde ich mir ...(1) zutrauen ...(5) nicht zutrauen	1,6	1,0
Die Spezifikation der visuellen Muster auf textueller Ebene würde ich mir ...(1) zutrauen ...(5) nicht zutrauen	3,3	1,5

Abbildung 75: Einschätzung der DEViL Konzepte

also gewirkt.

Fazit Die Evaluierung hat gezeigt, dass es der DEViL-Designer auch Anfängern ermöglicht, kleine Strukturediteren zu erstellen. Mit einer Einführung in das Thema und einigen Hilfestellungen ist es Anfängern möglich, auch eigenständig Strukturediteren zu spezifizieren. Die Umgebung hat sich dabei als stabile Entwicklungsplattform erwiesen. Fortgeschrittene Anwender, die bisher nur an der Codegenerierung gearbeitet haben, können die Konzepte der Sichtspezifikation relativ schnell erlernen. Gerade die visuelle Sichtspezifikation bietet zudem einen Geschwindigkeitsvorteil zur herkömmlichen textuellen Variante. Spezielle Konzepte der visuellen Muster, die besonders Anfängern Probleme machten, können durch den nachträglich implementierten Muster-Assistenten leichter erlernt werden. Die Tabelle aus Abbildung 75 zeigt, dass sich auch Anfänger nach der Spezifikation einiger Strukturediteren mit dem DEViL-Designer durchaus trauen würden, mit DEViL auf textueller Ebene zu arbeiten. Die Umgebung vermittelt also die wichtigsten Konzepte von DEViL auf visueller Ebene.

7 Resümee

In dieser Arbeit wurde ein visuelles Frontend, der DEViL-Designer, für den Generator DEViL entwickelt. Dabei wurden diverse Teilsprachen entworfen, die es erlauben, graphische Strukturediteren visuell zu erstellen. Der Spezifikationsprozess für den Benutzer sollte sowohl einfach als auch intuitiv sein. Eine Spezifikation von inkonsistenten Strukturediteren wurde durch umfangreiche Konsistenzchecks verhindert. Diverse Programm-Assistenten sollten den Anwender bei der Entwicklung unterstützen, auch sollte der DEViL-Designer keine Einschränkungen hinsichtlich des Sprachtyps an den Benutzer stellen. Das Spektrum der visuellen Sprachen, die entworfen werden können, sollte so breit wie möglich sein. Trotzdem habe ich mich auf kleine visuelle Sprachen konzentriert, da die späteren Benutzer des DEViL-Designers hauptsächlich Anfänger sein werden, denen Konzepte des Generators vermittelt werden sollen.

Der Hauptteil dieser Arbeit war die Entwicklung einer visuellen Sprache, die die textuellen DEViL-Spezifikationssprachen für das Modell, die Sichten und die Codegenerierung darstellt. Dazu musste ich zunächst geeignete Repräsentationen finden, die einfach, übersichtlich und intuitiv die jeweiligen Konzepte widerspiegeln. Die Schwierigkeit dabei war, dass es für den Entwurf visueller Sprachen keine vorgefertigten “Rezepte“ gibt. Deshalb habe ich sowohl bestehende Generatoren als auch bereits in der Praxis eingesetzte visuelle Sprachen auf ihre Repräsentationskonzepte hin untersucht. Für die Spezifikation des Modells ergab sich so eine UML-nahe visuelle Sprache. Die Spezifikation von Sichten und Codegenerierung erfolgte auf Basis einer Baumstruktur, die den Arbeitsfluss des Benutzers unterstützt und den DEViL-internen abstrakten Strukturbaum am Besten darstellt. Dazu musste mit Hilfe des DEViL-Kopplungsmechanismus eine Konsistenz zwischen den Sichten hergestellt werden.

Durch die Implementierung einiger Sprachen konnte ich die praktische Anwendbarkeit des DEViL-Designers zeigen. Ich habe dabei strukturell unterschiedliche Sprachen betrachtet, die sowohl einen hohen textuellen als auch einen hohen graphischen Anteil hatten. Durch ein formales Evaluationsmodell habe ich den DEViL-Designer auf Usability untersucht und verbessert, was dazu führte, dass ich im Bereich der Kommentierung neue Sprachkonstrukte hinzugefügt habe. In einem kontrollierten Experiment musste sich der DEViL-Designer in der Praxis bewähren. Hier zeigte sich, dass der DEViL-Designer einige zusätzliche Programm-Assistenten benötigte, die dem Anwender bei der Spezifikation graphischer Strukturediteren helfend unterstützten. Ich konnte zeigen, dass auch Probanden, mit wenig Vorkenntnissen im Bereich visueller Sprachen, schnell kleine graphische Strukturediteren implementieren können und dass Konzepte, wie Modell und Muster, gut vermittelt werden.

Abbildungsverzeichnis

8 Glossar

abgeleitetes Modell Das abgeleitete Modell ist ein Modell, dass mit einem Basismodell gekoppelt ist. Abgeleitetes Modell und Basismodell können dabei vollkommen andere interne Strukturen besitzen. Es gilt in der Regel jedoch immer, dass Instanzen von Klassen im abgeleiteten Modell auf Instanzen von Klassen im Basismodell referenzieren. Ist diese Referenz nicht vorhanden, so wird das Objekt im abgeleiteten Modell gelöscht. Ändern sich gekoppelte Attribute im Basismodell oder im abgeleiteten Modell, so werden auch die Attribute im gekoppelten Modell bzw. im Basismodell geändert.

abstrakte Struktur Die abstrakte Struktur ist eine Ausprägung des Modells, die klassenbasiert entworfen wird. In Kapitel 4 auf Seite 7 ist eine abstrakte Struktur für einen Statechart-Editor zu sehen.

Basismodell Mit dem Basismodell oder auch Modell können abstrakte Strukturen spezifiziert werden. Das Basismodell wird üblicherweise immer mit einem gekoppelten abgeleiteten Modell in Verbindung gebracht.

CASE Tools Computer-Aided Software Engineering (CASE) steht für die computerunterstützte Softwareentwicklung. Spezielle Software, wie UML-Editoren, helfen, den Softwareentwicklungsprozess zu visualisieren. Oft sind CASE Tools in Entwicklungsumgebungen eingebettet.

CONSTITUENTS/INCLUDING/SHIELD Dies sind LIDO Sprachkonstrukte, um im Strukturbaum auf entfernte Attribute zugreifen zu können. Mittels CONSTITUENTS kann auf Symbole die sich in Unterbäumen befinden zugegriffen werden. Dabei werden alle Objekte eines bestimmten Typs, die sich im Strukturbaum unterhalb befinden, erreicht. Dies kann mittels einer SHIELD Klausel verhindert werden. Mit INCLUDING kann auf entfernte Attribute, die sich im Strukturbaum oberhalb befinden, zugegriffen werden.

Container Ein Container ist Teil einer generischen Zeichnung. In Container können weitere graphische Repräsentationen von Strukturobjekten geschachtelt werden, z.B. können Listenelemente innerhalb eines Containers der graphischen Struktur liegen.

funktionale Programmierung Bei der funktionalen Programmierung werden Berechnungen unveränderbar an ein Attribut gebunden. Rekursion, Funktionen höherer Ordnung und frei kombinierbare Datentypen sind möglich.

generische Zeichnung Eine generische Zeichnung stellt mittels benutzerdefinierten Primitiven, wie Rechtecken, Kreisen, Linien oder Text, bestimmte Strukturobjekte dar. In generische Zeichnungen können Container eingebettet werden, die je nach Platzbedarf

von Unterobjekten gedehnt werden können. Eine generische Zeichnung kann als spezielles Kontrollattribut des Formular-Musters aufgefasst werden.

Grammatikabbildung Ein Modell wird in DEViL in eine Grammatik übersetzt, die dann von Eli weiterverarbeitet wird. Diese Grammatik kann vom Anwender verändert werden, indem Grammatiksymbole hinzugefügt oder weggelassen werden können.

Hypergraph Ein Hypergraph ist ein spezieller Graph, bei dem Kanten mehrere Knoten verbinden dürfen. Hypergraphen werden in Graphersetzungssystemen wie DiaGen eingesetzt. Mit einem Hypergraph werden Layout und räumliche Beziehungen zwischen Strukturobjekten berechnet. Ein Hypergraph wird i. d. R. vom Strukturbaum separat betrachtet, um ein geeignetes Austauschformat für die Grafikroutinen zu erreichen. Knoten sind graphische Objekte, wie Rechtecke, Pfeile oder Linien. Als Punkte werden üblicherweise so genannte “Attachmentareas“ gekennzeichnet. Dies sind Bereiche an denen andere graphische Objekte angebunden werden können. Kanten stellen Lagebeziehungen (wie “berührt“ oder “über“) oder Enthaltenseinbeziehungen dar. Attribute (abgerundete Rechtecke) speichern das Aussehen von Objekten oder Texten. Charakteristisch ist für Hypergraphen, dass Kanten auch mehrere Knoten verbinden können.

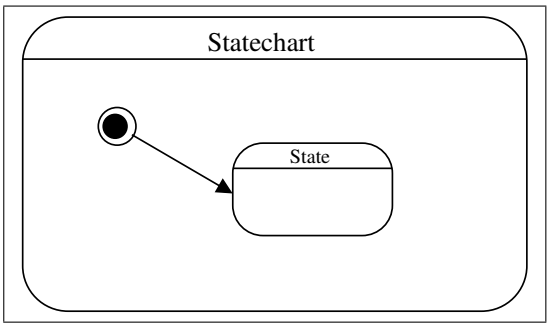

Abbildung 76: Statechartdiagramm

imperative Programmierung Bei der imperativen Programmierung wird das Programm als eine Folge von Befehlen angesehen. Es gibt eine Zuweisungssemantik.

Kontrollattribut Ein Kontrollattribut ist Teil einer Rolle eines Musters. Mit einem Kontrollattribut kann der Benutzer das Aussehen bzw. Layout eines Musters kontrollieren. Kontrollattribute werden in der Musterspezifikationssicht angelegt und können dann in der Sicht zum Anwenden visueller Muster modifiziert werden.

Kopplung Bei der Kopplung werden Teile des Modells aneinander gekoppelt. Das sogenannte abgeleitete Modell wird dabei durch Synchronisationsfunktionen an ein Basismodell gebunden. Änderungen an einer der Strukturen wirken sich auch auf das Pendant der anderen Struktur aus. Modell und abgeleitetes Modell sind dabei strukturell nicht

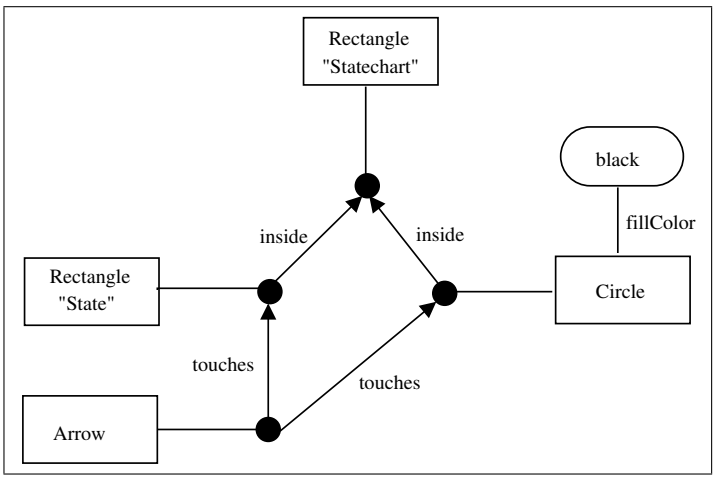

Abbildung 77: Hypergraph für das Statechartdiagramm

zwingend gleichartig, sondern können vollkommen verschiedene Strukturen implementieren. Es gilt dabei jedoch immer, dass ein Strukturobjekt des abgeleiteten Modells ein Strukturobjekt des Basismodells referenziert. Wird das Strukturobjekt in der Basisstruktur gelöscht, so wird auch das Objekt in der abgeleiteten Struktur entfernt (mehr zur Kopplung im DEViL-Designer in Kapitel 5.1 auf Seite 64).

LIDO Funktionale Sprache zur Spezifikation von Sichten in DEViL, mit der Attributberechnungen in Strukturbäumen ausgeführt werden. LIDO Wird in dieser Arbeit durch die Sicht zur Anwendung visueller Muster beschrieben. Bei der Codegenerierung liegen die Sichtberechnungen in Dateien mit der Endung **.lido*.

Modell Das Modell ist die Sprache in der abstrakte Strukturen spezifiziert werden können.

Muster Ein Muster kapselt Berechnungen, die Strukturobjekte visuell darstellen. Es besteht in der Regel aus mehreren Rollen, die in Rollendiagrammen spezifiziert werden. Kontrollattribute können das Layout und weitere Mustereigenschaften beeinflussen. Häufig benutzte Muster sind: VPForm, VPSimpleList, VPSet, VPFlowList.

PTG PTG (Pattern-based Text Generator) wird bei der DEViL-Codegenerierung benutzt, um Textfragmente mit Variablen und PTGNodes zu kombinieren. Dabei generiert PTG Funktionen, die innerhalb eines LIDO Codegenerierungsmoduls aufgerufen werden können, um schließlich in eine beliebige Zielsprache übersetzt zu werden.

Rolle Eine Rolle ist Teil eines Musters. Sie stellt eine Eigenschaft eines Musters dar, z.B. ist VPSetElement ein Mengenelement eines Mengenmusters. Rollen stellen Interfaces zur Verfügung (grün hinterlegter Teil in Rollendiagrammen), benötigen i.d.R. aber

auch Interfaces (rot hinterlegter Teil in Rollendiagrammen) und müssen deshalb mit anderen Rollen eines anderen Musters kombiniert werden.

Rollendiagramm Rollendiagramme stellen die Struktur von Mustern graphisch aufbereitet dar. Rollendiagramme werden in einer Baumstruktur dargestellt, die die spätere Beziehung der Rollen im Strukturbaum verdeutlichen soll. Wenn eine Rolle einer anderen Rolle im Rollendiagramm untergeordnet ist, so muss diese auch bei der Musteranwendung im Strukturbaum tiefer hängen. Im rot hinterlegten Bereich der Rollendiagramme werden benötigte Interfaces, im grünen werden bereitgestellte Interfaces angezeigt. Zusätzlich kommen in der DEViL-Designer Musterspezifikationssicht noch zwei weitere Teilbereiche hinzu: blau hinterlegt sind Kontrollattribute, die vom Benutzer verändert werden können. Gelb hinterlegt sind Kontrollattribute, die zusätzlich in das Modell mit hineinkopiert werden. Diese Attribute werden von der Rolle nur intern benutzt.

Sicht Eine Sicht beschreibt eine konsistente Musterkombination, die an Teile der abstrakten Struktur gebunden ist. Eine Sicht stellt einen Teil einer Sprache visuell dar. In DEViL werden Sichtberechnungen in LIDO Dateien gespeichert. Mit dem DEViL-Designer ist es möglich, Sichten visuell zu spezifizieren.

Strukturediteor Ein Strukturediteor stellt ein abstraktes Konzept graphisch oder textuell dar. Mit ihm lassen sich Teile des abstrakten Strukturbaums in einer mehrdimensionalen Darstellung manipulieren. Eine bekannte Suite von Strukturediteoren stellen UML-Softwareentwicklungs-Werkzeuge wie Together [23] dar.

Strukturobjekt Ein Strukturobjekt ist ein Objekt im abstrakten Strukturbaum. Es ist die Instanz einer Klasse des Modells.

visuelle Sprache Visuelle Sprachen werden u.a. in der Softwareentwicklung eingesetzt und stellen Strukturen mehrdimensional dar. Prägnante graphische Notationen erlauben den leichteren Zugang zu komplexen Problemstellungen. (siehe auch Kapitel 3.1 auf Seite 4

Literatur

[1] Mark Meyer, R. et al.: Towards a better visual programming language: critiquing Prograph's control structures, 2000

[2] Green, T.R.G. et al.: Usability Analysis of Visual Programming Environments: A "Congnitive Dimensions" Framework, 1996

[3] Fitter, M.J. et al.: When do Diagrams Make Good Computer Languages, 1979

[4] Green, T.R.G. et al.: When Visual Programs are Harder to Read than Textual Programs, 1992

[5] Glinert, E.P.: Nontextual programming environments. In Chang, S.-K. (Ed.) Principles of Visual Programming Systems. Prentice Hall, 1990

[6] Green, T.R.G. et al.: The art of Notation. In: Computing Skills and the User Interface. (M.J. Coombs, J. Alty, eds). Academic Press, London, pp 221-251, 1982

[7] J.H. Larkin et al.: Why a diagram is (sometimes) worth 10000 words. Cognitive Science 11, 65-100, 1987

[8] Petre, M. et al.: Where to draw the line with text: some claims by logic designers about graphics in notation. In: Human-Computer-Interaction - INTERACT'90 (D.Diaper, D. Gilmore, G.Cockton , B.Shackel, eds) Elsevier, Amsterdam, pp. 463-468, 1990

[9] Jeckle, M. et al.: UML 2 glasklar, Hanser Verlag, 2004

[10] Kastens, U. et al.: The Eli System. In Kai Koskimies, Hrsg., Proceedings of 7th International Conference on Compiler Construction CC'98, Nummer 1383 in Lecture Notes in Computer Science, S. 294-297. Springer Verlag, März 1998.

[11] Kemper et al.: Datenbanksysteme - eine Einführung 5., aktualisierte und erweiterte Auflage, Oldenbourg-Verlag, 2004.

[12] Schmidt, C. et al.: Implementation of visual languages using pattern-based specifications. Software - Practice and Experience, 33:1471-1505, Dezember 2003.

[13] Schwindt, E.: Ein graphischer Editor für Generische Zeichnungen. In Fachwissenschaftlicher Informatikkongress - Informatiktage 2002, Bad Schussenried, November 2003.

[14] Raines, P. et al.: TCL/TK in a Nutshell, O'Reilly, 1999

[15] Bardohl, R.: GenGed: A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars, Paper, 1998

[16] MetaEdit+: Domain-Specific Modeling, http://www.metacase.com, 27.07.2005

[17] Minas, M. et al.: Generating Diagram Editors with DiaGen, 1999

[18] Minas, M.: Bootstrapping Visual Components of the DiaGen Specification Tool with DiaGen, in Proc. International Workshop on Applications of Graph Transformations with Industrial Relevance, Charlottesville, Virginia, USA, September 28 - October 1, 2003, pages 391–405, 2003.

[19] Streets: http://ag-kastens.uni-paderborn.de/forschung/vl-eli/beispiele/streets/index.php, 2005

[20] Griebel, P.: Parcon - Paralleles Lösen von grafischen Constraints. PhD thesis, Paderborn University, February 1996

[21] Marriott, K. et al.: A Tableau Based Constraint Solving Toolkit for Interactive Graphical Application 4th International Conference on Principles and Practice of Constraint Programming, Pisa, Italy, 26th-30th October, 1998.

[22] Schürr, A.: Visuelle Programmiersprachen, http://www2-data.informatik.unibw-muenchen.de/Lectures/WT2001/VL/VL.html, 2001

[23] Borland Togethersoft: Together, http://www.borland.com/us/products/together/index.html, 2005

Anlage

Auf der beigefügten CD–ROM sind die Ergebnisse dieser Arbeit gespeichert.