DEViL

Omar Bouraga
223 320

Betreut von Dipl.-Inf. Ulrike Ranger

Zusammenfassung

Visuelle Entwicklungsumgebungen spielen eine wichtige Rolle bei der Modellierung von Softwaresystemen. Die Implementierung von visuellen Entwicklungsumgebungen setzt konzeptionelles und technisches Wissen in einem breiten Umfang voraus. Das DEViL-System generiert Entwicklungsumgebungen für visuelle Sprachen. In dieser Ausarbeitung wird ein Überblick des Spezifikationsmechanismus zur visuellen Sprachumgebung gegeben. Diese Spezifikation dient zur Beschreibung der abstrakten Syntax, der visuellen Darstellung von Diagrammen und der Textausgabe. Zur Spezifikation der visuellen Darstellungen werden visuelle Muster eingesetzt, die eine höhere Abstraktionsebene definieren.

Inhaltsverzeichnis

1 Motivation

Visuelle Sprachen dienen zur Modellierung von betrieblichen Anwendungssystemen. Im Gegensatz zur textuellen Notation können die Elemente der visuellen Sprachen von Menschen besser verarbeitet werden. Daher werden visuelle Sprachen in der Praxis sehr häufig eingesetzt. Sie stellen die Zusammenhänge zwischen Objekten besser dar als textuelle Sprachen. Beispielsweise können die Vorfahren einer Familie als ein Baum dargestellt werden. Diese Darstellung macht den Lesern die Erkennung der Vorfahren einfacher als bei einer textuellen Notation.
In dieser Seminararbeit wird der Generator DEViL vorgestellt. Mit Hilfe dieses Generators ist es möglich, visuelle Sprachumgebungen aus Spezifikationen zu generieren.

1.1 UML

UML (Unified Modeling Language) [1] ist eine der bekanntesten und häufig verwendeten visuellen Sprachen für die Modellierung von Softwaresystemen. Sie wird zur Modellierung von Anforderungen, Design und Implementierung von Softwaresystemen eingesetzt, die auf objektorientierten Sprachen wie *Java* beruhen. Zwei der wichtigsten Diagramme in UML sind die Klassen- und Zustandsdiagramme:

- Klassendiagramme beschreiben die Struktur eines Systems und stellen die Beziehungen zwischen Objekten dar. Die Klassendiagramme zeigen die Operationen und die Eigenschaft von Klassen beispielsweise die Funktionen einer Klasse und die Beziehung zwischen einer Klasse und einer anderen Klasse wie die Vererbung. Die zentralen Konstrukte bei der Modellierung sind Klassen, Attribute, Methoden und Vererbungs- und Assoziationsbeziehungen. Die Klassen werden im Klassendiagramm als Rechtecke repräsentiert, die jeweils den Namen der Klasse, die Attribute und Methoden enthalten. Die Verbindung zwischen den Klassen wird durch Assoziationen dargestellt, wobei eine Assoziation von einer Ausgangsklasse zur Zielklasse zeigt. Dabei wird ebenfalls die Kardinalität zwischen den beiden Klassen notiert. Als Kardinalitäten zwischen Klassen kommen nach [1] drei Fälle in Frage:

    1. 1: Jedem Objekt einer Klasse wird genau ein Objekt zugeordnet.

2. 0..1: Jedem Objekt einer Klasse wird entweder ein oder kein Objekt zugeordnet.
3. *: Jedem Objekt einer Klasse werden entweder beliebig viele Objekte zugeordnet oder kein einziges.

- In Zustandsdiagrammen wird das Verhalten von einem System in Form von endlichen Automaten dargestellt. Die wichtigsten Konstrukte bei der Modellierung sind Zustände und Transitionen. Ein Zustandsdiagramm hat einen Anfangszustand, eine endliche Anzahl von Zuständen, eine endliche Anzahl von Transitionen, die den Übergang von einem Zustand in den anderen Zustand repräsentieren, und kann mehrere Endzustände haben. Die Zustände werden in Form von abgerundeten Rechtecken dargestellt. Diese Zustände sind über gerichtete Transitionen verbunden. Der Anfangzustand im Diagramm wird durch einen gefüllten Kreis dargestellt, während die Endzustände durch Doppelkreise mit gefülltem Inhalt repräsentiert werden [1]. In einem Diagramm können mehrere Zustände und Transitionen gemeinsames Verhalten haben. In diesem Fall werden diese Zustände als Unterzustände bezeichnet und in einem Oberzustand zusammengefasst, wobei die Transitionen reduziert werden können.

Beispiel
Abbildung 1 stellt ein Beispiel für die *Steuerung eines Tischventilators* in einem UML-Zustandsdiagramm dar. Der Ventilator ist am Anfang ausgeschaltet. Daher zeigt der Anfangszustand auf den Zustand, in dem der Ventilator ausgeschaltet ist. Um den Ventilator einzuschalten, gibt es drei Möglichkeiten bzw. drei Knöpfe um die Geschwindigkeitsstufe zu regeln. Diese drei Möglichkeiten bzw. Zustände werden in einem Oberzustand *Ventilator eingeschaltet* zusammengefasst. Die Transitionen zwischen den einzelnen Stufen bedeuten, dass innerhalb des Oberzustands von Zustand zu Zustand gewechselt werden kann. Mit der Transition *Knopf Stop drücken* wird vom Oberzustand *Tischventilator eingeschaltet* wieder in den Zustand *Tischventilator ausgeschaltet* gewechselt.
Im Folgenden wird ein durchgängiges Beispiel von einem Zustandsdiagramm dargestellt, das die Steuerung eines Tischventilators modelliert.

1.2 Editor für ein Zustandsdiagramm

Um Entwicklungsumgebungen für visuelle Sprachen zu generieren, werden Generatoren wie DEViL (Development Environment for Visual Languages) [4] ein-

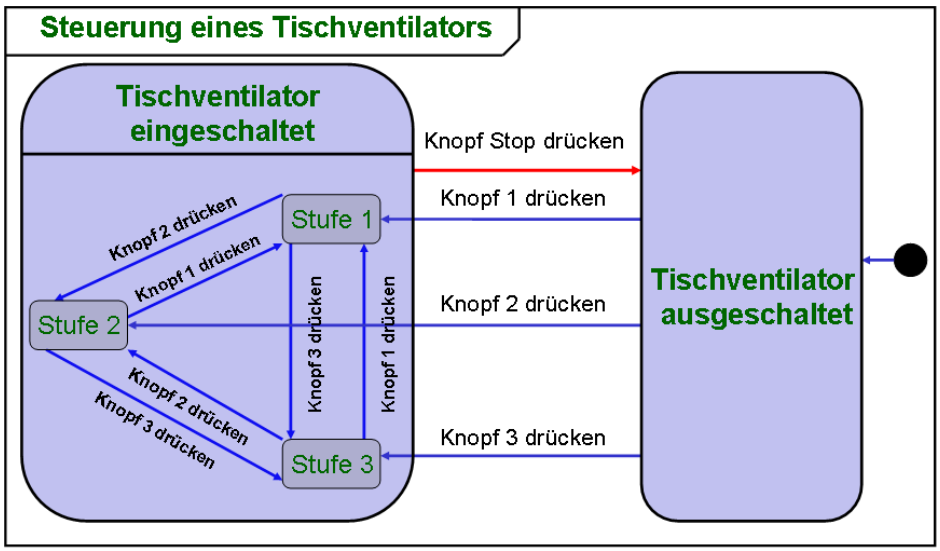

Abbildung 1: Steuerung des Ventilators

gesetzt. Die Entwicklungsumgebungen, die von DEViL generiert werden, enthalten einen visuellen Strukturedtor, um die visuellen Sprachen zu verarbeiten. Zur Veranschaulichung wird in der Abbildung 2 ist ein Screenshot von einem Editor dargestellt, der von DEViL generiert wurde. In diesem Editor wird ein UML-Zustandsdiagramm dargestellt. In der Abbildung 2 werden drei Fenster des Editors angezeigt.
Das Fenster ① zeigt auf der linken Seite Schaltflächen, die für die Erzeugung der Zustände und Transitionen dienen. Auf der rechten Seite wird das gezeichnete Zustandsdiagramm dargestellt. Es kann in diesem Zustandsdiagramm eine beliebige Anzahl von Zuständen und Transitionen erzeugt werden. Der Benutzer kann neue Zustände und Transitionen von der linken Seite des Fensters ① auswählen, bewegen oder löschen z.B. die Position und die Größe der Zustände ändern, um die Überlappung der Zustände bzw. Transitionen zu vermeiden. Die Zustände können an einer beliebigen Position in dem Diagramm platziert werden. Der Entwickler kann die Editieroperationen wie Einfügen von neuen Elementen wie Zustände oder Transitionen auf dem Darstellungsbereich anwenden. In diesem Zustandsdiagramm ist ein Oberzustand *the and-state* dargestellt, der die Unterzustände *b, c, d, e, f, g, h, H* enthält. Dieser Oberzustand ist durch gestrichelte Linien in drei nebenläufige Regionen mit jeweils einem Zustandsdiagramm aufgeteilt. Wird in diesen Oberzustand eingetreten, dann hat das zur Folge, dass sich das System in den Zuständen *b* und *c*, im Zustand *e* und im Zustand *h* befindet. In der mittleren Region kann vom Zustand *e* der Zustand *d* erreicht werden und von *d* wiederum Zustand *e*. Zusätzlich existiert ein History-Zustand *H*, der dazu dient, bei einem

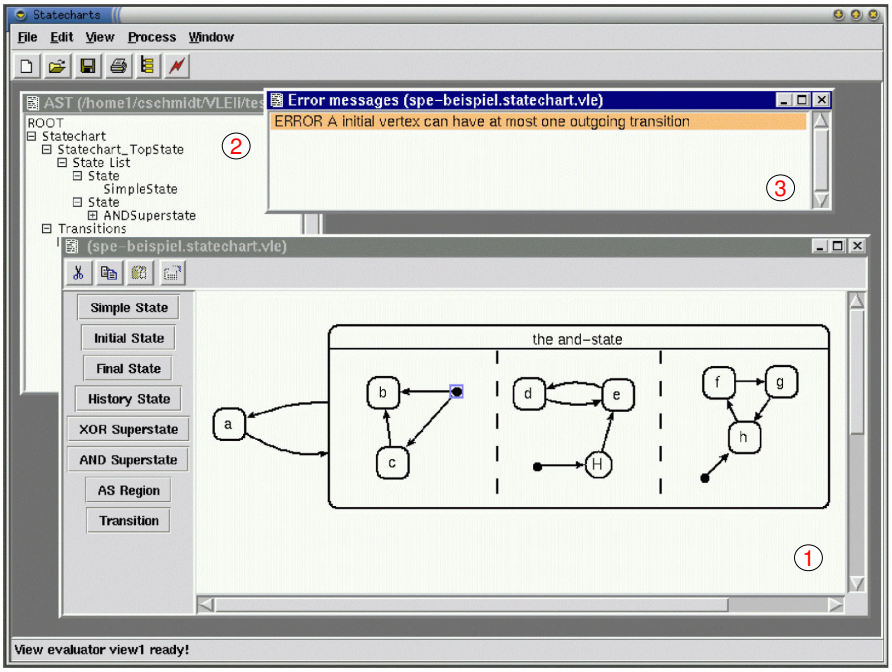

Abbildung 2: Screenshot von einem Zustandsdiagramm

Übergang von Zustand *a* in den Oberzustand *the and-state* den zuletzt besuchten Zustand einzunehmen. Der vom History-Zustand abgehende Pfeil zeigt, welcher Zustand eintreten soll, wenn in den Oberzustand übergangen wird und noch keine History-Daten vorhanden sind. Vom Oberzustand *the and-state* kann in Zustand *a* gewechselt werden.
Das Fenster ② zeigt die Struktur der visuellen Repräsentation des Zustandsdiagramms vom Fenster ① in Form eines Baums, wobei die Zustände und die Transition als Knoten dargestellt sind.
Das Fenster ③ zeigt eine Fehlermeldung. Diese Fehlermeldung wird angezeigt, da in diesem Diagramm ein Anfangszustand nur eine ausgehende Transition haben darf, jedoch im Fenster ① in der linken Region diese Eigenschaft verletzt ist.

1.3 DEViL-System

Die Abbildung 3 verschafft einen groben Überblick über das DEViL-System [4]. Der obere Teil der Abbildung 3 stellt eine Menge von visuellen Mustern (fertige Modellierungselemente mit Implementierung) dar. Die Implementierungen dieser visuellen Muster stellen visuelle Darstellungskonzepte her. Sie bauen auf den Spezifikationsmechanismen des VL-Generators auf. In dem mittleren Teil der Abbildung ist der VL-Generator mit den Spezifikationsaspekten abgebildet. Die Generierung von visuellen Editoren lässt sich in drei Spezifikationsaspekte unterteilen: abstrakte Syntax, die visuelle Sicht und die Transformation in ein Ausgabetext, die auch semantische Analyse zur Verfügung stellt. Dabei stellt die abstrakte Syntax die Basis der Spezifikation dar. Die Spezifikation der visuellen Sichten und der Textausgabe beziehen sich auf diese abstrakte Syntax.
Aus der Spezifikation kann nun der VL-Generator visuelle Strukturediteren generieren. Der Generator benutzt Mittel und Werkzeuge, wie Tool command language/Toolkit und Parcon um die grafische Oberfläche zu implementieren und Eli für Sprachimplementierungen. TcL ist eine Skriptsprache [7]. Diese wird eingesetzt um grafische Oberflächen bequem und leicht zu programmieren. Zur Lösung der grafischen Randbedingungen, dass beispielsweise zwei Linien gleich lang sein müssen, wird Parcon angewendet [3]. Das Parcon-System besteht aus einer Menge von Variablen und mathematischer Gleichungen und Ungleichungen. Er führt automatische Berechnungen zur Layoutdarstellung aus. Das Eli-System enthält Bibliotheken und Funktionen, die für die Spezifikation in DEViL benutzt werden. Dieses System kann Implementierungen von Sprachen und Übersetzungen in mehrere Zielsprachen wie C automatisch generieren.

1.4 Aufbau der Arbeit

Im Rahmen des Seminars *Unterstützung modellgetriebener Entwicklungsprozesse* wird in dieser Seminararbeit ein Generator vorgestellt, der visuelle Sprachumgebungen generiert. Diese generierten Editoren können beispielsweise als Werkzeuge bei der Modellierung von Softwaresystemen eingesetzt werden.
In Kapitel 2 wird der Generator DEViL für visuelle Sprachen präsentiert. Dabei werden in den Unterkapiteln einzeln auf die Spezifikationen der abstrakten Syntax, der visuellen Sichten und der Textausgabe eingegangen. Für die Spezifikation dieser visuellen Sichten auf einer höheren Abstraktionsebene werden in Kapitel 3 die visuellen Muster eingeführt. Für die komplette Beschreibung dieser Muster werden die sogenannten *Berechnungsrollen* und die *Kontrollattribute* definiert. In

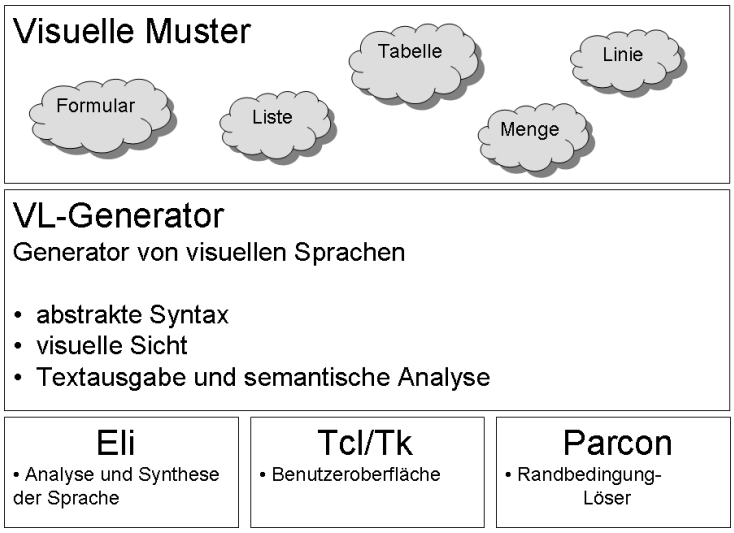

Abbildung 3: DEViL-System

Kapitel 4 werden die Vorteile und Nachteile von DEViL vorgestellt. Zum Abschluss wird die gesamte Arbeit in Kapitel 5 zusammengefasst und eine eigene Meinung zum Generator DEViL gegeben.

2 Der Generator DEViL

DEViL unterscheidet drei Spezifikationsaspekte [4]: Die Spezifikation der abstrakten Syntax, die Spezifikation der visuellen Sichten und die Spezifikation der Textausgabe und die semantische Analyse.
Die abstrakte Syntax beschreibt die Sprachkonstrukte, während die Spezifikation der visuellen Sichten eine Menge von Sichten auf der abstrakten Syntax definiert, beispielsweise die Sichten für die visuelle Darstellung oder Sicht für die Fehlermeldungen. Die Textausgabe und die semantische Analyse sind für eine Transformation eines Diagramms in einen Ausgabetext und für die Fehlerbehandlung zuständig. Die Abbildung 4 stellt diese drei Spezifikationsaspekte dar, aus denen DEViL einen Editor für visuelle Sprachen generiert.

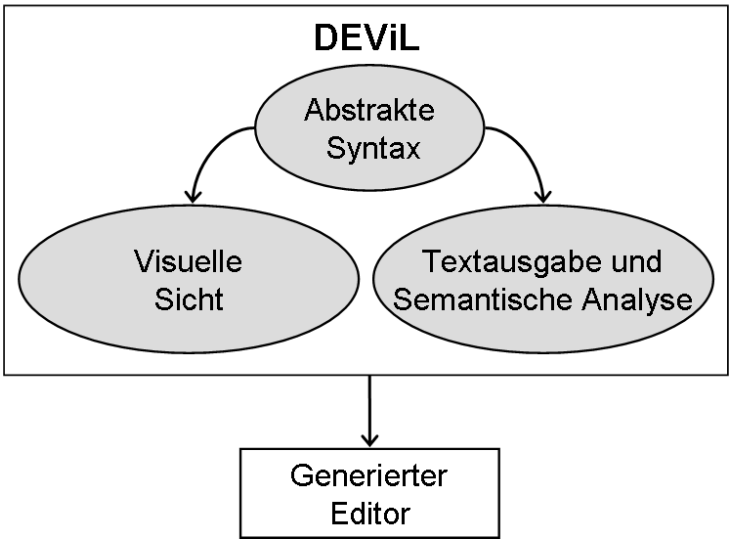

Abbildung 4: Spezifikationsaspekte in DEViL

2.1 Spezifikation der abstrakten Syntax

Die abstrakte Syntax wird in der Spezifikationssprache DSSL (DEViL Structure Specification Language) [4] spezifiziert. Die Abbildung 5 zeigt ein Beispiel für die Spezifikationssprache DSSL, in der die abstrakte Syntax eines Zustandsdiagramms spezifiziert wird. Die Notation bei dieser Spezifikationssprache ist den objektorientierten Programmiersprachen wie Java sehr ähnlich. Jede nichtabstrakte Klasse spezifiziert dabei ein bestimmtes Sprachkonstrukt. Die Klassen können Attribute besitzen, die die Eigenschaften des Sprachkonstrukts modellieren und im Rumpf der Klassen deklariert sind. DEViL unterscheidet zwischen drei Typen von Attributen [4]:

- VAL-Attribute speichern einen Wert. Dabei haben die Attribute einen Datentyp wie *VLString* oder *VLInt* oder aber auch selbstdefinierte Datentypen. Beispielsweise hat in Abbildung 1 jeder Zustand ein Attribut *name*: *Tischventilator ausgeschaltet* ist vom Typ *VLString*.
- In SUB-Attribute werden Unterstrukturen gespeichert, die Instanzen der spezifizierten Klasse darstellen. Als Unterstrukturen kommen nur die Objekte in Frage, die einem Untertyp dieser spezifizierten Klassen gehören. Die Kardinalität der Objekte kann nach der Typangabe spezifiziert werden. Dabei bedeutet ein * die Kardinalität „0..*“, also keine oder mehrere Elemente. Der Unterschied zu VAL-Attribute besteht darin, dass bei SUB-Attributen anstelle von einfachen Datentypen Objekte gespeichert werden.

Beispielsweise hat in Abbildung 1, der Oberzustand *Tischventilator eingeschaltet* drei Unterzustände. Diese drei Zustände stellen die SUB-Attribute vom Typ *State* des Oberzustands dar.

- In REF-Attribute werden die Referenzen zu anderen Objekten gespeichert. *from*- und *to*-Attribute in Abbildung 5 stellen solche REF-Attribute dar. Beispielsweise zeigt in Abbildung 1 die Transition *Knopf Stop drücken* von dem Zustand *Tischventilator eingeschaltet* auf den Zustand *Tischventilator ausgeschaltet*. Diese zwei Zustände stellen die REF-Attribute dieser Transition dar.

Das Schlüsselwort *INHERITS* beschreibt die Unterklassen-Beziehung zwischen Klassen. Die Unterklassen besitzen die Attribute von ihren Oberklassen, aber auch die selbst definierten Attribute. DSSL erlaubt die Mehrfachvererbung, d.h. es kann von mehreren Klassen geerbt werden. Die Abbildung 5 definiert die abstrakte Syntax von einem Zustandsdiagramm folgendermaßen [6]:

In jeder DSSL-Spezifikation wird eine Wurzelklasse definiert. Die Wurzelklasse *Root* aus Abbildung 5 besteht aus einer Menge von Objekten vom Typ *Statechart-Diagram*. Die Klasse *StatechartDiagram* hat die SUB-Attribute *states* und *transitions*, die die Menge von Zuständen und Transitionen eines Zustandsdiagramms modellieren. Zudem haben *StatechartDiagrams* die Attribute *name* vom Typ *VL-String* und *setSize* vom Typ *VLPoint*, die für die Name bzw. die Größe des Diagramms zuständig sind. Mit *INIT* wird der Anfangswert angegeben, d.h. die Größe des Diagramms wird mit einem bestimmten Wert initialisiert.
Die abstrakte Klasse *State* hat den Attribut *position*, der für die Position des Zustands zuständig ist. Ein Zustand kann entweder ein Anfangzustand (*InitialState*), ein Oberzustand (*XORState*) oder ein einfacher Zustand (*SimpleState*) sein. Das Attribut *position* wird von den Klassen *SimpleState*, *InitialState* und *XORState* geerbt. *XORStates* beinhalten wiederum eine Liste von Zuständen und die Attribute *name* und *setSize*. Beispielsweise ist in der Abbildung 1 der Anfangzustand vom Typ *InitialState*, der *Tischventilator ausgeschaltet* ist vom Typ *SimpleState*, und der Oberzustand *Tischventilator eingeschaltet* ist vom Typ *XORState*. Oberzustand *Tischventilator eingeschaltet* enthält Unterzustände wie *Stufe 1* vom Typ *SimpleState* und die Transitionen wie *Knopf 1 drücken*. Die Klasse Transition hat zwei Referenzattribute *from* und *to* um die Start- bzw. Endpunkte der Transition zu speichern. Sie hat auch den Attribut *position*, der zur Speicherung der Layoutinformation verwendet wird [5].

```
CLASS Root {
        statechartDiagrams: SUB StatechartDiagram*;
}
CLASS StatechartDiagram {
        name: VAL VLString;
        states: SUB State*;
        transitions: SUB Transition*;
        setSize: VAL VLPoint INIT "400 300" ;
}
CLASS Transition {
        from: REF State;
        to: REF State;
        position: VAL VLPoint ;
}
ABSTRACT CLASS State {
        position: VAL VLPoint ;
}
CLASS SimpleState INHERITS State {
        name: VAL VLString;
}
CLASS InitialState INHERITS State {
}
CLASS XORState INHERITS State {
        name: VAL VLString;
        subStates: SUB State*;
        setSize: VAL VLPoint INIT "140 80" ;
}
```

Abbildung 5: Abstrakte Syntax eines Zustandsdiagramms

Die Abbildung 6 zeigt ein Screenshot eines Fensters von einer Sicht eines Diagramms. Dieses Fenster zeigt die Struktur eines erzeugten Zustandsdiagramms, die in Form eines Baumes dargestellt ist. Die Knoten wie *Root (unnamed[1])*, *StatechartDiagramm (MyStatechartDiagram)*, *SimpleState (A)*, *XORState (B)*, *InitialState (unnamed[2])* und *Transition (unnamed [3])* stellen die erzeugten Objekte im Zustandsdiagramm dar. Die anderen Knoten wie *name*, *position*, *setSize*, *from* und *to* repräsentieren die Attribute dieser Objekte. Die Werte von VAL-, SUB- oder REF-Attribute werden in Anführungszeichen geschrieben, wie beispielsweise der Wert des Attributs *position* („170 270“). Die Werte der REF-Attribute von *Transition (unnamed[4])* sind *unnamed[2]* und *C*.

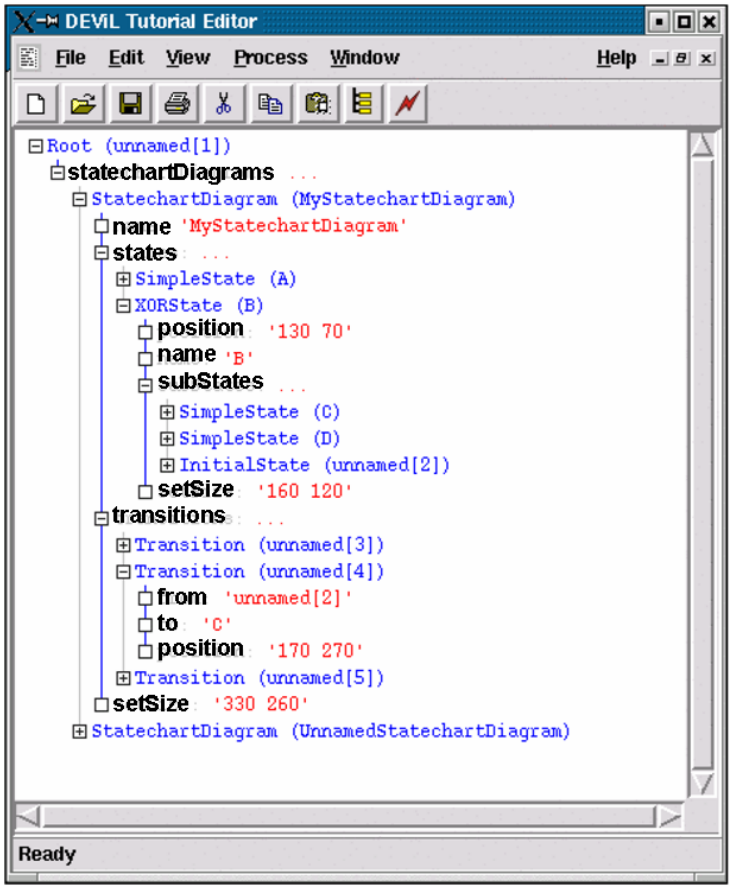

Abbildung 6: Screenshot von Root in DEViL [6]

2.2 Spezifikation der visuellen Sichten

Auf der abstrakten Syntax können mehrere Sichten definiert werden. Jede Sicht definiert eine konkrete Darstellung eines bestimmten Teils der abstrakten Syntax. Die Spezifikation einer Sicht besteht nach [4] aus Deklaration der Sichten und Schaltflächen und aus Berechnung der Darstellung.

2.2.1 Deklaration der Sichten und Schaltflächen

Bei der Deklaration der Sichten bestimmen Angaben beispielsweise die Schaltflächen, welche speziellen Teile der abstrakten Syntax visuell dargestellt werden.

```
SYMBOL  StatechartDiagram
COMPUTE
  DrawObject ("Form", ...)
END;
```

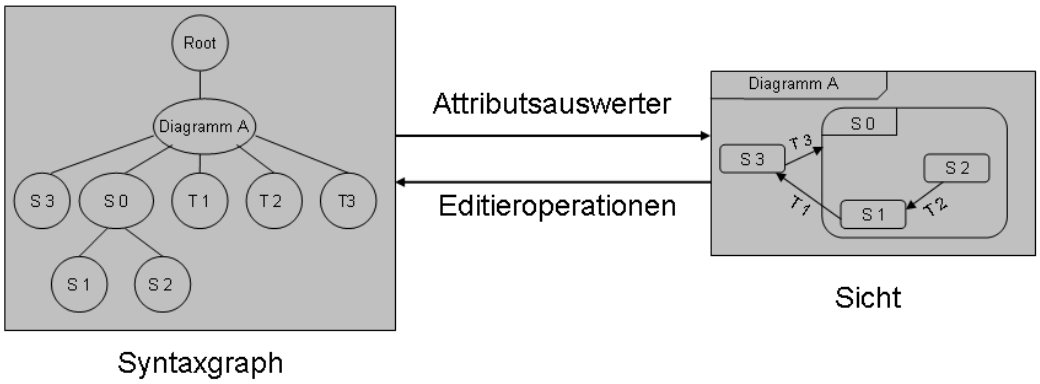

Abbildung 7: Berechnung der Darstellung eines Zustandsdiagramms

In den Sichten werden Knöpfe oder Schaltflächen definiert, die die grafische Interaktion ermöglichen (Benutzerschnittstelle). Im Folgenden wird beispielsweise eine Sicht für den Sprachkonstrukt *root* mit dem Namen *rootView* deklariert:
VIEW rootView root {
BUTTON *Statechart Diagram* INSERTS StatechartDiagram; }

In dieser Sicht werden Zustandsdiagramme erzeugt [6]. Dieser Sicht wird eine Schaltfläche mit dem Namen *Statechart Diagram* hinzugefügt, die zur Erzeugung von *StatechartDiagram*-Objekten dient. Mit Hilfe des Schlüsselworts *INSERT* werden diese *StatechartDiagram*-Objekte hinzugefügt.

2.2.2 Berechnung der Darstellung

Mit Hilfe der Attributberechnungen werden die grafische Darstellung und deren Ausrichtung festgelegt. Abbildung 7 zeigt ein Beispiel der Spezifikation der visuellen Darstellung. Rechts unten ist die visuelle Sicht eines Zustandsdiagramms dargestellt. Links unten wird der Syntaxgraph des Zustandsdiagramms gezeigt. Oben wird die Berechnung der Darstellung des Symbols *StatechartDiagram* an-

```
SYMBOL StatechartDiagram_states
                INHERITS VPFormElement, VPSet
COMPUTE
    SYNT.formElementName = "body";
END;

SYMBOL Simplestate  INHERITS VPSetElement,
                VPForm, VPConnectionEndpoint
COMPUTE
    SYNT.drawing = ADDOF(SimpleStateDrawing);
END;
```

Abbildung 8: Sichtspezifikation mit visuellen Mustern

gezeigt. Es wird eine Funktion aufgerufen, die für die Berechnung der visuellen Darstellung des Zustandsdiagramms zuständig ist.
Die Funktion *DrawObject(„Form“,...)* berechnet die Darstellungsform von *StatechartDiagram*. Die Darstellung von den erzeugten Zustandsdiagrammen hat die Form eines Formulars. Die visuelle Darstellung bietet den Benutzern die Möglichkeit, Editieroperationen ausführen zu können. Beispielsweise in Abbildung 7 würde das Löschen des Zustands *State 3* ausgeführt. Dieser Vorgang führt zu einem automatischen Löschen des entsprechenden Knotens in der Struktur, die eine Form eines Baumes hat.
Zur Spezifikation der visuellen Darstellungen werden visuelle Muster eingesetzt, die eine höhere Abstraktionsebene definieren. Bei visuellen Mustern handelt es sich zum Beispiel um Listen, Mengen, Tabellen, Bäume, Verbindungen oder Formen [4], die wieder verwendet werden können. Diese Muster werden zu den Symbolen zugeordnet und definieren ihre Eigenschaften.

Die Abbildung 8 stellt ein Teil der visuellen Sichtspezifikation dar, bei der visuelle Muster eingesetzt sind. Diese visuellen Muster werden in Kapitel 3 näher erklärt [4]. *VPSet* stellt eine Menge von Elementen dar, während *VPSetElement* die Elemente dieser Menge repräsentieren. *VPForm* ist ein Formular-Muster und besteht aus einem Tupel von Elementen, die in einer festen Stellung zueinander stehen, während *VPFormElement* ein Element dieses Formulars darstellt. *VPConnectionEndpoint*-Elemente stellen Zustände dar, die mit Linien verbunden

werden dürfen. In dieser Sichtspezifikation erbt *StatechartDiagram_states* vom *VPSet*, da dieses Symbol eine Menge von Zuständen darstellt. Dieses Symbole erbt ebenfalls von *VPFormElement*, da diese Zustände in einem Element des Formulars dargestellt werden. Das Symbol *StatechartDiagram* hat die Eigenschaft eines Formular-Musters, da es aus zwei Teilen besteht: ein Teil mit dem Namen des Zustandsdiagramms, und der andere Teil, indem sich die Zustände und Transitionen befinden. Daraus folgt, dass der *StatechartDiagram_states* vom *VPFormElement* erben soll.
Das Symbol *SimpleState* erbt von dem visuellen Muster *VPSetElement*, da das Symbol ein Element einer Menge von Zuständen in der dargestellten Sicht ist. Die Darstellung der Zustände haben die Form von einem Form-Muster, so dass *SimpleState* vom *VPForm* erbt. Außerdem erbt sie von dem Ausdruck *VPConnectionEndpoint*, da *SimpleState* ein Endpunkt einer Verbindung sein kann.
Vererbte Attributberechnungen können überschrieben werden und bereits existierende Berechnungen können erweitert werden, um Details in der grafischen Repräsentation zu beeinflussen. In der Abbildung wird das geerbte *formElementName*-Attribut von dem Ausdruck *VPFormElement* überschrieben, da der Name des *VPFormElements body* dem Namen eines Containers in der generischen Zeichnung entsprechen muss, d.h. der Teil, indem die Zustände und die Transitionen gezeichnet werden. In der Abbildung 9 ist eine solche generische Zeichnung mit den Namen der Container angezeigt. Beispielsweise befinden sich in Abbildung 1 die Unterzustände des Oberzustands *Tischventilator eingeschaltet* in dem unteren Teil, da der obere Teil für den Namen des Zustandsdiagramms ist. Genauso wird auch das *drawing*-Attribut, das vom visuellen Muster *VPForm* geerbt wurde, überschrieben. Der Wert dieses Attributs kann auf die grafische Darstellung Einfluss nehmen. Dieser Attribut stellt die visuelle Darstellung des Symbols *SimpleState* dar, wobei *SimpleStateDrawing* für die Zeichnung der *SimpleState*-Zustände geeignet ist und diese Zeichnung wird in den Zustandsdiagramm eingefügt. Beispielsweise ist in Abbildung 1 der Zustand *Tischventilator ausgeschaltet* als abgerundetes Rechteck gezeichnet, da er vom Typ *SimpleState* ist und die Darstellung von solchen Zuständen als abgerundetes Rechtecke definiert ist. Die Zeichnung der Symbole werden in den nächsten Absatz näher beschrieben.

Generische Zeichnungen
Die konkrete Darstellung von Objekten der Sichten werden von sogenannten „generischen Zeichnungen“ definiert, d.h. sie spezifizieren die Darstellungsdetails dieser Objekte. Beispielsweise ist in Abbildung 1 das Zustandsdiagramm als ein Rechteck dargestellt. Die Abbildung 9 zeigt ein Screenshot bei der Erstellung eines Formular-Muster, der für die Darstellung eines *StatechartDiagrams* geeig-

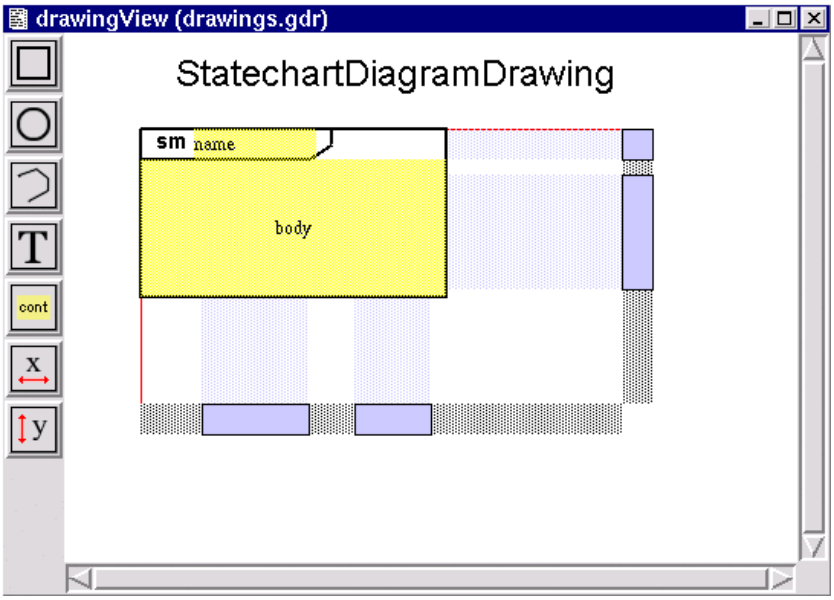

Abbildung 9: Screenshot bei der Erstellung eines Formulars

net ist. Dieses Formular-Muster besteht aus zwei Container *name* und *body*. Diese Container werden hinzugefügt, um die Position zu spezifizieren, an welchen *name* und *body* positioniert werden sollen. In dem *name*-Container wird der Name des Zustandsdiagramms gespeichert und in dem *body*-Container werden die Zustände und Transitionen gezeichnet. Beispielsweise wäre in Abbildung 1 *Steuerung des Tischventilators* in *name*-Container gespeichert und in *body*-Container alle Zustände und Transitionen, die die Steuerung des Ventilators modellieren. Auf der linken Seite befinden sich die Knöpfe für die Erstellung des Formular-Musters und die Dehnungsintervalle auf der X- bzw. Y-Achse dieses Objektes. Auf diese Weise wird ein Objekt visuell spezifiziert und erstellt, der bei den Sichten benutzt wird.

2.3 Textausgabe und semantische Analyse

Die Aufgabe der Textausgabe besteht darin, das erzeugte Zustandsdiagramm automatisch in einen Ausgabetext zu transformieren. Diese Transformation wird durch die abstrakte Syntax spezifiziert. Für diese Textausgabe wird PTG (pattern based text generator) eingesetzt [4], also ein muster-basierter Textgenerator. Eine PTG Spezifikation ist eine Menge von Mustern, welche die Struktur und die Textbestandteile eines Ausgabetextes beschreiben. PTG generiert eine Funktion für jedes Muster. Aufrufe dieser Funktionen bei der Spezifikation der Textausgabe erstellen

eine Instanz des Ausgabetextes. Dieser Ausgabetext stellt die Struktur des erzeugten Zustandsdiagramms dar.
Beispielsweise wird für die Abbildung 1 ein Ausgabetext generiert. Ein Textteil dieses Ausgabetextes ist *SIMPLE STATE „Tischventilator ausgeschaltet“*. Als erstes wird eine PTG-Funktion für ein SimpleState-Muster generiert, die einen Parameter, der für den Namen des Zustands steht, erwartet. Die entsprechende Spezifikation der Textausgabe lautet:

```
SYMBOL codegen_SimpleState
COMPUTE
SYNT.code = PTGSimpleState(THIS.pers_name);
END;
```

Die Funktion *code* ruft die generierte *PTGSimpleState*-Funktion auf, wobei *THIS.pers_name* für den Namen des erzeugten Zustands steht. Dann wird das Textmuster

SIMPLE STATE „Tischventilator ausgeschaltet“

ausgegeben. Auf diese Weise werden die Textteile des Ausgabetextes ausgegeben.
Mit Hilfe der semantischen Analyse werden Fehlermeldungen automatisch ausgegeben. Beispielsweise ist in der Abbildung 2 eine Fehlermeldung ausgegeben, da ein Zustand keine zwei ausgehende Transitionen haben darf.

3 Visuelle Muster

Die visuellen Muster wurden bereits in der Spezifikation der visuellen Sichten eingeführt. In diesem Kapitel werden die visuellen Muster genauer betrachtet und deren Implementierung vorgestellt.

3.1 Definition und Eigenschaften

Visuelle Muster führen eine höhere Abstraktionsebene für die Spezifikation der visuellen Darstellungen ein und stellen Teile der abstrakten Syntax grafisch dar.

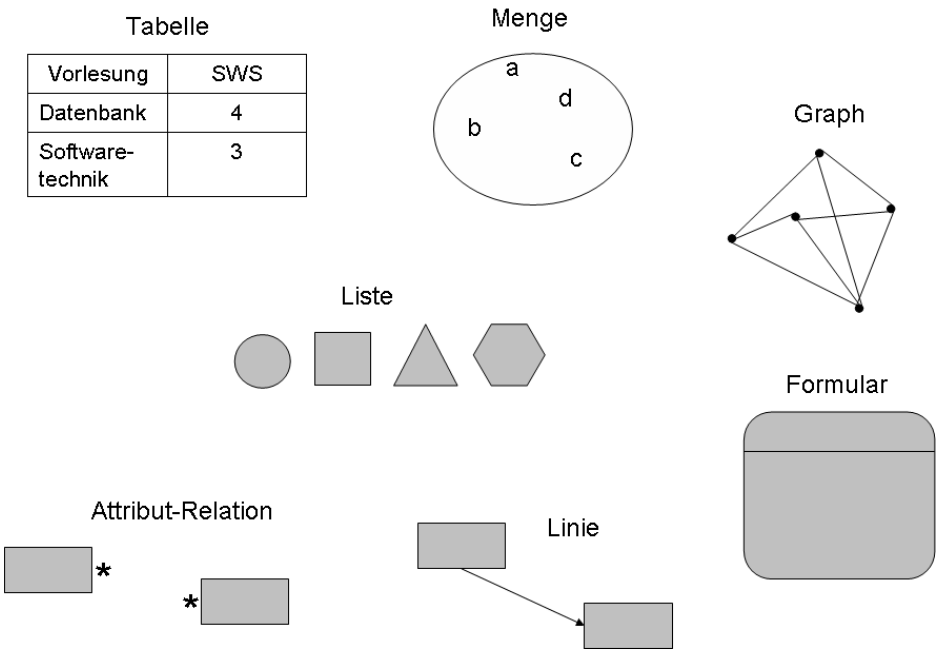

Abbildung 10: Visuelle Muster [5]

Visuelle Muster sind Implementierungen von gemeinsamen Repräsentationskonzepten wie Listen, Mengen, Tabellen, Bäume, Verbindungen oder Formen [4], die wiederverwendet werden können. Neben den Darstellungseigenschaften der visuelle Konstrukte kapseln visuelle Muster auch Interaktions- und Layoutmechanismen für eine benutzerfreundliche Entwicklungsumgebung. Visuelle Muster können miteinander kombiniert werden um eine vollständige Beschreibung einer grafischen Darstellung zu erhalten. Die Spezifikationen dieser grafischen Darstellung wird mit Hilfe dieser Muster kurz und übersichtlich. Im Folgenden werden visuelle Muster vorgestellt [5]:

1. Das *Tabellen*-Muster stellt eine Sequenz von Tupeln mit gleicher Struktur dar. Sie erfassen die Beziehungen zwischen den Spalten. Außerdem bieten *Tabellen*-Muster Mechanismen an, mit denen Überschriften für die Spalten definiert werden können.

2. Das *Mengen*-Muster visualisiert eine Menge von Elementen, wobei für die Menge die Ordnung bzw. Reihenfolge der Elemente irrelevant ist.

3. Das *Graph*-Muster stellt einen Graphen dar. Dazu werden die *Linien*- und *Mengen*-Muster kombiniert.

4. Das *Listen*-Muster stellt eine Folge von Elementen dar, die in einer bestimmten Reihenfolge angeordnet sind.

5. Das *Linien*-Muster visualisiert eine Linienverbindung zwischen zwei Komponenten,

6. Das *Attribut-Relations*-Muster visualisiert eine Relation zwischen zwei Komponenten, so dass mindestens ein Attribut der beiden Komponenten gleich ist. Diese Attribute sind beispielsweise der Name und die Farbe.

7. Das *Formular*-Muster stellt ein Tupel aus n Elementen dar, wobei die Elemente des Tupels in einer festen Stellung zueinander stehen.

All diese Muster sind durch ihre visuellen Eigenschaften gekennzeichnet. Diese Eigenschaften beschreiben die Darstellung der Muster auf den Darstellungsbereich und wie sich zueinander dargestellt werden, wie zum Beispiel

- die Reihenfolge entlang einer bestimmten Richtung, beispielsweise eine Liste mit vertikaler Anordnung der Elemente,

- eine bestimmte Position in einem bestimmten Bereich, beispielsweise ein Form-Muster, wobei die Elemente des n-Tupels in einer festgelegten Position zueinander stehen.

3.2 Implementierung der visuellen Muster

Die Muster, wie *List*-Muster, werden implementiert, d.h. es werden konkrete Attribute zu diesen Muster zugeordnet. Mit Hilfe dieser Eigenschaften kann ein Muster Operationen kapseln. Das sind Operationen, die

- die Komponenten der Struktur „layouten“ bzw. strukturieren, wie zum Beispiel die Elemente einer Liste innerhalb eines bestimmten Bereichs,

- die grafische Darstellung zeichnen, wie beispielsweise der Rahmen um eine Liste zusammen mit den Trennungslinien zwischen einzelnen Listenelementen,

- die Möglichkeit für Benutzerinteraktionen anbieten, wie zum Beispiel das Einfügen oder Löschen von Elementen einer Liste.

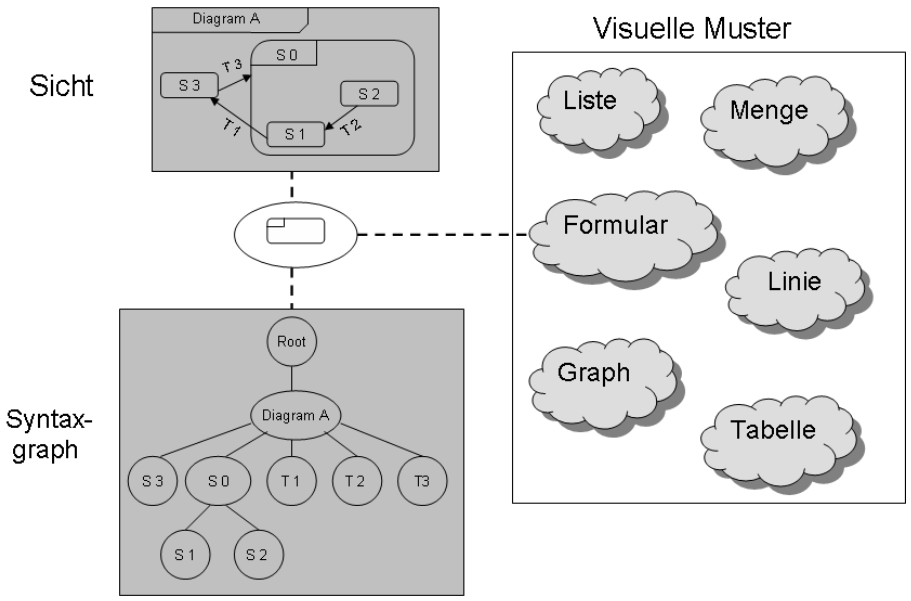

Abbildung 11: Konzept der visuellen Muster

Die visuellen Muster [5] werden benutzt, um die abstrakte Syntax, die editierbaren Operationen bei der Interaktion und das visuelle Repräsentation der Sprachkonstrukte zu beschreiben.

In Abbildung 11 wird im rechten Fenster eine Menge von visuellen Mustern *Menge*, *Linie*, *Liste*, *Formular* und *Tabelle* dargestellt. Links oben wird die visuelle Sicht von einem Zustandsdiagramm angezeigt. Links unten wird die Struktur dieser Darstellung gezeigt. Das Zustand *State 0* wird durch ein *Formular*-Muster definiert. Dieses Muster hat zwei Teile: Ein Teil, wo der Name des Zustands *State 0* enthalten ist und ein anderen Teil, wo sich die Zustände wie *State 1* und Transitionen befinden. Das Zustandsdiagramm wird durch ein *Mengen*-Muster definiert, wobei die Zustände als Elemente dieser Menge definiert sind. Das bedeutet, dass er eine Menge von Zuständen hat ohne Beachtung der Reihenfolge, sie sind also beliebig angeordnet. Würde ein anderes Muster wie das *Listen*-Muster dafür benutzt werden, wäre die Eigenschaft dieses Diagramms verletzt, da dann die Zustände in einer bestimmten Reihenfolge angeordnet wären. Beispielsweise bilden in Abbildung 1 der Anfangszustand, der Zustand *Tischventilator ausgeschaltet* und der Oberzustand *Tischventilator eingeschaltet* bilden die Elemente einer Menge und müssen nicht in einer bestimmten Richtung zugeordnet sein.

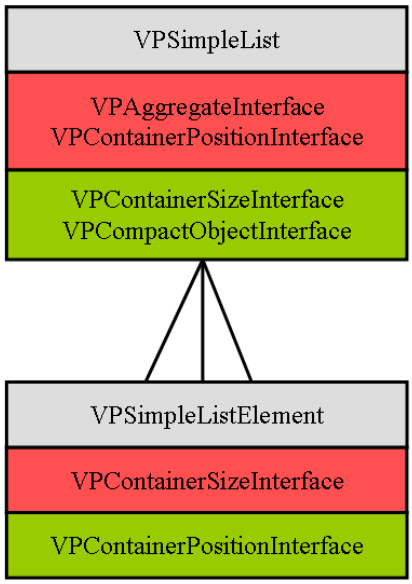

Abbildung 12: Berechnungsrollendiagramm für List-Muster

3.2.1 Berechnungsrollen

Im Folgenden wird die Implementierung der visuellen Muster betrachtet. Diese Implementierungen setzen sich aus Berechnungsrollen zusammen. Eine Berechnungsrolle stellt Teileigenschaften des Musters grafisch dar. Die Berechnungsrollen eines Musters arbeiten zur Berechnung der visuellen Darstellung und des Layouts miteinander. Die Muster kooperieren miteinander, indem den Grammatiksymbolen Berechnungsrollen verschiedener Muster zugeordnet werden, beispielsweise wird dem Symbol *StatechartDiagram_states* aus Abbildung 8 die Berechnungsrollen *VPFormElement* und *VPSet* zugeordnet. Dabei erwartet und implementiert jede Berechnungsrolle bestimmte Schnittstellen.

In Abbildung 12 ist ein Berechnungsrollendiagramm des *List*-Musters dargestellt. Dieses Muster besteht aus zwei Berechnungsrollen *VPSimpleList* und *VPSimpleListElement*. Es hat die Form eines Baumes und besteht aus zwei Knoten. Diese Knoten sind durch drei Linien verbunden. Die Struktur bedeutet, dass die *VPSimpleList*-Berechnungsrolle eine Liste ist, der mehrere *VPSimpleListElement*-Berechnungsrolle als Elemente haben kann, und *VPSimpleListElement*-Berechnungsrolle ein Element von der Liste *VPSimpleList*-Berechnungsrolle darstellt, also eine Kardinalität von 0..*. Die Schnittstellen werden in dem mittleren und in dem unteren Teil eines Knotens dargestellt [3]. Es gibt zwei Arten

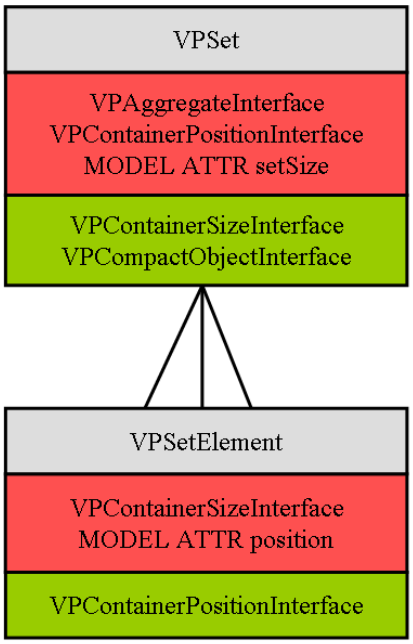

Abbildung 13: Berechnungsrollendiagramm für *Mengen*-Muster

von Schnittstellen: Die geforderten und die bereitgestellten Schnittstellen. In Abbildung 12 stellen *VPAggregateInterface*, *VPContainerPositionInterface* und *VPContainerSizeInterface* die geforderten Schnittstellen dar und befinden sich in dem zweiten Teil eines Knotens während *VPContainerSizeInterface*, *VPCompactObjectInterface* und *VPContainerPositionInterface* die bereitgestellten Schnittstellen darstellen. Diese befinden sich in dem dritten Teil eines Knotens. Schnittstellen geben Auskunft über bestimmte Eigenschaften einer Rolle, beispielsweise die Größe und Position. Diese Eigenschaften können ebenfalls anderen Berechnungsrollen zur Verfügung stehen. Jede Schnittstelle stellt eine Menge von Attributen dar. Die Attribute der geforderten Schnittstellen werden für andere Berechnungen benutzt und die bereitgestellten Schnittstellen benutzen diese Attribute.
Um den Unterschied zwischen der geforderten und der bereitgestellten Schnittstelle zu verdeutlichen, werden die Schnittstellen der Abbildung 12 näher betrachtet. Durch das Lesen der geforderten Schnittstelle *VPContainerSizeInterface* von *VPSimpleListElement* wird die gewünschte Größe eines Elements festgelegt. Daraus wird nun die gewünschte Größe der gesamten Liste berechnet und der Berechnungsrolle *VPSimpleList* bereitgestellt.

In Abbildung 13 wird ein weiterer Berechnungsrollendiagramm mit der Struktur und den Schnittstellen des *Mengen*-Musters gezeigt. Es besteht aus zwei Berechnungsrollen: *VPSet*-Berechnungsrolle stellt eine Menge dar, während die *VPSetElement*-Berechnungsrolle die Elemente der Menge repräsentieren. *VPAggregateInterface*, *VPContainerPositionInterface*, *VPContainerSizeInterface* und die Attribute *MODEL ATTR position* und *MODEL ATTR setSize* stellen die geforderten Schnittstellen bzw. Attribute dar, *VPContainerSizeInterface*, *VPCompactObjectInterface* und *VPContainerPositionInterface* die bereitgestellten Schnittstellen. Die Attribute *MODEL ATTR position* und *MODEL ATTR setSize* bedeuten, dass die Größe des Mengenbereichs und die Position der Elemente dieser Menge festgelegt sind. Die Berechnungsrollen *VPSet* und *VPSetElement* werden bei der Sichtspezifikation aus Abbildung 8 den Symbolen *StatechartDiagram_states* und *SimpleState* zugeordnet. Die geerbte Berechnungsrolle *VPSet* vom Symbol *StatechartDiagram_states* stellt die Schnittstelle *VPContainerSizeInterface* bereit. Diese Schnittstelle wird in der geerbten Berechnungsrolle *VPFormElement* gefordert. Auf diese Weise lässt sich die Korrektheit der Kombination der geerbten Berechnungsrollen für einen Symbol prüfen. Das bedeutet, dass die Spezifikation von einem Symbol gültig ist, falls alle geforderten Schnittstellen der geerbten Berechnungsrollen von anderen geerbten Berechnungsrollen bereitgestellt werden. Dabei dürfen zwei Berechnungsrollen nicht die gleiche Schnittstelle bereitstellen.

3.2.2 Kontrollattribute

Für die komplette Beschreibung der Muster spielen neben den Berechnungsrollen auch die sogenannten *Kontrollattribute* der Berechnungsrollen eine große Rolle. Die Funktion dieser Kontrollattribute ist die Konfiguration von Darstellungseinzelheiten der verschiedenen Muster. Sie repräsentieren Eigenschaften der Berechnungsrollen, die für jedes Muster angepasst werden können. Beispielsweise hat die Schnittstelle *VPContainerPositionInterface* der Berchnungsrolle *VPSetElement* den Kontrollattribut *oPosition*, der für die Position des Darstellungsbereichs zuständig ist. Die Kontrollattribute können in zwei Klassen aufgeteilt werden:

- Layoutattribute wie zum Beispiel die Richtung der Anordnung der Elemente und
- Darstellungsattribute wie beispielsweise die Farbe des Hintergrunds.

Die Abbildung 14 zeigt die Kontrollattribute der Berechnungsrollen des List-Musters [5]. Dabei stellen *minWidth*, *direction*, *pad*, *alignX*, *alignY* und *elementDistance* beispielsweise Layoutattribute und *bgFillColor, bgOutlineColor* und *se-*

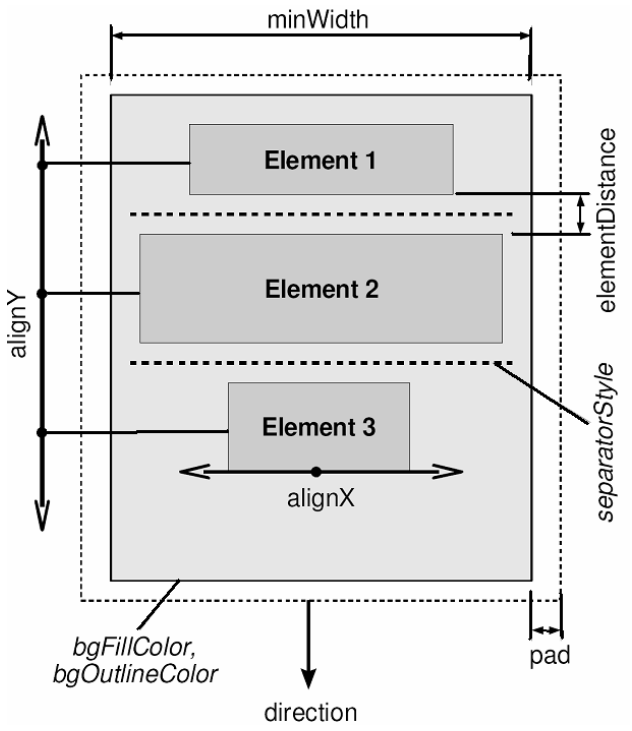

Abbildung 14: Darstellung eines List-Musters mit konkreten Parametern

paratorStyle Darstellungsattribute dar. *minWidth* gibt die minimale Breite der Liste, *direction* die Richtung der Liste (horizontale oder vertikale Ausrichtung) an, *pad* und *elementDistance* geben die Details des Layouts einer Liste an wie die Breite des Randes und der Abstand zwischen den Elementen einer Liste. Die Darstellungsattribute *bgFillColor, bgOutlineColor* beschreiben die Färbung des Hintergrunds und des Rahmens um die Liste und *separatorStyle* gibt die Eigenschaften der Linien an, die Listenelemente voneinander trennen.
VPSet hat ähnliche Kontrollattribute wie *VPSimpleList* wie beispielsweise *pad*, *bgFillColor* und *bgOutlineColor*. Zusätzlich hat *VPSet* Kontrollattribute *sizeAttr* und *defaultCursorSize*. *sizeAttr* ist ein Referenzattribut zur Speicherung der Mengengröße, während *defaultCursorSize* die Größe des Einfügecursors angibt, der die aktuelle Bearbeitungsposition anzeigt.
In der Abbildung 1 hat der Oberzustand *Tischventilator eingeschaltet* Kontrollattribute wie *drawing* und *bgFillColor*, die die visuelle Darstellung dieses Oberzustands und die Hintergrundfarbe beschreiben. In diesem Fall ist die Farbe des Hintergrunds blau.

4 Vorteile und Nachteile von DEViL

DEViL und sein Vorgänger VL-Eli haben sich bei vielen Sprachimplementierungen als erfolgreich erwiesen. Im Folgenden werden eine Reihe von Vorteilen, die das DEViL-System bietet, vorgestellt [3]:

- DEViL-Spezifikationen haben eine einheitliche Struktur. Diese einheitliche Struktur sorgt dann für eine strukturierte Übersicht.
- Die Editoren von visuellen Sprachen können in kurzer Zeit effizient implementiert werden.
- Die visuelle Darstellung, Generierung des Ausgabetextes und die semantische Analyse können unabhängig voneinander spezifiziert werden.
- Wenn neue Anforderungen gestellt werden, können die abstrakte Syntax und die grafische Repräsentation mit geringem Aufwand verändert und erweitert werden.

Obwohl die Entwicklung von DEViL recht fortgeschritten ist, bestehen dennoch Verbesserungsmöglichkeiten. Für zukünftige Erweiterungen kämen folgende Aspekte in Frage [2]:

- Um den Neulingen den Umgang mit DEViL einfacher zu machen, könnte eine visuelle Entwicklungsumgebung für DEViL selbst generiert werden. Ein Beispiel für die visuelle Entwicklungsumgebung für DEViL ist *DEViL Designer* [3]. Dieses System erlaubt, die abstrakte Syntax, die visuelle Darstellung und die Textausgabe visuell zu modellieren.
- Bei den visuellen Sprachen besteht der Wunsch, Animationen in visuelle Programme einzubeziehen. Mit der Animation ist es ebenfalls möglich, den Ablauf von Vorgängen darzustellen.
- Visuelle Muster können eine große Rolle beim Sprachentwurf spielen. Da bei der Spezifikation der visuellen Sprachen in DEViL häufig visuelle Muster eingesetzt werden, wäre es sinnvoll, die Konsequenzen der Musterwahl auf die Benutzbarkeit visueller Sprachen zu überprüfen. Dabei wäre es wichtig, die Vor- und Nachteile von einigen visuellen Mustern zu forschen und die wichtigen Muster und Musterkombinationen weiter zu untersuchen und einzusetzen.

5 Zusammenfassung und Bewertung

In diesem Kapitel werden eine Zusammenfassung dieser Ausarbeitung und eine eigene Bewertung eingeführt.

5.1 Zusammenfassung

In dieser Ausarbeitung wurde der Generator DEViL für visuelle Sprachen vorgestellt. DEViL unterscheidet drei Spezifikationsaspekte:

- Spezifikation der abstrakten Syntax: Die Spezifikation der abstrakten Syntax wird in einer Spezifikationssprache DSSL formuliert. Diese abstrakte Syntax der Sprache dient als Basis für alle anderen Spezifikationen.
- Spezifikation der visuellen Sichten: Eine Sicht definiert die konkrete Repräsentation von einem Teil der abstrakten Syntax und erlaubt somit, Änderungen an der Syntax vorzunehmen. Für die Spezifikationen der visuellen Sichten auf einem höheren Niveau werden visuelle Muster eingesetzt.
- Textausgabe und semantische Analyse: Die Aufgabe der Textausgabe besteht darin, einen Ausgabetext zu generieren. Während die semantische Analyse ist für die Fehlerbehandlung zuständig.

Für die Spezifikation der visuellen Darstellungen auf einer höheren Abstraktionsebene wurden visuelle Muster eingesetzt. Zur Implementierung dieser visuellen Muster wurden Berechnungsrollen mit ihren Schnittstellen und die Kontrollattribute vorgestellt. Diese wurden ausführlich an den Beispielen *VPSimpleList* und *VPSet* dargestellt. Bei der Implementierung dieser Muster wurden diese Berechnungsrollen den Symbolen zugeordnet und dadurch werden diesen Symbolen Attributberechnungen zugefügt, die überschrieben werden können.

5.2 Bewertung

Die mit DEViL generierten Editoren sind benutzerfreundlich. Das bedeutet, dass sie einfach zu bedienen sind, zum Beispiel beim Ändern eines Attributs in der visuellen Darstellung einer Sicht wird automatisch das Attribut auch in der Struktur dieser Sicht verändert. Jeder Editor bietet eine Baum-Sicht, die die Struktur des erzeugten Programms darstellt. Außerdem ist es ein wesentlicher Vorteil, dass in

dem von DEViL generierten Editor Sprachkonstrukte wie Zustände oder Transitionen durch einen Klick eines Buttons bzw. Knopfes an einer gewünschten Einfügestelle eingefügt werden können. DEViL bietet auch das *Kopieren/Einfügen*-Mechanismus für bessere benutzerfreundlichere Sprachumgebungen.

Literatur

[1] FOWLER, MARTIN: *UML konzentriert, 3 Auflage*. Addison-Wesley, ISBN: 3-8273-2126-3, 2004.

[2] MATTHIAS JUNG, UWE KASTENS, CARSTEN SCHMIDT CHRISTIAN SCHINDLER: *Visual Pattern in the VL-Eli System*. In Proceedings of the 7th International Conference on Compiler Construction (CC'2001), Nummer 2027 in Lecture Notes in Computer Science, S. 361-364. Springer Verlag, 2001.

[3] SCHMIDT, CARSTEN: *Generierung von Struktureditoren für anspruchsvolle visuelle Sprachen*. Dissertation, Universität Paderborn, 2006.

[4] UWE KASTENS, CARSTEN SCHMIDT, BASTIAN CRAMER: *Using DEViL for Implementation of Domain-Specific Visual Languages*. In Proceedings of the 1st Workshop on Domain-Specific Program Development, Nantes, France, 2006, Webworkshop ECCOP 2006. Organisationsseite: http://phoenix.labri.fr/DSPD.

[5] UWE KASTENS, CARSTEN SCHMIDT: *Implementation of Visual Languages using Pattern-based Specifications*. Technischer Bericht, Reihe Informatik tr-ri-02-233, Universität Paderborn Fachbereich Mathematik-Informatik, 2002.

[6] UWE KASTENS, CARSTEN SCHMIDT: *DEViL Tutorial*. Dokumentation, Universität Paderborn, http://ag-kastens.uni-paderborn.de/forschung/devil/documentation/index.php, Datum des Zugriffs: 07.02.2007.

[7] WIKIPEDIA, DIE FREIE ENZYKLOPÄDIE: *Datum des Zugriffs: 18.12.2006*. http://de.wikipedia.org.