

---

# Generierung von Animation und Simulation für graphische Struktureditoren

---

## **Dissertation**

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

vorgelegt von  
**Bastian Cramer**

Paderborn, September 2010

**Datum der mündlichen Prüfung:**

6. Dezember 2010

**Gutachter:**

Prof. Dr. Uwe Kastens, Universität Paderborn

Prof. Dr.-Ing. Mark Minas, Universität der Bundeswehr München

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Grundlagen</b>	<b>7</b>
2.1	Visuelle Sprachen	8
2.2	Eigenschaften visueller Sprachen und ihre Historie	10
2.3	Programm- u. Softwarevisualisierung und visuelle Programmierung	14
2.4	Animation	17
2.5	Simulation	22
2.5.1	SIMAN	27
2.5.2	GPSS	28
2.5.3	SIMSCRIPT III	29
2.5.4	Simula	30
2.5.5	Bibliotheken zur Simulationsunterstützung	31
2.6	Programmvisualisierungssysteme	31
2.6.1	Sorting out Sorting	32
2.6.2	BALSA	32
2.6.3	TANGO	32
2.6.4	POLKA-RC	34
2.6.5	Lens	35
2.6.6	Forms/3	37
2.6.7	Pictorial Janus	37
2.6.8	Alma	39
2.7	Generatorsysteme	41
2.7.1	MetaEdit+	41
2.7.2	GenGed/Tiger	43
2.7.3	DiaGen/DiaMeta	44
2.7.4	Moses Toolsuite	47
2.7.5	AToM <sup>3</sup>	48
2.8	Relevanz im Kontext dieser Arbeit	51
2.9	Das DEViL-System	54
2.9.1	Spezifikation der abstrakten Struktur	55
2.9.2	Spezifikation der graphischen Darstellung	59
2.9.3	Zusammenhang zur Repräsentationsgraphik	62
2.9.4	Das Klassenmodell und Pfadausdrücke in DEViL	64
2.9.5	Eigenschaften generierter Umgebungen	65

<b>3</b>	<b>Simulier- und Animierbarkeit visueller Sprachen</b>	<b>69</b>
3.1	Struktur und Daten . . . . .	70
3.2	Zustand und Zustandsübergänge . . . . .	73
3.3	Repräsentation . . . . .	76
3.4	Klassifikation visueller Sprachen . . . . .	77
<b>4</b>	<b>Simulationskonzept</b>	<b>87</b>
4.1	Der Simulator . . . . .	88
4.1.1	Der erweiterte Spezifikationsprozess . . . . .	88
4.1.2	Anforderungen an den generierten Simulator . . . . .	90
4.1.3	Einbettung des Simulators in die Werkzeugkette . . . . .	92
4.1.4	Zeitliches Verhalten des Simulators . . . . .	93
4.2	Entwurf der Simulationssprache . . . . .	96
4.2.1	Imperativ vs. regelbasiert . . . . .	99
4.3	Die Simulationsspezifikationsprache DSIM . . . . .	99
4.3.1	Spezifikation der Simulationsstruktur . . . . .	100
4.3.2	Spezifikation des Simulationsverhaltens . . . . .	102
4.3.3	Spezifikation von Ereignissen . . . . .	104
4.3.4	Spezielle Funktionalitäten . . . . .	106
4.3.5	Textuell vs. visuell . . . . .	110
4.4	Verwandte Arbeiten . . . . .	111
<b>5</b>	<b>Animationskonzept</b>	<b>113</b>
5.1	Das Animationsframework . . . . .	114
5.1.1	Einordnung der Animationskomponente . . . . .	114
5.1.2	Der “ <i>interesting-events</i> ”-Ansatz . . . . .	116
5.1.3	Animation durch lineare graphische Interpolation . . . . .	117
5.1.4	Der deklarative Animationsansatz . . . . .	122
5.1.5	Berechnung der Animation . . . . .	126
5.1.6	Dynamische Animationsobjekte . . . . .	127
5.1.7	Statische Animationsobjekte . . . . .	128
5.1.8	Graphische Anknüpfungspunkte . . . . .	130
5.1.9	Informationsverlust zwischen Simulation und Animation . . . . .	131
5.2	Eigenschaften generierter Umgebungen . . . . .	135
5.3	Diskussion . . . . .	138
5.4	Verwandte Arbeiten . . . . .	139
<b>6</b>	<b>Analyse</b>	<b>141</b>
6.1	Analyse der Simulationsspezifikation . . . . .	142
6.2	Analyse des visuellen Programms . . . . .	145
6.3	Aspektororientierte Analyse . . . . .	146
<b>7</b>	<b>Evaluation</b>	<b>149</b>
7.1	Grundlagen der Usability . . . . .	151

7.2	Usability-Maße . . . . .	152
7.3	Allgemeine Methoden zur Usability Untersuchung . . . . .	153
7.4	Usability des Generators . . . . .	154
7.4.1	Implementierung von Beispielsprachen . . . . .	155
7.4.2	Feld-Beobachtung . . . . .	166
7.4.3	Expertenbewertung . . . . .	170
7.4.4	Kontrolliertes Experiment . . . . .	173
7.5	Geschwindigkeit . . . . .	177
<b>8</b>	<b>Resümee und Ausblick</b>	<b>183</b>



# 1 Einleitung

Anwendungsspezifische Sprachen werden zur Zeit in der Forschung diskutiert [28, 29]. Sie sollen das momentan vorherrschende objekt-orientierte Paradigma erweitern und so Lösungen zu spezialisierten Teilproblemen bereitstellen. Es sollen Spezialsprachen entwickelt werden, die ineinander greifen und auch in allgemeine Programmiersprachen eingebettet werden können. Durch diesen Ansatz können Probleme auf einem viel höheren Abstraktionsniveau und mit Metaphern der Domäne gelöst werden. Dies führt zu einer Entkopplung der Spezifikation von der Implementierung.

Zur schnellen Entwicklung solcher Spezialsprachen stehen diverse Generatorsysteme zur Verfügung (vgl. Abschnitt 2.7). Allen ist gemein, dass sie auf Basis einer abstrakten Beschreibung einer Sprache vollständige graphische<sup>1</sup> Entwicklungsumgebungen generieren. Graphisch bedeutet dabei, dass wesentliche Anteile der dargestellten Sprache durch Konstrukte wie Farbe, Form, Verbindung, Enthalten-sein etc. dargestellt werden. Diese Umgebungen enthalten zudem alle wichtigen Werkzeuge um Sprachinstanzen komfortabel zu manipulieren und Zielcode zu erzeugen. Generatoren für Entwicklungsumgebungen für visuelle Sprachen kapseln Expertenwissen und stellen es zur Wiederverwendung zur Verfügung. Sie machen damit Gebrauch eines der mächtigsten Programmierparadigmen der Informatik [62].

Aktuelle Generatoren für visuelle Entwicklungsumgebungen generieren in der Regel nur graphische Umgebungen, die statischer Natur sind. Visuelle Sprachinstanzen können zwar vom Benutzer sehr einfach manipuliert und erstellt werden, trotzdem besteht immer noch eine große kognitive Distanz zwischen der visuellen Beschreibung und dem, was aus der Beschreibung generiert wird: nämlich ein ausführbares Programm. Dieses Problem ist auch als "gulf of execution" bekannt [78], weshalb das Programmieren mental sehr aufwändig ist.

Gerade bei der Spezifikation von nebenläufigen Prozessen, aber auch bei der Erstellung von komplexen Algorithmen im imperativen Paradigma ist der Anwender oft bereits sehr früh überfordert. Eine dynamische Repräsentation der Zustandsübergänge mittels einer Animation kann hier beim Verifikations- bzw. Verständnisprozess helfen. Auch in anderen Bereichen, z. B. der Lehre, kann Simulation und Animation zu erhöhtem Systemverständnis führen.

---

<sup>1</sup>Ich werde in dieser Arbeit *graphisch* und *visuell* als Synonym verwenden.

Den Nutzen von Animation hat bereits Knuth 1973 in *The Art of Computer Programming* [59] beschrieben:

*“Es sollte sofort erwähnt werden, dass der Leser nicht erwarten sollte, einen Algorithmus wie ein Buch zu lesen; solch ein Versuch würde es sehr schwierig machen, zu verstehen, was dieser leistet. Ein Algorithmus muss gesehen um verstanden zu werden und der beste Weg einen Algorithmus zu lernen ist es ihn auszuprobieren. Der Leser sollte immer Stift und Papier bereithalten...”*

Dass die *dynamische Zeichengebung*, wie Schiffer sie bezeichnet [100], bisher vernachlässigt wurde, hat mehrere Gründe. Zum einen gibt es bisher nur wenig Unterstützung auf Seiten der Generatorsysteme. Oft existiert hier nur eine Beschreibung der Zustandsübergänge (meist durch Graphtransformation), jedoch nur wenig Hilfe bei der Entwicklung einer sanften Animation der Übergänge. Diese ist jedoch für ein Systemverständnis unerlässlich. Insbesondere die syntax-gerichteten Editoren verhalten sich bei der Animation nicht so wie gewünscht (vgl. Kapitel 5.1).

Zum anderen ist die dynamische Repräsentation - auf Systemebene betrachtet - sicherlich nicht einfach zu implementieren. Expertenwissen auf mehreren Gebieten ist notwendig. Oft gibt es in bestehenden Systemen (vgl. [17]) schlicht nur unzureichende Erweiterungsmöglichkeiten.

Der wichtigste Aspekt ist jedoch sicherlich, dass es keinen Konsens über eine Spezifikationsmethode gibt. Es existiert zwar eine Menge von Spezifikationsmethoden (AsmL [66], Graphtransformation [31]), jedoch keine allgemein anerkannte Vorgehensweise. Dies liegt wahrscheinlich am formalen Charakter der bisherigen Ansätze.

Ein letzter Aspekt ist, dass die Möglichkeit der Analyse einer dynamischen Repräsentation bisher unterschätzt wurde. Durch die dynamische Repräsentation ist es möglich, Fehler im Modell viel früher zu finden. Ein zeitaufwändiger Umweg über das generierte Produkt ist nicht mehr nötig, wie es ein bekanntes Generatorsystem versucht.

In dieser Arbeit soll das Generatorsystem DEViL (Development Environment for Visual Languages) zur Erzeugung von Struktureditoren für visuelle Sprachen so erweitert werden, dass die Spezifikation der Simulation und Animation einer visuellen Sprache ermöglicht wird. Dabei sind zwei Aspekte wichtig: Erstens soll das Spektrum der Sprachen, die mit Simulations- und Animationsunterstützung ausgestattet werden sollen, möglichst groß sein. Existierende Systeme beschränken häufig die Art der visuellen Sprache, z. B. auf graphartige Strukturen [33]. Dies wirkt sich direkt auf den zweiten Aspekt aus: die Spezifikationsmethodik. Sie soll besonders einfach sein. Dies ist wichtig, da Generatoren für visuelle Sprachen (VLs) besonders im *Rapid Prototyping* eingesetzt werden und die Einfachheit der Spezifikation die Akzeptanz und die Einsetzbarkeit des Systems erhöhen. Damit die Spezifikationsmethode für den Sprachentwickler einfach ist, sollen bereits bewährte



---

Konzepte des DEViL Systems übernommen und adaptiert werden. Dies wird sich im Laufe der Arbeit in mehreren Teilbereichen zeigen: die entwickelte Simulationssprache wird Strukturspezifikationskonzepte wieder aufgreifen, die Animation wird erreicht, indem der bereits bestehende deklarative Ansatz der so genannten visuellen Muster erweitert wird.

**Methodischer Beitrag dieser Arbeit** Diese Arbeit soll einen methodischen Beitrag zur Simulation und Animation im Rahmen der visuellen Sprachen darstellen. Existierende visuelle Sprachen, ebenso wie Simulationssprachen und existierende Generatorsysteme werden systematisch untersucht. Dabei zeigt diese Arbeit eine Kategorisierung von visuellen Sprachen in Bezug auf ihre Simulier- und Animierbarkeit auf. Die Sprachen werden hinsichtlich ihrer Struktur, Ausführungssemantik sowie ihrer Simulations- und Animationsstruktur untersucht. Somit kann abgeleitet werden, wann und wie eine Sprache simulier- und animierbar ist. Aus dieser Untersuchung ergibt sich eine methodische Vorgehensweise für die Spezifikation visueller Sprachen mit dynamischer Repräsentation. Diese zeichnet sich dadurch aus, dass vorhandene Konzepte wiederverwendet werden und so die Simulation als einfache Erweiterung einer bestehenden Sprachspezifikation gesehen werden kann. Wohlbekannte Konzepte und Methoden der Simulationsforschung werden übernommen und integriert. Wiederverwendung zeigt sich bei der Spezifikation der Simulationstruktur, die den bisherigen klassenbasierten Entwurf von semantischen Strukturen für VLs benutzt und diesen durch Attribute und Pfadausdrücke erweitert. Somit wird eine passende Simulationsstruktur erreicht.

Der entwickelte Ansatz zeigt, dass ein großes Spektrum visueller Sprachen simuliert werden kann. Trotzdem ist die Simulationssprache dabei einfach gehalten, da sie eine standardisierte und enge Schnittstelle zur Manipulation von Simulationsstrukturinstanzen bereitstellt, an die auch gleichzeitig die Animation anknüpfen kann.

Bereits hier sei gesagt, dass die entworfene Simulation in keiner Konkurrenz zu spezialisierten Simulatoren aus dem wissenschaftlichen Bereich stehen soll. Sie ist lediglich Mittel zum Zweck der Animation einer visuellen Sprache. Das heißt jedoch nicht, dass die Simulation in ihrem Anspruch weniger von Bedeutung ist.

Aus einer Simulation kann mit dem hier gezeigten Ansatz automatisch eine Animation abgeleitet werden. Es wird sich zeigen, dass diese generierte Animation als Standardfall häufig schon die Bedürfnisse einer animierten visuellen Sprache abdeckt. Nur in Ausnahmefällen muss durch den Sprachdesigner eine Anpassung vorgenommen werden. Dies geschieht deklarativ durch eine Bibliothek von parametrisierten Animationsmustern. Die Animation kann flexibel angepasst werden. Durch die nahezu beliebige Kombination von Bibliotheksbausteinen kann eine anspruchsvolle Animation erstellt werden. Auch hier werden bekannte und bewährte

Spezifikationsmethoden wiederverwendet. Somit ist der Spezifikationsaufwand im Verhältnis zum erzielten Ergebnis gering. Ein Sprachdesigner, der die Entscheidung fällt eine Sprache zu simulieren, bekommt eine automatisch generierte Animation nebenbei.

Zudem wird in dieser Arbeit gezeigt, dass die Animation eine formal korrekte Abbildung ihrer Simulation ist. Dies wird durch standardisierte Änderungsoperationen in der Simulation erreicht, an die die Animation anknüpft. Durch die einfache und kleine Schnittstelle werden Animationen automatisch berechnet. So kann eine erstellte Simulation direkt durch ihre Animation getestet werden.

Des Weiteren soll diese Arbeit einen Beitrag zur Analyse visueller Sprachen aufzeigen. Neben der Animation als Analysemittel für den Sprachanwender gibt es noch weitere Methoden, die es erlauben ein genaues Bild des Ausführungszustandes der Sprachinstanz zu erlangen. Dies sind u.a.:

- Logging Mechanismen zur späteren statistischen Auswertung.
- automatisch generierte Auslastungsgraphen.
- ein integrierter Debugger für die Simulationssprache. Somit können auch Algorithmenanimationssysteme nachgebildet werden.
- Ereignisgraphen.
- Modellinspektion; zur schrittweisen Analyse der Simulationsstruktur.

Die Generierung von Struktureditoren für anwendungsspezifische Sprachen ist mit dem bereits entwickelten Generator DEViL unproblematisch. Struktureditoren lassen sich in kurzer Zeit spezifizieren. Entworfen wird zunächst die abstrakte Struktur, also das Modell der Sprache. Dieses wird dann mit so genannten visuellen Mustern dekoriert, sodass eine graphische Repräsentation entsteht. Die Spezifikation findet dabei auf einem sehr hohen Abstraktionsniveau statt, das es dem Sprachentwickler erlaubt, von Implementierungsdetails zu abstrahieren und das im Generator gekapselte Expertenwissen zu nutzen. Das DEViL System generiert aus den Spezifikationen Struktureditoren, die Eigenschaften moderner Systeme wie eine Multi-Fensterumgebung, Interaktionsmechanismen und Konsistenzüberprüfungsfunktionen enthalten.

Um visuelle Sprachen mit dynamischer Repräsentation generieren zu können, werde ich im nächsten Kapitel den Bereich der Programmvisualisierung vorstellen. Hier werden sich bereits Anforderungen sowohl an die Ausführung eines Programms als auch an dessen Darstellung ergeben. Ein Blick auf verschiedene Algorithmenanimationssysteme wird zeigen, welche Stilmittel eingesetzt werden um eine anspruchsvolle Animation zu erreichen. Im Anschluss werden diverse Simulationssprachen und Bibliotheken betrachtet. Diese geben Aufschluss über die Beschreibung und Implementierung von Simulationsspezifikationen. Insbesondere wird der Begriff *Simulationsmodell* eingeführt und beschrieben. Die vorgestellten

---

Simulationssprachen betrachte ich unter besonderer Berücksichtigung des hier eingesetzten Kontextes: der Simulation und Animation visueller Sprachen. Danach betrachte ich existierende Generatorsysteme für visuelle Sprachimplementierungen. Besonders im Bereich der Graphtransformation existieren bereits Systeme, die Sprachinstanzen simulieren können. Ich werde die Systeme später nochmals in der Evaluation aufgreifen um einen Vergleich mit der hier vorgestellten Implementierung zu ziehen. Eine Vorstellung des Spezifikationskonzepts von DEViL schließt das Grundlagenkapitel ab.

Des Weiteren muss geklärt werden, wann und unter welchen Umständen eine visuelle Sprache simuliert und animiert werden kann. Ich werde zu diesem Zweck in Kapitel drei diverse Sprachen untersuchen und am Ende des Kapitels eine Klassifizierung dieser Sprachen vorstellen. Diese dient als Grundlage für das Simulationskonzept in Kapitel vier und die Animationskomponente, die in Kapitel fünf vorgestellt wird. Das Simulationskonzept wird mit dem Simulator in das allgemeine Spezifikationskonzept eingebettet. Die Animationskomponente wird die Besonderheit des vorgestellten Ansatzes zeigen: die *graphische Interpolation*. Kapitel sechs geht auf den Aspekt der Analyse von visuellen Sprachen ein. Ich werde zeigen, dass es diverse Analysekomponenten auf Spezifikationsebene gibt, die dem Sprachspezifizierer helfen, eine korrekte Simulation bzw. Animation für eine Sprachimplementierung zu entwerfen. Auch auf Anwendungsseite wird es Methoden geben, eine Animation auszuwerten. Kapitel sieben stellt eine Evaluation des entworfenen Konzepts dar. In Kapitel acht resümiere ich das Konzept und stelle weitere zukünftige interessante Forschungsaspekte vor.



# 2 Grundlagen

## Inhalt

---

<b>2.1</b>	<b>Visuelle Sprachen</b> . . . . .	<b>8</b>
<b>2.2</b>	<b>Eigenschaften visueller Sprachen und ihre Historie</b> . . . . .	<b>10</b>
<b>2.3</b>	<b>Programm- u. Softwarevisualisierung und visuelle Programmierung</b> . . . . .	<b>14</b>
<b>2.4</b>	<b>Animation</b> . . . . .	<b>17</b>
<b>2.5</b>	<b>Simulation</b> . . . . .	<b>22</b>
2.5.1	SIMAN . . . . .	27
2.5.2	GPSS . . . . .	28
2.5.3	SIMSCRIPT III . . . . .	29
2.5.4	Simula . . . . .	30
2.5.5	Bibliotheken zur Simulationsunterstützung . . . . .	31
<b>2.6</b>	<b>Programmvisualisierungssysteme</b> . . . . .	<b>31</b>
2.6.1	Sorting out Sorting . . . . .	32
2.6.2	BALSA . . . . .	32
2.6.3	TANGO . . . . .	32
2.6.4	POLKA-RC . . . . .	34
2.6.5	Lens . . . . .	35
2.6.6	Forms/3 . . . . .	37
2.6.7	Pictorial Janus . . . . .	37
2.6.8	Alma . . . . .	39
<b>2.7</b>	<b>Generatorsysteme</b> . . . . .	<b>41</b>
2.7.1	MetaEdit+ . . . . .	41
2.7.2	GenGed/Tiger . . . . .	43
2.7.3	DiaGen/DiaMeta . . . . .	44
2.7.4	Moses Toolsuite . . . . .	47
2.7.5	AToM <sup>3</sup> . . . . .	48
<b>2.8</b>	<b>Relevanz im Kontext dieser Arbeit</b> . . . . .	<b>51</b>
<b>2.9</b>	<b>Das DEViL-System</b> . . . . .	<b>54</b>
2.9.1	Spezifikation der abstrakten Struktur . . . . .	55
2.9.2	Spezifikation der graphischen Darstellung . . . . .	59
2.9.3	Zusammenhang zur Repräsentationsgraphik . . . . .	62
2.9.4	Das Klassenmodell und Pfadausdrücke in DEViL . . . . .	64
2.9.5	Eigenschaften generierter Umgebungen . . . . .	65

---

Im folgenden Kapitel soll die Basis zum Verständnis dieser Arbeit geschaffen werden. Visuelle Sprachen werden vorgestellt und ihre Eigenschaften untersucht. Danach werden Programmvisualisierung, Programmanimation sowie Algorithmenanimationssysteme betrachtet, da sie sowohl eine Simulations- als auch eine Animationskomponente enthalten. Auch andere Generatorsysteme im Umfeld der visuellen Sprachen werden anschließend untersucht. Zum Schluss des Kapitels wird auf die Architektur sowie das Spezifikationskonzept von DEViL eingegangen. Dies ist notwendig, da die Ergebnisse dieser Arbeit, Spezifikationsaspekte des Generators wieder aufgreifen.

### 2.1 Visuelle Sprachen

Graphische Darstellungen haben in der Softwareentwicklung eine lange Historie. So konnte Haibt in den 1950er Jahren aus Flußdiagrammen *Assembler*- und *Fortran*-Code generieren [1]. Diese Diagramme können bereits als "visuelle Sprache" zur Darstellung von Software bezeichnet werden. Um jedoch den Begriff "visuelle Sprache" zu definieren, ist es zunächst wichtig zu definieren, was "visuell" eigentlich bedeutet.

Der Begriff "visuell" haftet heute aus marketingtechnischen Gründen auch Systemen an, die im hier später definierten Sinn nicht wirklich visuelle Sprachelemente benutzen. VisualBasic und Visual C++ sind so zum Beispiel Programmierumgebungen, die einen im Wesentlichen textuellen Editor erweitern und Methoden integrieren, die das Programmieren erleichtern. Zu nennen sind hier das automatische Ergänzen von Methodennamen, umfangreiche Refactoringfunktionalitäten oder eine direkte Hilfe zu API-Funktionen. Das eigentliche Erstellen der Software erfolgt allerdings häufig noch mit einer herkömmlichen textuellen Sprache. Visuelle Sprachen (VL) wie z. B. UML setzen dabei auf vorwiegend graphische Notationselemente wie Farbe, Form, Layout etc., sie schränken jedoch meist auch die Anwendungsdomäne stärker ein. Sie werden also domänen- oder auch anwendungsspezifischer. Dabei gilt in der Regel, dass  $\text{Ausdrucksstärke} \times \text{Domänengröße}$  konstant ist [79]. Schiffers [100] Definition von *visuell*:

*"Visuell ist die Bezeichnung für jene Eigenschaft eines Objekts, durch die mindestens eine Information über das Objekt, die für das Erreichen eines Handlungsziels unverzichtbar ist, nur durch das visuelle Wahrnehmungssystem des Menschen gewonnen werden kann."*

Schiffer meint damit, dass das visuelle Wahrnehmungssystem des Menschen als "unverzichtbares" Instrumentarium notwendig ist. Textuelle Spezifikationen schließt er nicht aus, verweist jedoch darauf, dass diese primär das verbale Verarbeitungssystem des Menschen ansprechen. Price et al. gehen in [10] bei der Definition von "visuell" noch einen Schritt weiter und verweisen auf eine Definition von "visuell" im Oxford Dictionary, wobei visuell sich nicht auf die Wahrnehmung mit dem Auge beschränkt,

sondern eher ein mentales Bild im Geiste formen soll. Diese Definition des Begriffs "visuell" ist die Umfassendste, denn sie schließt auch andere Sinne mit ein. So gibt es auch Versuche den Gehörsinn in der Softwareentwicklung zu berücksichtigen [11].

Myers [71] definiert eine visuelle Sprache als ein System, das Graphik einsetzt. Diese Definition ist sehr schwammig und bezieht sogar Systeme wie VisualBasic ein, die eine ikonische Oberfläche anbieten, deren inhärente Programmierstruktur jedoch rein textuell ist.

Schiffers Definition ist auch hier genauer:

*"Eine visuelle Sprache ist eine formale Sprache mit visueller Syntax oder visueller Semantik und dynamischer oder statischer Zeichengebung."*

Schiffers Definition geht insbesondere auf die visuelle Syntax ein und meint damit eine graphische Notation der Grundsymbole. Mit visueller Semantik ist der Laufzeitzustand der visuellen Objekte gemeint, der ebenfalls visuell sein kann. Dynamische Zeichengebung bedeutet dabei die Darstellung von flüchtigen Vorgängen, z. B. das Aufblinken eines Objekts, das sich verändert.

Schmidt hat in [101] den Begriff der visuellen Sprache noch verfeinert. Er unterscheidet zwischen echt visuellen Sprachen, deren "wesentliche Teile" eine graphische Notation haben und textuellen Sprachen. Insbesondere definiert Schmidt visuelle Sprachen als eine Obermenge der textuellen Sprachen. Er nennt Python [47] als Beispiel, das - neben einer rein textuellen Syntax - Einrückung als semantisches Äquivalent zu Blöcken betrachtet. Ich werde seine Definition beibehalten und mit visuellen Sprachen *echt* visuelle Sprachen bezeichnen. Textuelle Sprachen verstehe ich als Sprachen mit hauptsächlich "verbaler" Syntax, wie Schiffer es auch definiert.

Die von Schiffer angesprochene dynamische Zeichengebung, also die Darstellung flüchtiger Vorgänge werde ich im Folgenden als *Animation* bezeichnen. Eine Animation ist somit die Visualisierung eines Zustandes oder einer Zustandsänderung einer Programminstanz einer visuellen Sprache durch eine geeignete graphische Darstellung. Ihr Zweck ist die Darstellung diskreter Zustandsübergänge, die ansonsten nicht erfasst werden könnten oder das Hinweisen auf interessante Stellen in einer Programmausführung. Am häufigsten eingesetzt wird die Animation um eine Bewegung einer Struktur darzustellen. Dabei visualisiert die Animation kontinuierliche Transformationen der Programminstanz.

Die Transformation eines visuellen Programms bedeutete bisher im statischen Sinne, dass eine Manipulation durch einen Benutzer stattfindet. Der Benutzer editiert die Struktur durch vordefinierte Funktionen. In Struktureditoren, wie sie in dieser Arbeit betrachtet werden, sind dies: das Einfügen, Löschen, Klonen und das Bewegen von Objekten bzw. das Ändern von Attributen eines Objekts. Diese Manipulation wird innerhalb einer Simulation nun durch einen Zustandsübergangsautomaten ausgelöst. Dieser Automat wird später als *Simulator* bezeichnet. Die

Simulation ist dann der Vorgang der durch den Simulator ausgelöst wird. Die kontinuierliche Interpolation zwischen diskreten Simulationsschritten ist die Animation.

Die Simulation visueller Sprachen und insbesondere die Animation der Simulationsschritte wird heute noch nicht häufig eingesetzt [89]. Dies erscheint aufgrund der oben genannten Vorteile von Animation, und der Tatsache dass dies bereits sehr früh erkannt wurde, als unverständlich. Es ist aber erklärlich aufgrund des mehrdimensionalen Schwierigkeitsgrades. Es müssen Techniken aus der Simulation und aus der Visualisierung zusammengeführt und angewendet werden.

In den nächsten Abschnitten sollen visuelle Sprachen und ihre Eigenschaften in Bezug zu herkömmlichen allgemeinen Sprachen gesetzt werden. Insbesondere gehe ich auf den Nutzen von Animation ein.

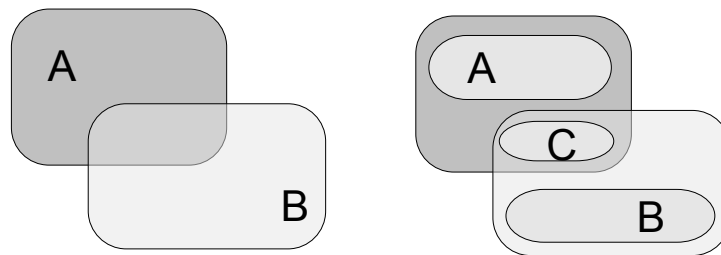
## 2.2 Eigenschaften visueller Sprachen und ihre Historie

Schiffer nennt eine ganze Reihe von Vorteilen visueller Sprachen gegenüber rein textuellen. So werden vom Menschen Bilder viel schneller als Texte verstanden. Laut Kognitionspsychologie werden verbale Informationen sequentiell verarbeitet, wohingegen visuelle parallel verarbeitet werden können. Das führt dazu, dass mit einer visuellen Notation mehr Informationen gleichzeitig wahrgenommen werden können.

Die Einführung von piktogrammorientierten Systemen mit natürlichen Metaphern zur Steuerung des Rechners (mit Desktop-Metapher) hat die Bedienung des Computers wesentlich vereinfacht; sie sind heute nicht mehr aus dem Alltag wegzudenken. Gerade bekannte visuelle Metaphern führen dazu, dass sich ein künstliches System besser verstehen lässt. Für abstrakte Konzepte gibt es häufig jedoch keine Bilder oder die Benutzung von Bildern kann zu Missverständnissen führen. In Abbildung 2.1 sind Harels Higraphen [45] als ein Beispiel zu sehen, wie Missverständnisse bei der visuellen Notation von Mengen entstehen können. Laut Harel ist die ursprüngliche Mengennotation mehrdeutig. Woraus besteht die Menge  $A$ ? Sich überlagernde Mengen in der Abbildung links können nicht eindeutig definiert werden. Harel schlägt eine alternative Darstellung vor (Abbildung 2.1 rechts). Dieses kleine Beispiel verdeutlicht bereits die Interpretationsproblematik bei visuellen Darstellungen, obwohl nur wenige graphische Konstrukte benutzt werden.

Shu [108] bezeichnet Bilder als mächtige Kommunikationsmittel, da es keine Sprachbarrieren gibt. Allerdings gibt es trotzdem Probleme bei der Interpretation, die sich hauptsächlich aus der Erfahrung eines Betrachters ergeben. So ist der Verständnisprozess eines visuellen Ausdrucks bei einem geübten Menschen wesentlich schneller





**Abbildung 2.1:** Mißverständnisse bei visueller Notation: Woraus besteht die Menge A?

als bei einem ungeübten. Außerdem benötigen visuelle Ausdrücke nicht unbedingt ein tiefes Verständnis von zu Grunde liegenden computerspezifischen Methoden. Die hohe Dichte an Informationen in einem Bild und die direkte Manipulation von Objekten können die Arbeit mit visuellen Ausdrücken wesentlich vereinfachen. Trotzdem ist die geometrische Darstellung von Software problematisch, weil sie keine Struktur im Raum hat. In visuellen Darstellungen ist aber gerade die räumliche Anordnung häufig wichtig. Konzepte wie räumliche Positionierung, "Enthaltensein" oder "Überlappung" tragen hier in der Regel eine Semantik. Diese kann jedoch bei der Darstellung als Graphik zu Platzproblemen führen, was die Informationsdichte wiederum verringert. Auch können komplexe Abläufe ebensolche Darstellungen zur Folge haben.

Der oft zitierte Satz "Ein Bild sagt mehr als 1000 Worte" scheint bei der visuellen Programmierung nicht immer zu stimmen, denn unterschiedliche Betrachter interpretieren Bilder anders. Häufig hängt die Interpretation einer visuellen Notation auch vom Kontext ab. Eine einheitliche Übereinkunft, um visuelle Programme eindeutig zu machen, besteht nicht [86].

Diagramme können oft nicht sofort erfasst werden, da die visuelle Notation erst erlernt werden muss. Es gibt keine Regeln, wie Diagramme zu lesen sind. Falls mehrere Diagramme zu einer visuellen Sprache gehören, ist auch nicht unmittelbar klar, wo der Einstiegspunkt ist. In textuellen Sprachen ist dies jedoch recht klar bestimmt (durch die Leserichtung) und auch Mehrdeutigkeiten können anhand des Kontextes erkannt werden.

Visuelle Sprachen sind somit besonders von Vorteil wenn Metaphern einer be-

stimmten Domäne benutzt werden oder wenn graphische Objekte strengen allgemein gültigen Konventionen unterliegen (z. B. das "Heimat"-Piktogramm in graphischen Bedienoberflächen, das immer den Heimatordner des jeweiligen Benutzers referenziert). Eine gute Gestaltung, der Einsatz von sekundärer Notation und Mechanismen, um relevante Daten zu zeigen, machen also eine gute visuelle Sprache aus.

Schmidt geht sogar noch weiter: Wenn visuelle Sprachen eine Obermenge der textuellen Sprachen sind, dann gibt es für jedes Problem eine visuelle Sprache, die mindestens genauso gut ist wie jede textuelle Sprache, die das Problem beschreibt. Insbesondere verweist Schmidt auf die Layoutfreiheiten visueller Sprachen, die bestimmte Strukturen besser abbilden und auf zusätzliche informelle Zusatzinformationen, die eingebettet werden können. Zusätzlich können Interaktionsmechanismen durch geeignete Symbolik direkt in visuelle Repräsentanten eingebaut werden. Beispiele sind "Griffe" zum Anfassen visueller Objekte oder ein "+"-Symbol zum Ausklappen bestimmter Bereiche.

Inkrementelle Entwicklung ist ein weiterer Vorteil von visuellen Sprachen. Objekte können leicht nachträglich manipuliert oder modifiziert werden. So kann in einem UML-Klassendiagramm sehr einfach eine neue Klasse eingefügt werden, indem andere Objekte entsprechend verschoben werden. In einer rein textuellen Darstellung ist dies nicht so einfach möglich. Oft müssen mehrere Dateien geöffnet werden und Fragmente von verschiedenen Positionen kopiert werden. Dann muss Platz geschaffen werden, was in der Regel mehr Aufwand erzeugt. Die Hauptnachteile einer visuellen Sprache werden im Allgemeinen im Platzbedarf und beim Aufwand der Konstruktion gesehen. So benötigt ein visuelles Programm - bedingt durch Verschnitt - wesentlich mehr Platz auf dem Bildschirm. Dies führt dazu, dass in der Regel weniger Informationen gleichzeitig dargestellt werden können als bei einer textuellen Sprache. Die Konstruktion eines Programms mittels einer visuellen Sprache ist - bedingt durch die großen Mauswege - recht aufwändig. Gute Tastaturinteraktionsmechanismen für den versierten Benutzer sind somit unumgänglich.

Ein wesentlicher, wenn nicht gar der wichtigste Vorteil von visuellen Sprachen ist hingegen, dass auf einem extrem hohen Abstraktionsniveau gearbeitet werden kann. Syntaktische Elemente einer textuellen Sprache werden durch graphische Objekte mit Attributen repräsentiert. Das hohe Abstraktionsniveau erlaubt eine Entkopplung von der Zielsprache, die ggf. sogar austauschbar ist. Spezielle Editoren können dem Anwender bei der Konstruktion eines visuellen Programms helfen.

Gerade in speziellen Domänen ist ein Einsatz sinnvoll, wenn die Anzahl der Sprachkonstrukte eng gefasst ist oder der Anwenderkreis auf Nicht-Experten eingeschränkt ist.

**Historie** Bereits in den 50er Jahren wurden graphische Systeme zur automatischen Generierung von Flußdiagrammen aus Assembler- und Fortran-Code erstellt. Der Nutzen solcher Flußdiagramme wurde bereits 1947 durch von Neumann und Goldstein gezeigt. 1966 wurde das wohl erste graphische Programmiersystem von Sutherland entwickelt. Der "Graphical Program Editor" erlaubte die Spezifikation von Programmen ähnlich der von Hardwareplänen. Die Programme konnten dann interpretiert werden. Die Entwicklung von Pygmalion 1975 [109] durch David C. Smith war ein weiterer Meilenstein. Die Idee war, Programme durch Vorzeigen geeigneter Beispiele zu entwerfen, anstatt konkret den Programmablauf zu modellieren.

Ende der 70er, Anfang der 80er Jahre wurden Tabellenkalkulationssprachen entwickelt, die weite Verbreitung fanden. VisiCalc war das erste Tabellenkalkulationsprogramm, das auf einer 2-D-Darstellung basierte. Dieses System war so erfolgreich, dass sich in den nächsten Jahren etliche Arbeiten mit tabellen- und formularbasierten Ansätzen beschäftigten. Für komplexere objektorientierte oder komponentenbasierte Ansätze sind diese jedoch untauglich.

In den 80er Jahren wurde bereits viel im Bereich der visuellen Programmierung geforscht. Visuelle Programmierung sollte auch den Endanwender in die Lage versetzen Programme zu erstellen, so die Hoffnung.

Seit den 90er Jahren werden vermehrt kommerzielle Werkzeuge zur visuellen Programmierung angeboten, was an der höheren Leistungsfähigkeit von Computersystemen und deren graphischen Bedienoberflächen liegt. So erfreut sich LabVIEW [124] im Bereich der Hardware-Spezifikation großer Beliebtheit.

Heute ist ein neuer Trend absehbar: Die Einbettung spezialisierter domänen-spezifischer Sprachen in allgemeine Hochsprachen [28, 29] sowie der Einsatz visueller Sprachen im Rapid Prototyping.

Visuelle Sprachen mit dynamischer Repräsentation gehen noch einen Schritt weiter. Sie unterstützen den Benutzer dabei, nicht nur ein syntaktisch oder semantisch korrektes Programm zu erstellen, sondern zusätzlich noch die in der Programmbeschreibung inhärenten Zustandsübergänge zu visualisieren. Die Programminstanz ist nun nicht mehr nur statisch, sondern kann direkt ausgeführt werden. Die Kluft zwischen Programminstanz und Programmausführung, die jeder Anwender einer visuellen Sprache mental überbrücken muss, wird kleiner [78].

Die direkte Ausführbarkeit einer visuellen Sprache hilft auch im Rapid Prototyping. Hier kann von Systemeigenschaften abstrahiert und auf einem hohen Niveau spezifiziert werden. Die Simulation auf derselben Ebene führt zu einem direkterem Systemverständnis. Auftretende Fehler können direkt exploriert und entfernt werden. Somit muss nicht erst das spezifizierte System generiert, übersetzt

und getestet werden, was eine erhebliche Zeitersparnis bedeutet.

In [82] werden noch einige weitere Vorteile von dynamischen Darstellungen genannt. Diese sind:

1. die Demonstration sequentieller Aktionen in prozeduralen Aufgaben.
2. die Simulation von Modellen mit komplexem Verhalten.
3. die explizite Darstellung von eigentlich unsichtbarem Verhalten.
4. die Illustration von Verhalten, das verbal schlecht zu beschreiben ist.
5. die Bereitstellung einer visuellen Analogie für ein abstraktes Konzept.
6. die Ausrichtung der Aufmerksamkeit des Anwenders auf einen spezifischen Aspekt.
7. die Validierung eines spezifizierten Verhaltens.

Park und Hopkins [82] zeigen Forschungsergebnisse zum Einsatz von dynamischen Darstellungen in unterschiedlichen Medien. Die oben genannten Punkte zeigen sowohl Einsatzzwecke von Animation auf, z. B. in der Simulation (2) und auch in der Lehre (4, 5). Dazu werden in den folgenden Kapiteln dieser Arbeit Simulationswerkzeuge untersucht, aber auch Werkzeuge zur Vermittlung des Verständnisses von Algorithmen beschrieben.

Des Weiteren zeigen die Punkte bereits Eigenschaften, die ein gutes Animationsystem bereitstellen sollte: Nämlich die Fähigkeit einer Animation Abläufe (seien sie in Wirklichkeit sogar unsichtbar (3)) in geeigneter Weise darstellen zu können (5) und dem Betrachter im richtigen Moment wichtige Teilaspekte der Simulation vermitteln zu können (6).

## 2.3 Programm- u. Softwarevisualisierung und visuelle Programmierung

Schon früh hat man sich im Rahmen der Softwarevisualisierung mit Algorithmenanimation beschäftigt. Erste Systeme wurden bereits Anfang der 1980er Jahre entwickelt. Die folgende Einordnung der Softwarevisualisierung orientiert sich stark an der Klassifikation von Price et al. [1], da diese sehr ausführlich ist und sie noch weiter geht als die von Myers [71], der Programmvisualisierung nur anhand von zwei Achsen unterscheidet. Zum einen, ob Code oder Daten visualisiert werden und zum anderen, ob die Visualisierung dynamisch oder statisch ist. Die folgende Klassifikation von Price beruht auf der Untersuchung einer Reihe von Algorithmenanimationssystemen bzw. visuellen Programmierwerkzeugen, berücksichtigt jedoch keine domänenspezifischen Sprachen (DSLs). Die Klassifikation ist jedoch so allgemein, dass sie auch für DSLs benutzt und ggf. erweitert werden kann.

Nach Price et al. wird Softwarevisualisierung in der Literatur häufig missverstanden und mit den Begriffen *“visuelle Programmierung“*, *“Programmvisualisierung“*, *“Algorithmenanimation“* und *“Programming-by-example“* in Zusammenhang gebracht, wobei alle Begriffe austauschbar untereinander verwendet werden. Dies ist jedoch falsch und eine genauere Betrachtung der Begriffe ist nötig.

Softwarevisualisierung ist demnach *“das Handwerk Typographie, Graphikdesign, Animation und Kinematographie mit moderner Mensch-Maschine-Interaktion und Computer-Graphik Technologie einzusetzen um beides, das menschliche Verständnis und die effektive Benutzung von Computer-Software voranzutreiben“*.

Programmvisualisierung hingegen ist die Visualisierung des aktuellen Programmcodes oder dessen Datenstruktur in entweder statischer oder dynamischer Form. Der Begriff Programmvisualisierung wurde früher auch für Softwarevisualisierung benutzt.

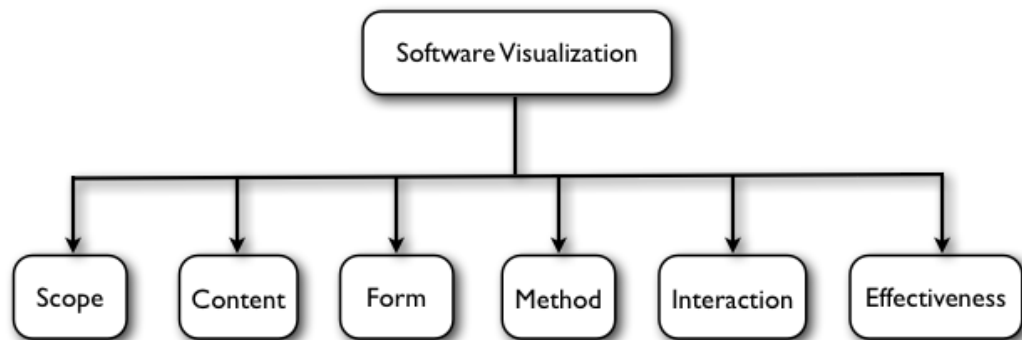
Der wesentliche Unterschied zwischen visueller Programmierung und Programmvisualisierung ist, dass visuelle Programmierung es erlauben soll, Programme einfacher zu spezifizieren. Softwarevisualisierung soll Programme und visuelle Spezifikationen leichter verständlich machen.

Programmvisualisierung ist jedoch mit dem Programmcode, also einem niedrigen Abstraktionslevel konnotiert, wohingegen Softwarevisualisierung eher mit dem Algorithmenlevel verbunden ist. Myers Klassifikation unterscheidet zwischen Code und Datenvisualisierung und zwischen statischer und dynamischer Animation. Eine statische Code-Visualisierung ist somit z. B. ein Prettyprinter wie Baeckers SEE Program Visualizer [5], der Quellcode für das menschliche Auge aufbereitet und Gebrauch von Einrückungen und Kommentaren als sekundäre Notation macht.

Ein Beispiel für eine statische Datenvisualisierung wäre die Darstellung einer verknüpften Liste mit ihren aktuellen Werten. Eine animierte Code-Visualisierung wäre die Darstellung eines Debuggers, der anzeigt, welche Programmzeile gerade ausgeführt wird. Eine dynamische Datenvisualisierung wäre somit eine statische Datenvisualisierung mit veränderlichen Inhalten, also z. B. eine Algorithmenvisualisierung wie das Vertauschen von Elementen bei einem Sortieralgorithmus.

Shu unterteilt visuelle Sprachen in der Klassifikation aus [108] in drei Ausprägungen: Ihrem Anwendungsgebiet, dem Ausmaß an rein visuellen Konstrukten, die die Sprache benutzt und wie groß die Entfernung zwischen Problem und Problemlösung durch das Programm ist.

Die Abbildung zwischen Problem und Problemlösung betrachten auch Cox et al. in [22]. Dort wird nicht nur die Entfernung betrachtet, sondern die Art, wie



**Abbildung 2.2:** Framework zur Klassifizierung von Softwarevisualisierung

Programminformationen dargestellt werden. Dies rückt die zu untersuchende Problematik in den Vordergrund. Bereits hier sei gesagt, dass diese Arbeit und das erweiterte DEViL Framework alle Klassifikationen nach Cox abdecken kann. Cox unterscheidet folgende Abbildungen:

- *Direct*: ist die unmittelbare Abbildung eines Aspekts des Programms auf ein Bild. Es wird keine weitere Abstraktion hinzugefügt, z. B.: beim Sortieren von Listenelementen wird die Größe eines Elements auf ein Rechteck entsprechender Größe abgebildet.
- *Structural*: in der Simulation versteckte oder zusätzliche Daten werden dargestellt, z. B. das Pivotelement im Quicksort-Algorithmus.
- *Synthesized*: hierbei werden Daten gezeigt, die nicht unmittelbar aus der Simulation abgeleitet werden können, aber aus zusätzlichen Berechnungen ermittelt werden können, z. B. die akkumulierte Ausführungshistorie des Sortierens.
- *Analytical*: hierbei wird von der Repräsentation differenziert und abstrakte Eigenschaften der Struktur oder des Verhaltens des Programms dargestellt, z. B. Aussagen wie "alle Elemente kleiner als Index  $i$  sind korrekt sortiert".
- *Explanatory*: hierbei wird versucht das Verständnis des Beobachters zu erweitern indem zusätzliche visuelle Hinweise in das Programm eingearbeitet werden, z. B. die weiche Animation oder zusätzliche textuelle Hinweise, die die Aktion erläutern.

In [97] heben Cox et al. noch hervor, dass besonders Abbildungen der letzten Kategorie ein hohes Maß an Flexibilität erfordern. Programmzustand, Zustandsübergänge und deren Animation liegen weit voneinander entfernt.

Price et al. schlagen ein Framework zur Klassifizierung von Softwarevisualisierungssystemen vor (Abb. 2.2). Aus diesem Framework lassen sich direkt Anforderungen für simulierbare DSLs ableiten. So sollte eine gute Software-Visualisierung in allen von Price et al. genannten Teilbereichen hohe Werte erzielen. Prices Framework ist jedoch so umfangreich, dass eine detaillierte Darstellung an dieser Stelle den Rahmen

der Arbeit sprengen würde.

Stasko et al. nennen im Rahmen ihres Algorithmenanimationssystems *DANCE* [112] weitere Anforderungen, die nicht nur für Algorithmenanimation gelten können:

- *Ease of Design*: Animationsdesigner können von der Simulation abstrahieren und beliebige Animationen erstellen. Low-Level-Details sind dabei für den Anwender transparent.
- *Program development support*: Es existieren diverse Mechanismen um Programmierfehler zu vermeiden und zusätzliche Hilfen um Simulationen zu entwickeln bzw. Fehler zu verstehen.
- *Rapid-Prototyping*: Die Animation (in *DANCE*) wird visuell spezifiziert, somit ist sehr schnelles Feedback möglich.
- *Ability to fine-tune*: Überschreiben von Code um Animationen zu erreichen, die "genauso und nicht anders aussehen" müssen.

Dass graphische Modelle und deren Animation das Verständnis einer Simulation unterstützen ist bereits seit langem bekannt [99]. Trotzdem ist die Simulation im Bereich der DSLs noch nicht weit verbreitet [89]. Vielleicht auch, weil es durchaus schwierig ist, den Ablauf von DSLs einfach zu spezifizieren und für einen ggf. anderen Nutzerkreis verständlich und übersichtlich aufzubereiten. Die Animationskomponente wurde hier häufig vernachlässigt.

## 2.4 Animation

In der Programmvisualisierung wird der Begriff der Animation häufig nur implizit eingesetzt, in dieser Arbeit ist er jedoch sehr wichtig. Es muss genau betrachtet werden, ob es sich lohnt eine Animation zu erstellen. Diese ist nicht immer nötig und kann oft auch sehr teuer sein [85]. Mit automatisch generierten Animationen ist diese Designentscheidung jedoch nicht mehr so wichtig. Der Mehrwert einer Animation muss genau abgewogen werden und liegt in der zusätzlichen Erfahrung, die aus einer Animation gewonnen werden kann [95]. Für Lehr- und Lernzwecke kann eine visuelle Simulation eine gute Basis sein. Auch in Systemen in denen Aktionen gleichzeitig ausgeführt werden, z. B. in Petri-Netzen oder in der nebenläufigen Programmierung ist eine Animation besonders geeignet.

Programmvisualisierung bzw. Animation ist jedoch kein Allheilmittel [7] für das Verständnis von Software. Eine Programmvisualisierung ist nicht unmittelbar verständlich und muss ebenfalls gelernt werden, da auch hier sehr stark vom eigentlichen Problem abstrahiert werden kann und sich die Darstellung vom zu Grunde liegende Problem stark unterscheiden kann. Eine Animation ist nicht zwingend für jedes Problem geeignet, sie stellt auch die Problematik nicht unmittelbar

einleuchtend dar. Sie muss der Zielgruppe angepasst werden und so ist es oft von Vorteil, wenn die Animation von Experten des Problembereichs entworfen wird. Einfache Spezifikationstechniken werden somit unumgänglich.

Der Einsatzzweck einer Animation muss genau überlegt werden. Simulationen sind oft sehr groß. Der animierte Teil, der von Interesse ist, kann jedoch sehr klein sein. Hier kann Animation als Validierungsmittel für die Simulation eingesetzt werden. Manchmal ist auch der Modellierer der Simulation der Entscheidungsträger und eine Animation erübrigt sich. Andere Simulationssysteme sind evtl. stochastischer Natur, bei der Zwischenstände nicht von Belang sind. Hier sind mehrere Simulationsläufe, die Log-Ausgaben erzeugen relevanter als eine Animation. Oft wird eine Animation auch erstellt um einen anderen Nutzerkreis, z. B. Entscheidungsträger anzusprechen.

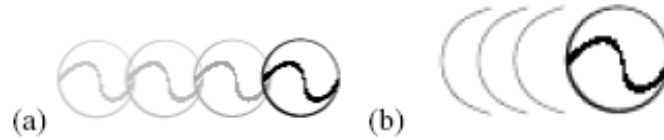
Animation ist also dann von Vorteil, wenn aus ihr ein echter Mehrwert gewonnen werden kann. Dies kann zu Ausbildungszwecken, zur Validierung oder auch für Präsentationen der Fall sein. So wird beispielsweise in [76] eine primitive Animation eingesetzt um eine natürliche Sprache zu lehren.

Die Animationskomponente (als Teil einer Programmvisualisierung oder einer Simulation) sorgt für die graphische Animation von Objekten in all ihren Facetten: Von einfachen Farbänderungen bis zu komplexen animierten Pfaden in unterschiedlichen Geschwindigkeiten. Die Simulation sorgt für die zeitliche Abstimmung und die Triggerung der Animation. Sie teilt der Animation mit, was wann animiert werden soll. Die Animationskomponente kann man sich dabei als eine Sammlung abstrakter Funktionen vorstellen, die auf graphischen Objekten operieren. Die Sammlung an sich besteht aus einer Menge von Animationstypen, die vordefiniert sind. Die Simulation füllt diese abstrakten Funktionen mit konkreten Werten. Sie bildet ihren Zustand auf die Animation ab.

Nach Baecker [4] sollte Animation dabei unterstützen komplexe Strukturen und Funktionen darzustellen und zu untersuchen. Animation hat dabei unterschiedliche Ausprägungen in Form und Funktion. Prinzipiell soll sie eine kausale Verbindung zwischen einem alten und einem neuen diskreten Zustand schaffen, der ohne Animation nicht unmittelbar klar ist [18]. Durch diesen hinweisenden Effekt muss der Anwender nicht darüber nachdenken was sich wo geändert hat und kann sich der eigentlichen Aufgabe widmen. Bei der visuellen Programmierung kommt hinzu, dass der Programmierer bereits mit visuellen Konstrukten arbeitet, diese Konstrukte jedoch nur statisch sind. Das Programm, das er spezifiziert, ist jedoch in der Regel dynamisch. D. h. er muss trotzdem ein mentales Bild des dynamischen Ausführungsverhaltens seines Programms vor Augen haben. Warum sollte eine Animation ihm also nicht dabei helfen, die Kluft zwischen Programm und Programmausführung zu verringern?

In [73] wird zunächst zwischen *Basic Animation*, *Motion Animation* und *Cartoon*





**Abbildung 2.3:** Sogenannte “Effect Lines” (Schattenrisse) um Geschwindigkeit zu verdeutlichen, aus [55]

*Animation* unterschieden. Mit *Basic Animation* sind einfache Animationen wie Blinken oder das Wechseln von Farbe bzw. Bitmap Graphiken gemeint. Diese Art der Animation hat in der Regel nur einen hinweisenden Effekt und soll häufig nur das Auge des Betrachters auf eine Stelle lenken in der etwas geschieht. Das Ändern von Bitmap Graphiken zeigt häufig den Zustand des Objekts an.

*Motion Animation* stellt die Bewegung von Objekten dar. Häufig wird ein Pfad zwischen zwei so genannten Schlüsselbildern interpoliert. Die Bewegung kann allerdings auch dem “Path-Transition“ Paradigma folgen und eine beliebige Punkteliste darstellen, über die sich ein Objekt bewegt. Minas stellt in [69] eine so genannte “parametrisierbare Bewegung“ vor, in der die Bewegung von  $(x_0, y_0)$  nach  $(x_1, y_1)$  nicht durch zwei Schlüsselbilder, sondern durch Gleichungssysteme der Form

$$x = (x_1 - x_0) \lambda + x_0$$

$$y = (y_1 - y_0) \lambda + y_0 \quad \lambda \in [0, 1]$$

beschrieben wird. “Programmierbare Bewegungen“ erweitern dieses Konzept noch durch algorithmische Spezifikationen.

*Cartoon Animation* übernimmt klassische Effekte der Trickfilmanimation, die die Animation somit einfacher verständlich und flüssiger erscheinen lässt. Physikalische Effekte der echten Welt werden in das System übernommen. So stoppen Objekte nicht abrupt in einer Bewegung, sondern pendeln sanft aus. Weitere Effekte sind z. B. ein Vibrieren von Objekten, die im Begriff sind sich zu bewegen. Auch “Anticipation“, d. h. Objekte bewegen sich zunächst in die entgegengesetzte Richtung bzw. “Exaggeration“, also die Übertreibung sind beliebte Stilmittel um anspruchsvolle Animationen zu erreichen. In [55] werden zusätzliche Linien und Schattenrisse hinter sich schnell bewegendem Objekten gezeichnet (Abb. 2.3). Das Morphen von Farben zur Darstellung des Alters von Objekten geht in eine ähnliche Richtung wie die Effect-Lines. Hier tragen graphische Konstrukte semantische Informationen. Die Bedeutung von Farben in der Animation heben Nardi et al. in [77] hervor. Die

Effect-Lines führen später zum Konzept der statischen Animationsobjekte (vgl. Abschnitt 5.1.7). All diese Varianten der Animation dienen dazu, die dynamische Zeichengebung leicht verständlich zu machen und die Simulation in geeigneter Weise darzustellen.

Lasseter et al. zeigen in [63] einen guten Überblick zur Animation anhand der Illustration von Filmen. Sie identifizieren im Wesentlichen folgende Animationsarten:

- *Squash and Stretch* definiert die Masse oder Festigkeit eines Objekts. Objekte bleiben während einer Bewegung in der Regel nicht starr. Sie werden gedehnt und gestaucht. Dadurch erreicht man, dass Geschwindigkeit besser wahrgenommen wird. Betrachter können den Objekten besser folgen.
- *Timing* also die Geschwindigkeit einer Aktion ordnet ihr Bedeutung zu. Je nachdem wie schnell eine Aktion ausgeführt wird, kann sie eine unterschiedliche Semantik tragen. So kann gleichartigen Objekten durch ein unterschiedliches Timing ein unterschiedliches Gewicht zugeordnet werden. Das schwerere Objekt wird nur langsam beschleunigt, während das leichte Objekt sehr schnell seine Endgeschwindigkeit erreicht.
- *Anticipation* beschreibt die Vorbereitung auf eine Aktion. Sie wird benutzt um den Anwender auf einen Ort hinzuweisen, an dem in Kürze etwas passieren wird. Diese Art der Animation führt später in der Arbeit zum Konzept des Triggers von Animationsobjekten.
- *Staging* bedeutet eine Aktion so zu präsentieren, dass sie unmittelbar einleuchtend erscheint. Dies wird häufig dadurch erreicht, dass zu einer wichtigen Stelle hingezoomt wird oder die Beleuchtung entsprechend angepasst wird, sodass das Objekt quasi im Rampenlicht steht.
- *Follow Through and Overlapping Action* wird dazu verwendet, eine Aktion zu beenden und zu einer anderen überzuleiten. Dass eine Aktion abrupt zur nächsten Aktion führt ist in der Realität eher selten zu sehen, ein Überlappen von Aktionen führt zu einer harmonischeren Animation.
- *Straight ahead action and pose-to-pose action* sind zwei gegensätzliche Ansätze zur Entwicklung von Animationen. Eine Variante basiert auf einem ad-hoc Entwicklungsansatz, wodurch Spontanität entsteht, während die andere Variante planvoller ist.
- *Slow In and Out* bezieht sich auf die zeitliche Abstimmung in Bewegungen. Objekte bewegen sich nicht konstant sondern werden langsam beschleunigt und abgebremst.
- *Arcs* sind geschwungene Bewegungspfade für Animationsobjekte. Bewegungen erscheinen so natürlicher.
- *Exaggeration* also die Übertreibung ist gerade in Cartoon Animation ein beliebtes Stilmittel.

Automation level	Technique
1	frame-wise
2	keyframe
3	algorithmic / task-level Animation
4	actor-based Animation

**Abbildung 2.4:** Technische Klassifikation von Animation

- *Secondary Action* sind Aktionen, die als Resultat von anderen Aktionen entstehen. Sie werden eingesetzt um ein Interesse zu erhöhen und der Animation eine realistische Komplexität zu geben. Die Bewegung einer Figur als Kernidee wird beispielsweise durch eine zweitrangige Aktion, wie die Mimik unterstützt.

Als letzten Aspekt nennen Lasseter et al. *Appeal* also den Anreiz. Eine Animation wird erst dann gut, wenn ihre Visualisierung sorgsam ist und die Idee, die sie vermitteln soll, dem Publikum klar wird.

Paelke beschreibt in [81] noch einige weitere Techniken um gute Animationen zu erstellen. Sie basieren darauf, geeignete Kameraperspektiven oder auch mehrere unterschiedliche Sichten auf denselben Sachverhalt zu wählen. Außerdem kann eine Szenerie geeignet beleuchtet werden um eine Idee zu transportieren. Diese eher auf die Animation von dreidimensionalen Szenen ausgelegten Animationsarten werde ich hier nicht genauer beschreiben, da sich diese Arbeit im Rahmen der zweidimensionalen visuellen Sprachen bewegt.

Nach Bertin [8] sind so genannte “Marks” die Grundelemente auf denen Visualisierungen aufbauen. Sie sind durch ihre Position und sechs weitere Attribute (Größe, Orientierung, Grauwert, Farbe, Text sowie Form) charakterisiert. Die Definition der “Marks” charakterisiert auch die Repräsentation von DEViLs Strukturobjekten sehr gut. Auch diese werden durch einfache Primitive zu komplexen graphischen Darstellungen zusammengestellt. In [73] ist eine knappe Übersicht gegeben, wie Animation technisch klassifiziert wird:

Bei der *frame-wise*-Animation (Level 1) werden die einzelnen Bilder der Animation in einem gesonderten Editor von Hand produziert. Ein Player akzeptiert diese Bilder und spielt sie dann ab. Bei der *keyframe*-Animation (Level 2) werden Schlüsselbilder vorgegeben. Die Animation wird dann durch Interpolation generiert. Die Bewegung kann dabei einem bestimmten Pfad folgen. Keyframe-Animation kann auch durch Attribute der Objekte parametrisiert werden. Bei der algorithmischen oder *task-level*-Animation (Level 3) wird definiert, was animiert werden soll und nicht wie dies geschehen soll. Die algorithmische Animation wird dabei durch eine Sequenz von parametrisierten Transformationen bestimmt. Bei der task-level Animation wird die Bewegung in abstrakten Aufgaben beschrieben wie “bewege Objekt von A nach B”. Die *actor-based*-Animation folgt dem objektorientierten Ansatz. Die Akteure führen dabei die Animation selbständig aus. Vorher werden Regeln definiert, die aussagen,

in welcher Weise Zustandsübergänge erfolgen sollen.

Ein weiterer Aspekt ist die Nachbildung realer physischer Abläufe. Hierzu ist es notwendig, dass physikalische Gegebenheiten, wie zum Beispiel Schwerkraft, Wind oder Federkräfte geeignet und einfach in die Animation mit eingebunden werden. Die physikalische Animation wird allerdings nur sehr selten von Animationsentwicklern eingesetzt [25] und wird in dieser Arbeit zunächst keine Rolle spielen. Außerdem wird sie häufig nur in sehr speziellen Anwendungsszenarien eingesetzt, die in der Regel anspruchsvolle Simulationen sind und nur am Rande etwas mit visuellen Sprachen zu tun haben.

Animationen können schnell sehr komplex und groß werden. Hier kann es notwendig sein weitere Techniken einzusetzen. Zum Beispiel Kamerafahrten zu Punkten in der Animation an denen gerade Aktionen stattfinden [81] (falls im aktuellen Fenster der entsprechende Teil nicht sichtbar ist). Weitere alternative Kameraperspektiven, semantisches Zoomen, transparente Objekte, Ebenen oder ganz andere visuelle Darstellungen (Fisheye, 3D,...) können verwendet werden [87].

Insgesamt lassen sich die in Abbildung 2.5 beschriebenen Grundtypen der Animationsarten definieren, die einen Ausschnitt aus dem Spektrum der Möglichkeiten sind. Prinzipiell sind hier dem phantasievollen Animator keine Grenzen gesetzt.

## 2.5 Simulation

Bisher wurde die Animation, also die dynamische Zeichengebung betrachtet. Ihr zu Grunde liegt ein Algorithmus, der die Animationen ausführt. Systematisch betrachtet wird eine Eingabe (das Programm) durch eine vorgegebene Semantik ausgeführt. Ausführung bedeutet, dass eine Struktur oder ein Modell existiert, in dem Zustände gespeichert und modifiziert werden können. Diese Zustandsübergänge triggern dann Animationen. Im Folgenden sollen Simulation und Simulationsstrukturen genauer betrachtet werden. Begriffe wie Simulationsmodell und ereignisbasierte Simulation werden eingeführt. Danach folgt ein Überblick auf existierende Simulationswerkzeuge und ihre Bedeutung im Kontext dieser Arbeit. Dazu gehören Simulationsprachen, allgemeine Hochsprachen mit Simulationsunterstützung sowie Bibliotheken zur Simulation.

Simulation ist die Disziplin ein Modell eines realen oder eines nur theoretisch existierenden Systems zu entwerfen, dieses Modell auszuführen und es zu analysieren [36]. Im Gegensatz zu mathematischen Modellen, die Endzustände berechnen, bilden Simulationen Dynamik ab. Simulation wird interdisziplinär angewendet. Wetter- und  $n$ -Körper-Simulationen sowie Anwendungen in der künstlichen Intelligenz und Virtual Reality zeigen die vielfältigen Facetten der Simulation. Simu-

Operation	Beschreibung
Objekte	
create	Objekt wird neu erstellt, es wird dabei ausgehend von einem Punkt bis zur endgültigen Größe vergrößert.
delete	Objekt wird bis zur Unkenntlichkeit verkleinert und dann entfernt (fade-out)
toFront	Objekt wird auf die oberste Ebene verlagert. Ggf. Durchdringung anderer Objekte via Transparenz.
toBack	Objekt wird auf die unterste Ebene verlagert. Ggf. Durchdringung anderer Objekte via Transparenz.
blink	Objekt blinkt
colorChange(Color)	Objekt verändert Farbe (abrupt oder kontinuierlich)
rotate(degree)	Objekt wird rotiert
scale(delta)	Objekt wird skaliert
substitute(X,Y)	Objekt X wird durch Objekt Y ersetzt.
shake	Objekt vibriert
Bewegung	
move(a, b)	direkte Bewegung eines Objekts von a nach b
moveSlowInAndOut(a, b)	Bewegung von a nach b, am Anfang/Ende langsamer
moveArcs(a,b)	Objekt wird in einer Kurvenform von a nach b bewegt.
parametricMove(f)	Objekt wird durch Funktion f bewegt.
programmedMove(alg)	Objekt wird durch Algorithmus bewegt.
Komposition	
$Anim^x$	Animation x Mal wiederholen
$AnimX \bullet AnimY$	Animation X vor Animation Y ausführen
$AnimX \Theta AnimY$	Animation X und Y gleichzeitig ausführen

Abbildung 2.5: Animationsarten

lationen die auf besonders großen Datenmengen arbeiten, z. B. Klimasimulationen, erfordern häufig den Einsatz von Großrechnern.

Eine Simulation führt häufig zu einem erhöhten Systemverständnis, auch kann ein komplexes reales System oft nicht analytisch erfasst werden bzw. das reale System ist nicht einfach zu manipulieren. Häufig existiert das reale System noch gar nicht. Dies ist beispielsweise bei der Simulation von Fabrikanlagen der Fall, bei der die Fertigungsstraßen schon vor dem eigentlichen Baubeginn auf Engpässe hin untersucht werden können. Simulationen werden auch angewendet, um eine bereits analytisch vorliegende Lösung zu verifizieren oder um Vorhersagen über ein zukünftiges Verhalten zu machen.

Das Simulationsmodell ist die abstrakte logische und mathematische Repräsentation eines Systems, das die Bezüge zwischen Objekten beschreibt. Durch das wiederholte Ausführen von Operationen auf dem Simulationsmodell mit unterschiedlichen Parametern können Erfahrungen mit dem System gesammelt werden. Das Simulationsmodell stellt dabei die Abbildung eines Modells in einer Struktur dar, die es erlaubt, bestimmte Aspekte detailliert zu untersuchen, d. h. es abstrahiert von nicht relevanten oder unnötigen Details. Diese Vereinfachung birgt jedoch Risiken, denn ein vereinfachtes Simulationsmodell  $\mathcal{A}'$  über ein System  $\mathcal{A}$  ist nur gut, wenn die Antworten, die  $\mathcal{A}'$  über  $\mathcal{A}$  gibt, dieselben sind, die  $\mathcal{A}$  geben würde.

Die Korrektheit des Simulationsmodells ist demnach für eine aussagekräftige Simulation besonders wichtig. Ein zu niedriges Detaillevel im Modell kann zu wertlosen Simulationsergebnissen führen. Auf der anderen Seite führt ein großer Detaillierungsgrad zu einem undurchschaubaren Modell, sowie zu langen Simulationslaufzeiten.

Simulationen benötigen den Zufall um realistische Modelle abbilden zu können. Dazu können Zufallsvariablen bestimmter Verteilungen eingesetzt werden. Bei einem stochastischen Simulationsmodell ist eine sorgfältige Analyse des Modells besonders wichtig. Hier können verlässliche Aussagen oft nur getroffen werden, wenn die Simulation häufig wiederholt wird, sodass eine Sättigung der Parameter eintritt.

Kontinuierliche Modelle beschreiben Zustandsänderungen über die Zeit. Ihr Zustandsübergangsmodell wird häufig durch Gleichungssysteme beschrieben. Diskrete Simulationsmodelle hingegen unterteilen die Zeit in gleich große Abschnitte. Das Zustandsübergangsmodell ist hier häufig durch einen Automaten beschrieben. Die Zeitunterteilung der diskreten Variante kann dazu führen, dass über einen langen Zeitraum keine Zustandsänderung eintritt. Dies wird durch die ereignisorientierte Simulation umgangen. Hier werden Zustandsänderungen durch Ereignisse, die in einer sortierten Liste gespeichert werden, realisiert. Die Zeit wird dabei immer zum nächsten Ereignis fortgeschaltet. Ereignisse können

dabei wieder Ereignisse auslösen, die dann auch ggf. weiter in der Zukunft in die Ereigniswarteschlange eingeordnet werden. Durch dieses Prinzip lassen sich auch elegant Benutzerinteraktionen oder Ausnahmebehandlungen als spezielle Ereignisse in das Simulationsmodell integrieren.

In [36] werden mehrere Simulationsmodelle diskutiert. Interessant sind das deklarative, das funktionale sowie das räumliche Simulationsmodell. Beim deklarativen Simulationsmodell wird das Modell anhand von Ereignissen und deren Modifikationen am Zustandsmodell beschrieben. Diese Zustandsänderungen werden durch DFAs oder Markovmodelle bzw. Ereignisgraphen beschrieben. Der deklarative Ansatz hat jedoch den Nachteil, dass sehr häufig sehr viele Zustände beschrieben werden müssen.

Beim funktionalen Modell wird die Simulation durch das Verknüpfen von Funktionen beschrieben. Eine Funktion hat dabei einen oder mehrere Eingabe-Parameter und einen Ausgang. Die Darstellung kann hier graphisch durch Blöcke erfolgen, die die Funktionen repräsentieren. Wird der Fokus auf eine ereignisorientierte Simulation gelegt, so führt dies zu einer funktionalen Sichtweise.

Das räumliche Simulationsmodell fügt der Anordnung von Objekten Semantik hinzu. Objekte können andere Objekte in der Nähe beeinflussen. Das räumliche Simulationsmodell wird häufig durch Regeln ausgedrückt. Regeln haben jedoch den Nachteil, dass sie sehr schnell unübersichtlich werden. Das wiederum führt zu einer schlechten Wartbarkeit [111]. Insbesondere können komplexe Beziehungen zwischen Objekten, die unter Umständen weit entfernt liegen nicht modelliert werden.

Des Weiteren wird häufig zwischen prozessorientierten Simulationen und ereignisorientierten Simulationen unterschieden. Bei ereignisorientierten Simulationen (Abb. 2.6) ist der Modellierungsansatz auf den Fluss von Simulationsobjekten ausgelegt. An interessanten Zeitpunkten werden Ereignisse ausgelöst. Die Eigenschaften von Objekten werden als Attribute gespeichert. Abbildung 2.6 zeigt dabei die Simulation einer einfachen Fertigungsstraße. Fertigungsteile durchlaufen dabei mehrere Stationen an denen sie verzögert werden. Während sich ein Fertigungsteil in einer Station befindet, können andere Teile auf diese Station nicht zugreifen. Sie werden in eine Warteschlange eingefügt. Nachdem die Station durchlaufen ist, verlässt das Fertigungsteil die Abteilung und ein weiteres Teil wird aus der Warteschlange genommen. Jeder Block kann hier als Funktion angesehen werden, der Ereignisse auslöst.

Bei ereignisorientierten Simulationen werden Simulationsentitäten häufig durch Klassen gekapselt, die durch Nachrichten miteinander kommunizieren (Abb. 2.7). Die Abbildung zeigt denselben Fertigungsprozess. Hier sind die einzelnen Stationen jedoch durch Nachrichten dargestellt. Die prozessorientierte Simulation mag einfacher erscheinen, erlaubt aber weniger Programmkontrolle. Für jedes Simu-

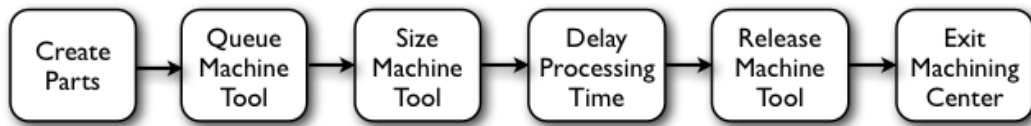


Abbildung 2.6: Ereignisorientierte Simulation (aus [96])

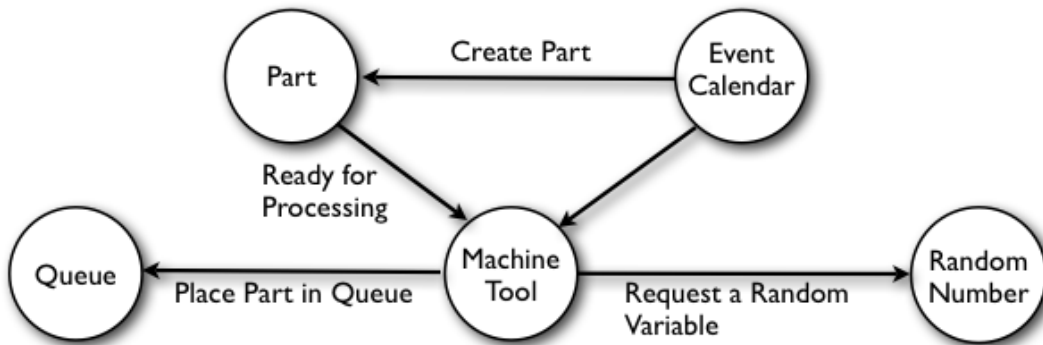


Abbildung 2.7: Prozessorientierte Simulation (aus [96])

lationsobjekt im System muss ein eigener Prozess mit eigener interner Ablaufzeit erstellt werden.

Beim diskreten Simulationsansatz wird die Zeit in kleinste Einheiten unterteilt und diskret weitergeschaltet. In Kombination mit Ereignissen spricht man häufig vom *“next-event to time advance”* Ansatz. In der Simulation werden dabei an bestimmten Zeitpunkten Ereignisse ausgelöst, die Simulationsobjekte modifizieren können. Diese Ereignisse werden in eine prioritätsbasierte Warteschlange eingeordnet und die Zeit bis zum nächsten Ereignis weitergeschaltet.

Es sollen im Folgenden einige Simulationsumgebungen auf ihre Konzepte und ihre Relevanz in Bezug auf die Simulation visueller Sprachen hin untersucht werden. Dabei ist zwischen Simulationsbibliotheken und Simulationssprachen zu unterscheiden. Simulationsbibliotheken sind in einer herkömmlichen Programmiersprache geschrieben und sollen den Benutzer dabei unterstützen, das eigene Programm um Simulationskonstrukte zu erweitern. Im Rahmen dieser Arbeit betrachte ich nur objektorientierte Simulationsbibliotheken.

Die hier betrachteten Simulationssprachen sind speziell auf die diskrete Simulation ausgelegt, sie sind quasi domänenspezifisch. Sie enthalten Spezialkonstrukte, die es dem Benutzer besonders einfach machen sollen, eine Simulation zu entwerfen. Die aus den Simulationssprachen generierten Programme können entweder direkt oder mit einem Interpreter ausgeführt werden. Einige Simulationssprachen, wie zum Beispiel Simula, sind jedoch ebenso mächtig, wie eine herkömmliche Programmiersprache.



Simulationssprachen werden häufig bei transportorientierten Systemen, wie Fertigungsstraßen, eingesetzt. Simulationsumgebungen basieren auf Simulationssprachen, haben jedoch meist eine visuelle Spezifikation und bieten eine Analyseumgebung.

### 2.5.1 SIMAN

SIMAN [83] ist eine Simulationssprache zur Modellierung von diskreten, kontinuierlichen oder hybriden Modellen. Diskrete Systeme können in SIMAN entweder prozess- oder ereignisorientiert entworfen werden. Ein kontinuierliches Simulationsmodell wird definiert, indem Zustandsvariablen Differentialgleichungen zugeordnet werden. In kleinen Zeitintervallen werden die Zustandsvariablen aktualisiert. SIMAN ist spezialisiert auf die Untersuchung von Fertigungsstraßen und Arbeitsabläufen. Es bietet ein Framework das zwischen der Modellkomponente und dem Experimentrahmen unterscheidet. Das Modell beschreibt die physikalisch vorhandenen Elemente, wie Maschinen, Arbeiter, Transporter etc. und deren logische Verknüpfung. Der Experimentrahmen definiert die Bedingungen unter denen das Modell ausgeführt wird.

Für die Beschreibung des Modells steht neben einer textuellen Sprache eine visuelle Blockdiagrammsprache zur Verfügung. In den Blockdiagrammen werden einzelne Stationen modelliert. Jede Station wird graphisch durch ein benanntes Rechteck dargestellt. Unterhalb der Blockdefinition stehen die Parameter der Blockfunktion. Die Simulationsobjekte "durchlaufen" dann das Blockdiagramm von oben nach unten und werden durch Prozessfunktionen oder das Warten in einer Schlange verzögert. Pfeilverbindungen zeigen den Fluss der Simulationsobjekte durch das Diagramm an. Simulationsobjekte können mittels spezieller Konstrukte zwischen verschiedenen Blockdiagrammen wechseln und simulieren so zum Beispiel den Transport von Waren.

In SIMAN können Ressourcen beliebiger Größe modelliert werden. In der Regel werden Ressourcen mit einer vorangehenden Warteschlange modelliert, sodass Simulationsobjekte warten müssen, wenn die Ressource belegt ist. Für die Modellierung von Schlangen stehen Konzepte wie LIFO, FIFO etc. bereit. Des Weiteren ist es in SIMAN möglich, Simulationsobjekte anhand probabilistischer Funktionen zu erstellen bzw. zu löschen. Auch Verzweigungen zwischen einzelnen Stationen sind anhand von Wahrscheinlichkeitsfunktionen möglich. Durch so genannte "Macro Submodels" können generische Stationen definiert werden, die mit konkreten Parametern quasi instanziiert werden können.

Im Rahmen der Arena Simulationsumgebung erlaubt es SIMAN eine einfache Animation zu definieren. Abbildung 2.8 zeigt ein Blockmodell für die Simulation

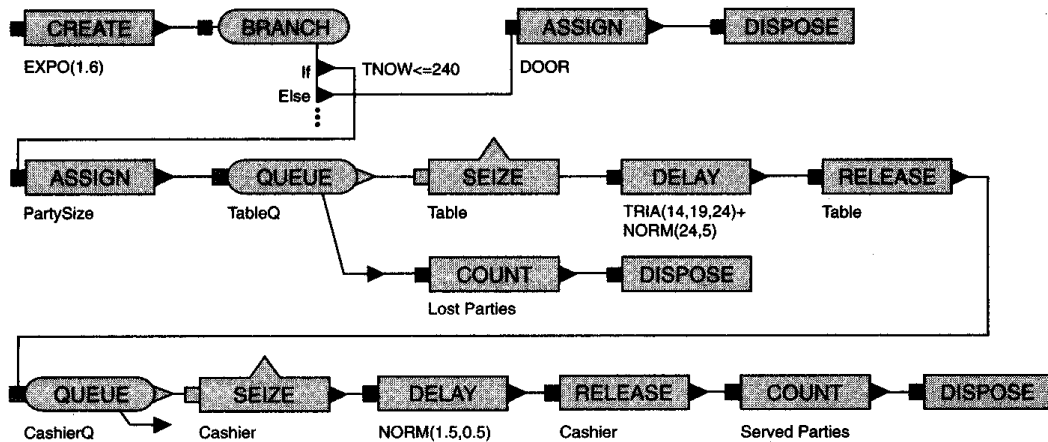


Abbildung 2.8: Blockmodell in SIMAN (aus [84])

eines Restaurants. Kunden kommen dabei entsprechend einer Exponentialverteilung im Restaurant an. Kunden, die vor neun Uhr ( $TNOW \leq 240$ ) ankommen, werden wieder weggeschickt. Die restlichen Kunden werden in einem DELAY-Block verzögert - sie essen - die Zeit ist normalverteilt mit 24 Minuten und einer Standardabweichung von fünf Minuten. Nach dem Essen gehen Kunden zum Kassierer. Der Kassiervorgang ist normalverteilt mit 1,5 Minuten und einer Standardabweichung von einer halben Minute.

## 2.5.2 GPSS

GPSS [106] wird seit den frühen 1960er Jahren entwickelt und ist heute mit den Nachfolgern GPSS/H und GPSS World für Windows [90] eine der am weitest verbreiteten Simulationssprachen. GPSS erlaubt die diskrete Simulation von Systemen auf Warteschlangensbasis und ähnelt sehr stark der Simulationssprache SIMAN. Jedoch gibt es in GPSS keine visuelle Modellierung von Stationen mittels Blockdiagrammen. Interessant an GPSS ist die flexible Modellierung von Warteschlangen, die schrittweise Simulation und die Definition von Haltepunkten (ähnlich zu denen in Debuggern). Außerdem erlaubt es GPSS Störungsszenarien zu spezifizieren, wie sie in Fertigungsanlagen entstehen können. Bei GPSS durchlaufen Transaktionen ebenfalls wie in SIMAN den Modellraum. Zusätzlich können Transaktionen geklont werden, z. B. wenn ein Auftrag mehrere Unteraufträge nach sich zieht. Die geklonten Transaktionen werden dann wieder zusammengeführt und warten gegenseitig aufeinander. Dies kommt zum Beispiel in Werkstätten vor, wenn ein Inspektionsauftrag weitere gleichzeitige Untersuchungen in verschiedenen Teilbereichen des Werkstücks nach sich zieht. GPSS erlaubt es, so genannte Matrizen zur Ressourcenbelegung zu definieren. So können räumliche Bezüge zwischen einzelnen Ressourcen modelliert werden. Dies tritt zum Beispiel bei Parkplätzen auf, wobei ein

```

GENERATE          7,6                               ;nächster Kunde kommt
Serv_Bob          TRANSFER        BOTH,Serv_Bob,Serv_Alice
SEIZE            Bob                               ;warten...
ADVANCE          8,4                               ;Haare schneiden...
RELEASE          Bob                               ;Haarschnitt fertig
Serv_Alice        TRANSFER        ,finito
SEIZE            Alice                            ;warten
ADVANCE          15,10                             ;Haare schneiden...
RELEASE          Alice                            ;Haarschnitt fertig
finito           TERMINATE      1                 ;der Kunde geht

```

Listing 2.1: Spezifikation mit GPSS aus [90]

LKW mehrere PKW Plätze beansprucht und eine Fragmentierung die Folge ist.

Das GPSS Skript in Listing 2.1 modelliert ein Frisörgeschäft mit zwei Frisören (Bob und Alice), die gleichzeitig Kunden bedienen. Ein Kundenobjekt wird dabei durchschnittlich alle sieben Minuten (mit einer Standardabweichung von sechs Minuten) generiert und an einen der Frisöre weitergeleitet. Dort wird dann gewartet bis der Kunde an der Reihe ist und die Haare geschnitten werden können. Im Anschluss verlassen die Kunden das Geschäft. Eine Anweisung in GPSS besteht dabei immer aus vier Teilen: dem Label, der Operation, den Parametern und einem optionalen Kommentar.

### 2.5.3 SIMSCRIPT III

SIMSCRIPT [94] ist eine objekt-orientierte Simulationssprache und eignet sich ebenfalls sowohl für diskrete als auch kontinuierliche Simulation. Gleichzeitig ist SIMSCRIPT auch als allgemeine Sprache zu benutzen. Die Syntax ist selbstdokumentierend. SIMSCRIPT erlaubt die Definition von Klassen mit Objektattributen und -methoden und Vererbung. Es können Warteschlangen mit unterschiedlichen Bedienungsvarianten definiert werden. Simulationsobjekte können gleichzeitig beliebig vielen Mengen angehören. Methoden können als so genannte Prozessmethoden deklariert werden, diese werden dann, mittels eines "schedule" Aufrufs, beliebig weit in der Zukunft ausgeführt. Des Weiteren können Prozessmethoden unterbrochen und wieder neu gestartet werden. Zu statistischen Zwecken gibt es spezielle Ausdrücke, die zum Beispiel die Auslastung eines Variablenwerts über die Simulationszeit bestimmen. Listing 2.2 zeigt die Definition einer Klasse `Vehicle` mit einer Prozessmethode `Trip`. Es wird ein Objekt `Car` vom Typ `Vehicle` angelegt und die Methode `Trip` nach zwei Tagen ausgeführt. In der Prozessmethode wird zunächst gewartet und anschließend die Wartezeit ausgegeben.

```
begin class Vehicle

    every Vehicle
        has a Trip process method
    define Trip as a process method
        given ''miles to travel and
            ''average speed in miles per hour
            2 double arguments
        yielding ''duration of trip in hours
            1 double argument
    end

    schedule a Trip(Car) given 200, 50 in 2 days

    methods for the Vehicle class

    process method Trip
        given Distance, Avg_Speed
        yielding Actual_Duration

        define Duration, Start_Time
            as double variables

        Duration = Distance /
            uniform.f(Avg_Speed-5, Avg_Speed+5, 1)

        Start_Time = time.v
        wait Duration hours
        Actual_Duration = (time.v -Start_Time) * hours.v
    end
end
```

**Listing 2.2:** Spezifikation mit SIMSCRIPT III

### 2.5.4 Simula

Simula [24] wurde in den sechziger Jahren am Norwegian Computing Center entwickelt. Mit Simula sollten zeitdiskrete Systeme simuliert und entwickelt werden. Der Focus liegt dabei nicht nur auf Fertigungsstraßen wie bei SIMAN oder GPSS sondern ist breiter gefächert, da Simula auch als General Purpose Sprache entworfen ist. Simula, das auf Algol 60 basiert, entwickelte sich zur ersten objektorientierten Programmiersprache. Dynamisch instanziierbare Koroutinen, auch Prozesse genannt, enthalten lokale Prozeduren und Daten. Koroutinen laufen quasi parallel ab, jedoch zu jedem Zeitpunkt ist nur genau eine Koroutine aktiv. Sie können auf die Daten anderer Koroutinen zugreifen und in beliebig vielen Mengen enthalten sein. Die enthaltenen Klassen SIMSET und SIMULATION bieten Unterstützung für die diskrete ereignisorientierte Simulation. Mit der "activate process [at|before|after|delay]"-Klausel können Prozesse zu beliebigen Zeitpunkten oder verzahnt mit anderen Prozessen gestartet werden.

### 2.5.5 Bibliotheken zur Simulationsunterstützung

Anders als die bisher angesprochenen Simulationssprachen, die eine eigene, ggf. spezialisierte Syntax (GPSS, SIMAN) haben, sind Simulationsbibliotheken in herkömmliche Programmiersprachen wie Java oder C/C++ geschrieben. Sie bieten somit eine hohe Flexibilität, jedoch mit dem Nachteil, dass sie nicht so spezialisierte Konstrukte enthalten und ein zusätzlicher herkömmlicher Programmrahmen geschrieben werden muss.

**SimJava** [61] ist eine Java Bibliothek zur diskreten ereignisbasierten Simulation. Es können statische Netzwerke mit aktiven Elementen, wie sie beispielsweise bei Schaltungen zu finden sind, modelliert werden. Simulationsobjekte, die in einem Java Thread laufen, werden durch so genannte Ports miteinander verknüpft. Über die Ports können die Simulationsobjekte dann miteinander kommunizieren, indem sie Nachrichten bidirektional verschicken. Das Verhalten der einzelnen Simulationsobjekte wird durch das Überschreiben einer geerbten Methode erreicht. Zusätzlich gibt es eine Reihe vordefinierter Methoden um Nachrichten zu verschicken, Log-Ausgaben zu erzeugen oder den aktiven Thread zu wechseln. Konstrukte für Warteschlangen, Ressourcenbelegung oder Zufallsvariablen existieren nicht.

**Silk** [46] ist eine weitere auf Java basierende objektorientierte Simulationssprache, die prozessorientierte Modellierungskonzepte bereitstellt. Silk enthält eine Reihe von Funktionen für Warteschlangen, die auch von SIMAN bekannt sind. Jedes Simulationsobjekt läuft hier in einem eigenem Java Thread ab und erbt von einer speziellen Klasse "Entity", die spezielle Methoden für Simulationsobjekte bereithält.

Die kommerzielle C/C++ Simulationsbibliothek **CSIM** [107] enthält weitaus mächtigere Konstrukte als SimJava. CSIM erlaubt die Modellierung von diskreten Systemen in einer prozessorientierten Sichtweise. Dabei hat CSIM ein ausgefeiltes Ressourcen Konzept: Ressourcen können so zum Beispiel auch nur teilweise von Prozessen belegt werden. Neben der Definition von Ereignissen, auf die Prozesse hören, können auch Mailboxen modelliert werden, in die Prozesse Nachrichten legen. Neben der Erzeugung von Zufallszahlenströmen hat CSIM umfangreiche Funktionen zur Reportgenerierung.

## 2.6 Programmvisualisierungssysteme

Werkzeuge zur Programmvisualisierung und besonders diejenigen, die graphische Darstellungen dynamischer Natur erzeugen, sind besonders interessant, da sie sowohl eine Simulations- als auch eine Animationskomponente enthalten. Ich möchte hier zunächst auf Systeme für die Algorithmenanimation eingehen. In diesen Systemen wird besonderer Wert auf die Animation gelegt, da das Ziel war, Verständnis für

einen bestimmten Algorithmus beim Anwender zu erreichen. Auf Seiten der Spezifikation herrscht hier besonders das Prinzip vor, den Code zu annotieren. Da bereits Anfang der 1980er Jahre mit diesem Forschungszweig begonnen wurde, existiert eine große Menge (ca. 100) an Systemen. Im folgenden Abschnitt sollen einige interessante Systeme vorgestellt werden.

### 2.6.1 Sorting out Sorting

Die erste Arbeit im Bereich der Softwarevisualisierung war der 1980 entstandene Film "Sorting out Sorting" von Baecker [3], der verschiedene Sortieralgorithmen zeigt, die animiert miteinander verglichen werden. SOS diente hauptsächlich Lehrzwecken und war Vorbild für die Animation in vielen weiteren Algorithmenanimationssystemen.

### 2.6.2 BALSА

BALSА, der "Brown University Algorithm Simulator and Animator" [12] war 1984 das erste interaktive System zur Algorithmenanimation, das die damaligen fensterbasierten Systeme ausnutzte. Balsa war in C geschrieben und konnte Pascal Code animieren. Es unterstützt mehrere Sichten, in denen auch unterschiedliche Algorithmen gleichzeitig betrachtet werden konnten. Die aktuelle Codezeile wurde graphisch hervorgehoben. BALSА beruht auf dem Prinzip der "Interesting Events", wobei der Benutzer bestimmte Stellen im Algorithmus identifiziert, die er für die Visualisierung wichtig hält. Diese Stellen werden dann dem Animationswerkzeug übergeben und entsprechend graphisch dargestellt. Das System der "Interesting Events" hat den Nachteil, dass der Animator den Algorithmus kennen muss. BALSА I lief auf Apollo Rechnern mit schwarz-weiß Displays, BALSА II auf Apple Rechnern mit Farbdisplays und Soundausgabe zu Ereignissen.

### 2.6.3 TANGO

TANGO [114] wurde 1989 von Stasko ebenfalls wie BALSА an der Brown Universität entwickelt. Es führt das so genannte "Path-Transition"-Paradigma ein, das für eine gleichmäßige und kontinuierliche Animation sorgt. Das Path-Transition-Paradigma wird in vielen anderen Animationssystemen eingesetzt und erlaubt eine einfache Spezifikation, ohne viel Code zu schreiben. In TANGO werden zunächst fundamentale Operationen des Algorithmus identifiziert, dann Animationen angefertigt und schließlich die Algorithmenoperationen auf ihre graphische Repräsentation abgebildet. Das Ziel von TANGO ist dabei, eine mächtige Graphikbibliothek bereitzustellen, die einfach zu verstehen ist und auch von Nicht-Experten angewendet werden kann. Die Animationskomponente von TANGO animiert im Wesentlichen Bilder, die

Image	Location	Path	Transition
Create	Create	Create	Rotate
Locate	X	Load	Scale
	Y	Store	Example
Modify	Equal	Length	Motion
		Dx	Distance
		Dy	Concatenate
		MakeType	Iterate
		Null	Compose
		Copy	AddHead
		Color	AddTail
		Extend	DeleteHead
		Interpolate	DeleteTail

Abbildung 2.9: Operationen auf TANGO Datentypen

hinsichtlich ihrer Größe, Position und Farbe verändert werden. Dazu stehen vier abstrakte Datentypen zur Verfügung: "Images" - Graphiken, die dargestellt werden, "Locations" - Positionen, die von Bildern eingenommen werden und "Paths" - Pfade, die die Bild-Positionen modifizieren. Animation wird dadurch erreicht, dass bestimmten Bildern während einer Transition ein Start- und ein Endpunkt zugewiesen wird. Anhand des entstehenden Pfades wird das Bild bewegt. Dies ist das "Path-Transition"-Paradigma. TANGO unterstützt neben der Bewegung von Graphikobjekten noch eine ganze Reihe anderer Transitionen: So gibt es neben Graphik Primitive wie Linien, Rechtecken, Kreisen und Text noch zusammengesetzte Bildobjekte, die es erlauben, mehrere andere Primitive zu kombinieren.

TANGO Datentypen und Operationen können der Tabelle aus Abbildung 2.3 entnommen werden.

Das Pfad Argument wird dabei je nach Transitionstyp interpretiert. Handelt es sich z. B. um eine "alter visibility" Transition, so wird für jedes Koordinatenpaar die Sichtbarkeit einmal an- und ausgeschaltet. Transitionen können auch vom Typ "Iteration", "Concatenation" und "Composition" sein.

Ein Beispiel zur konkreten Spezifikationssprache von TANGO ist in Listing 2.3 zu sehen. Das Beispiel erzeugt ein Rechteck und einen Kreis. Das Rechteck wird an eine neue Position bewegt, während der Kreisradius vergrößert wird. Problematisch am TANGO/XTANGO-Ansatz ist, dass Transitionen zwar hintereinander und nebenläufig modelliert werden können, aber nicht beliebig verschränkt nebenläufig. Dies ist besonders in Programmen mit mehreren Threads problematisch, die zu beliebigen Zeitpunkten gestartet werden können. POLKA-RC soll dies umgehen.

```
TANGO_IMAGE rect, circ;
TAGO_LOC center, to Loc;
TANGO_PATH path1, path2;
TANGO_TRANS trans1, trans2, doBoth;
double delta[5] = {0.1, 0.1, 0.1, 0.1, 0.1};

rect = TANGOimage_create(TANGO_IMAGE_TYPE_RECTANGLE,
    0.2, 0.2, 1, TANGO_COLOR_RED, 0.1, 0.3, 0.5);
circ = TANGOimage_create(TANGO_IMAGE_TYPE_CIRCLE, 0.6,
    0.8, 1, TANGO_COLOR_BLUE, 0.05, 0.0);
center = TANGO_image_loc(rect, TANGO_PART_TYPE_C);
toLoc = TANGoloc_create(0.2, 0.7);
path1 = TANGOpath_motion(center, toLoc,
    TANGO_PATH_TYPE_STRAIGHT);
path2 = TANGpath_create(5, delta, delta);
trans1 = TANGO_trans_create(TANGO_TRANS_TYPE_MOVE,
    rect, path1);
trans2 = TANGOtrans_create(TANGO_TRANS_TYPE_RESIZE,
    circ, path2);
doBoth = TANGO_trans_compose(2, trans1, trans2);
TANGOtrans_perform(doBoth);
```

**Listing 2.3:** Spezifikation in TANGO.

### 2.6.4 POLKA-RC

POLKAs [114] Unterschied zu TANGO ist der Animationszähler (frame-counter). Die Transitionen von TANGO werden durch *Actions* ersetzt, die eine ähnliche Semantik haben und zusätzlich eine globale Zeitvariable einführen.

```
time = Animate(time, 20)
```

erzeugt somit 20 neue Animationsframes, die zur Zeit `time` eingeplant werden. Der POLKA Code zum TANGO Beispiel oben ist in Listing 2.4 zu sehen. Zusätzlich bietet POLKA noch einige zusätzliche Eigenschaften gegenüber TANGO, wie Callback-Funktionen, dynamische Typänderung von Objekten und kontinuierliche Actions, die quasi dauerhaft laufen, sowie Anfügestellen für Objekte, an denen andere Objekte angeheftet werden können. Bewegt sich ein Objekt, so wird das angeheftete Objekt mitbewegt.

Ein Nachteil von POLKA ist, dass die Animation nicht zeitgesteuert ist. Dies hat zur Folge, dass die Animation auf schnellen Rechnern evtl. zu schnell abläuft und auf langsamen Maschinen ruckelt. POLKA-RC (real clock) [113] umgeht dieses Problem, indem Animationen auf Basis einer Echtzeit-Uhr berechnet werden. Die wesentlichen Konzepte von POLKA wurden dabei übernommen. Neu hinzugekommen sind die so genannten "Trajectories", die Flugbahnen von Objekten darstellen. Sie bekommen drei Parameter: 1. den Offset vom Start bis zum Ende der Flugbahn, 2. den Bewegungstyp, z. B. direkt oder im Uhrzeigersinn und 3. eine Funktion, die die Geschwindigkeit beschreibt. Actions haben nun zusätzliche Zeitparameter mit



```

Rectangle *rect;
Circle *cir;
Loc *center; *toLoc;
Action *action1, *action2;
double delta[5] = {0.1, 0.1, 0.1, 0.1, 0.1};
int len1, len2;

rect = new Rectangle(view, 1, 0.2, 0.2, 0.1, 0.3,
                    "red", 0.5);
circ = new Circle(view, 1, 0.6, 0.8, 0.05, "blue", 0.0);

center = rect->Where(PART_C);
toLoc = new Loc(0.2, 0.7);
action1 = new Action("MOVE", center, toLoc, STRAIGHT);
action2 = new Action("RESIZE", 5, delta, delta);
len1 = rect->Program(action1, time);
len2 = circ->Program(action2, time);
time = Animate(time, MAX(len1, len2));

```

Listing 2.4: Spezifikation in POLKA.

Werten in Sekunden, Millisekunden oder Spezialwerten wie "NOW", "ASAP" oder "START\_AFTER\_START\_OF".

Abbildung 2.10 zeigt einen Screenshot des Polka Systems. Mittlerweile ist von POLKA-RC auch eine 3-D-Variante verfügbar.

## 2.6.5 Lens

Das Lens System [70] verfolgt einen anderen Ansatz als die bisher genannten Algorithmenanimationssysteme. Lens soll die Lücke zwischen Programmvisualisierung und Algorithmenanimation füllen. Es soll beliebige Programme mit Sichten der Algorithmenanimation darstellen können. Die Autoren möchten dabei die einfache Darstellung eines visuellen Debuggers, der das Programm automatisch als Blockstruktur (Objekte oder Felder) darstellt, erweitern. Die Darstellung soll dem Abstraktionslevel einer Algorithmenanimation ähneln und einfach hergeleitet werden können, ohne dass es nötig ist, Programmierung auf Grafikebene zu kennen. Stasko et al. haben dazu zunächst Algorithmenanimationssysteme untersucht und eine Menge von graphischen Grundprimitiven identifiziert. Diese sind "Linien, Rechtecke, Kreise, Text" sowie "Objekt Arrays". Alle graphischen Primitive haben Kontrollattribute wie Position, Ausdehnung oder Radius. Des Weiteren wurde eine Menge von Grundoperationen auf graphischen Primitiven identifiziert. Diese sind:

- ein Objekt zu einer bestimmten absoluten/relativen Position oder zu einer Position relativ zu einem anderen Objekt bewegen.
- Änderung der (Füll-) Farbe eines Objekts.
- Aufblinken eines Objekts.
- Positionswechsel zweier Objekte.

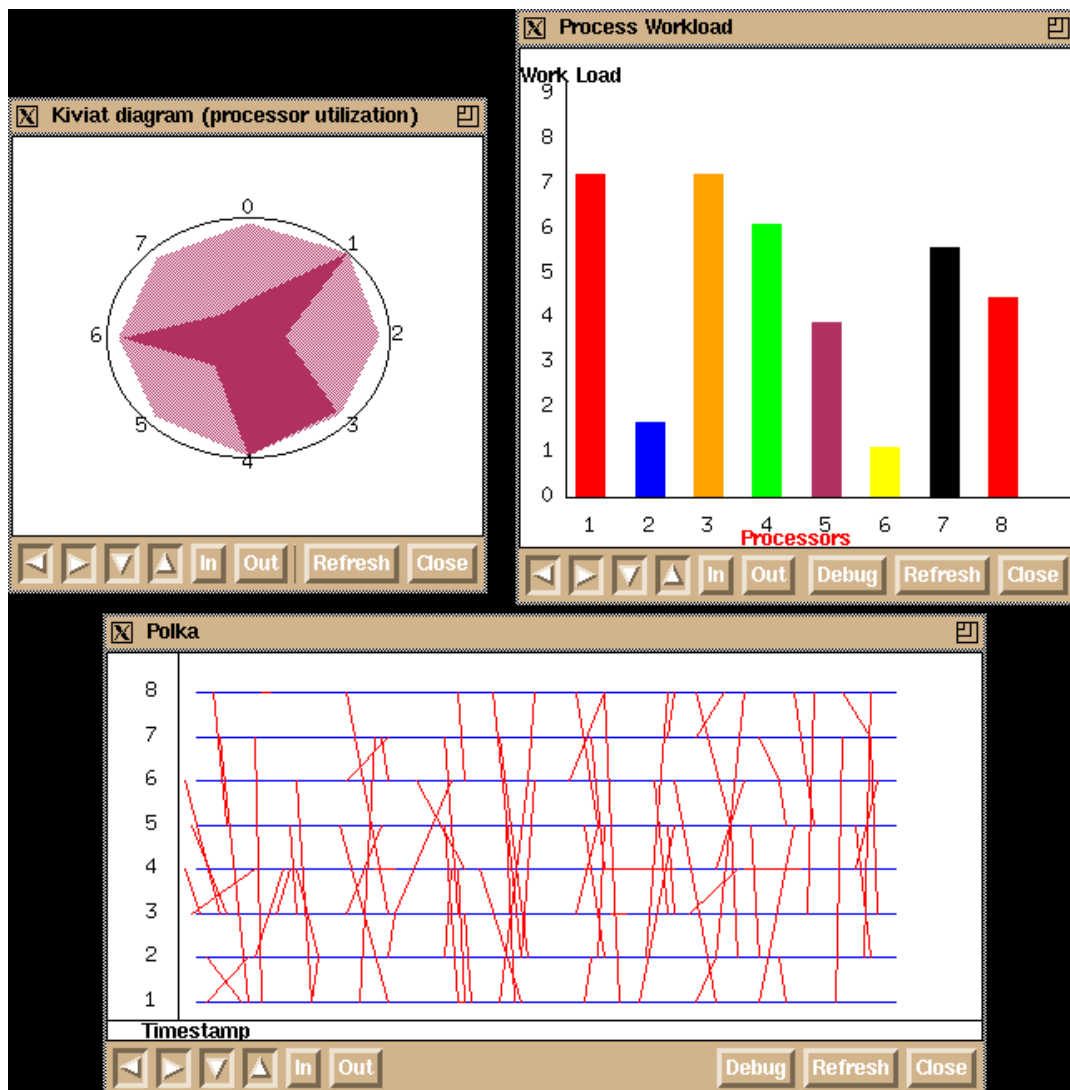


Abbildung 2.10: Polka

- Löschen eines Objekts.

Um im Lens System eine Animation zu entwickeln, wird der Programmcode dazu in den Lens Editor geladen. Dort kann in einer Zeichenfläche ein graphisches Objekt erstellt werden und das Erstellen des Objekts einer bestimmten Zeile im Code zugeordnet werden. Die Kontrollattribute des graphischen Objekts können mit Variablen des Programms assoziiert werden. Lens speichert dies in einer internen Datenbank.

Das Lens System verallgemeinert das Konzept der Algorithmenanimation. Es wird von zu Grunde liegenden Grafikkbibliotheken abstrahiert und es kann eine sehr allgemeine Abbildung von Programmelementen auf graphische Primitive geschaffen werden. Der Code wird nicht, wie beim Ansatz der "interesting events", direkt annotiert, sondern instrumentiert. Bis zu einem gewissen Level ist der Ansatz auch deklarativ,

da ausgesagt wird, *was* animiert wird und nicht *wie*. Allerdings kann es in manchen Animationen auch notwendig sein, an dieser Stelle einzugreifen und z. B. durch eine Boole'sche Formel oder eine textuelle Zusatzspezifikation anzugeben, wann eine Animation stattfindet. Problematisch sind auch graphische Abbildungen von primitiven Arrays. Hier ist eine Abbildung durch direkte Manipulation nur sehr eingeschränkt möglich.

### 2.6.6 Forms/3

Forms/3 [16] basiert auf dem Tabellenkalkulationsparadigma. Der Programmierer manipuliert Zellen in Formularen und definiert Berechnungsvorschriften für jede Zelle. Jede Berechnungsvorschrift darf Konstanten, Referenzen zu anderen Zellen oder Selbstreferenzen beinhalten. Dieser Ansatz ist deklarativ - die Beziehungen zwischen Ein- und Ausgabe werden definiert.

Effekte, wie Änderungen von Daten, wirken sich unmittelbar aus und sind sofort sichtbar. Burnett nennt dies "declarativeness and responsiveness". Der Benutzer muss sich nur um die gewünschte Ausgabe kümmern.

Die Animation folgt Staskos "Path-Transition" Paradigma. Die Animationsarten ähneln denen in Staskos POLKA. So gibt es z. B. Transitionen für Bewegung, Sichtbarkeit, oder Farbe. Transitionen können ebenfalls zusammengesetzt werden. Im Gegensatz zu POLKAs expliziten Aufrufen werden in Forms/3 die Transitionen prinzipiell kontinuierlich ausgewertet und brauchen deshalb nicht explizit ausgewertet werden. Deshalb wird POLKAs "perform" Kommando durch die zwei Kommandos "resetEvent" und "continueEvent" ersetzt. Das Prinzip der "interesting events" wird in Forms/3 deklarativen Ansatz durch Bedingungen im Programm erweitert. Ein Algorithmus muss nun nicht mehr eine bestimmte Stelle im Code erreichen, sondern eine beliebige Bedingung kann eine Algorithmensequenz auslösen. Ein Suchen im Code nach möglichen Stellen, in denen eine bestimmte Bedingung erfüllt sein kann, ist nun nicht mehr nötig. Da Forms/3 sofort auf Änderungen reagiert, kann der Benutzer auch während der Ausführung Animation oder Algorithmus ändern. Ein weiterer interessanter Aspekt von Forms/3 ist, dass mittels einer Zeitschiene die Ausführungsrichtung beliebig geändert werden kann.

### 2.6.7 Pictorial Janus

Pictorial Janus [52] wurde 1989 von Kahn und Saraswat entworfen und basiert auf der textuellen Sprache Janus. Janus wiederum basiert auf ähnlichen Konzepten wie Prolog. Ein Janus Programm ist dabei eine Menge von Klauseln. Jede Klausel hat einen *Head*, einen *Guard* und einen *Body*. Der Guard ist eine Vorbedingung, die erfüllt sein muss, um den Body anzuwenden. Body und Guard sind durch einen "Commit Operator" ("|") getrennt. Janus Programme definieren nebenläufige

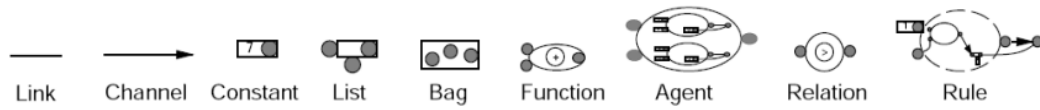


Abbildung 2.11: Pictorial Janus Syntax

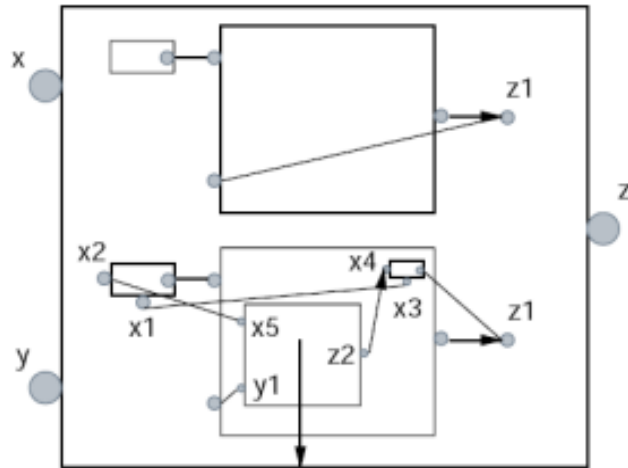


Abbildung 2.12: Pictorial Janus Beispiel: Verknüpfung zweier Listen

Agenten, die durch gerichtete Kanäle kommunizieren. Die Ausführung wird durch Pattern-Matching bestimmt. Falls mehrere Guards gleichzeitig erfüllt sind, so wird ein Guard nicht-deterministisch ausgewählt.

```
append (x, y, z) :- x = [], z = !z1 | z1 = y.
append (x, y, z) :- x = [x1|x2], z = !z1 |
    z1 = [x3|x4], x3 = x1, z.
```

Das Janus Programmlisting definiert das Aneinanderfügen zweier Listen. Pictorial Janus ist die visuelle Janus Variante und basiert auf den Prinzipien

- Bilder sind Programme.
- Bilder sind Schnappschüsse der Berechnung.
- die Berechnung ist die visuelle Manipulation von Bildern.

Pictorial Janus Programme sind topologisch vollständig, d. h. sie sind gegenüber Rotation oder einer beliebigen Ausdehnung wie Streckung, Schrumpfung oder Formveränderung invariant. Graphische Primitive werden durch geschlossene Konturen dargestellt und sind unabhängig von ihrer graphischen Erscheinung.

Abbildung 2.11 zeigt Primitive von Pictorial Janus.

Abbildung 2.12 zeigt eine Pictorial Janus Version des obigen Janus Beispiels. Wird das Programm ausgeführt, so wird je nach Pattern Matching eine Regel innerhalb des Agenten aktiviert und ausgeführt. Diese Regel wird dann bis zum ursprünglichen Agenten graphisch ausgedehnt und so zum neuen Agenten. Innerhalb einer Regel ist die Subkonfiguration des Agenten wiederum rekursiv vorhanden. Prinzipiell entsteht so eine Struktur unendlicher Tiefe, die jedoch nur endlich tief visualisiert wird. Wenn eine Regel aufgebläht wird, so wird die Subkonfiguration ebenfalls langsam sichtbar. Dies entspricht in gewisser Weise einem semantischen Zoom.

Interessant an Pictorial Janus ist, dass eine weiche (“smooth”) Animation erreicht wird, die scheinbar nahtlos zwischen den Systemzuständen interpoliert. Die zugrundeliegende Sprache Janus könnte hinsichtlich der Simulationskomponente Aufschluss über regelbasierte Konzepte geben, die in das zu entwickelnde Spezifikationskonzept einfließen könnten.

### 2.6.8 Alma

Alma [122] ist ein System zur Programmvisualisierung und Algorithmenanimation. Es versucht aus einem abstrakten Strukturbaum den Programmablauf zu visualisieren. Der Strukturbaum ist mit Attributen, die Zeichenfunktionen darstellen, dekoriert. Aus einem Programmtext baut es mit Hilfe des zu Grunde liegenden LISA Compiler Generators einen abstrakten Strukturbaum auf. An diesen können so genannte *Visualisierungsregeln* dekoriert werden. Die Regeln beschreiben ein Baummuster, das zutreffen muss und eine Reihe von Bedingungen, die für den Teilbaum gelten müssen. Sind Baummuster und Bedingungen erfüllt, so werden primitive Zeichenfunktionen aufgerufen. Um Animationen herleiten zu können, werden so genannte *Ersetzungsregeln* benötigt. Diese modifizieren Instanzen von Bäumen, indem zunächst ein Baummuster und konkrete Bedingungen gematched werden. Anschließend wird der Baum anhand der Ersetzungsregel modifiziert. Dies entspricht sehr dem Prinzip der Graphtransformation [38].

Listing 2.5 zeigt die Spezifikation von Visualisierungsregeln in Alma anhand der abstrakten Syntax zur Definition einer Variablenzuweisung (Produktionen p13–p16). Wird ein Baummuster erkannt, das einer Zuweisung entspricht, bei der der Wert des linken sowie des rechten Operanden undefiniert ist, wird eine graphische Darstellung wie in Abbildung 2.13 oben erzeugt. Sind beide Operanden definiert, so wird eine graphische Darstellung wie in Abbildung 2.13 unten erzeugt.

Um Animationen zu erhalten, werden noch Ersetzungsregeln benötigt. Listing 2.6 zeigt zwei Ersetzungsregeln für eine abstrakte Syntax, die einen bedingten Ausdruck modelliert (Produktionen p8 und p9). Die erste Regel trifft zu, falls ein bedingter Ausdruck gematched wird, dessen Bedingung *wahr* ist. Ist dies der Fall, so wird die Regel durch Regel p9 ersetzt, wobei der *THEN*-Teil ausgeführt wird. Andernfalls wird der *ELSE*-Teil ausgeführt.

```

p13: rel_oper : exp exp
p14: exp : CONST
p15: exp : VAR
p16: exp : oper

vis_rule(p13) = <oper,a,c>(
    (a.value=NULL) AND (c.value=NULL)
    AND (a.type=VAR) AND (c.type=CONST)),
    {drawRect(a.name), drawRect(), put(opr.name),
    put('?')}

vis_rule(p13) = <oper,a,c>(
    (a.value!=NULL) AND (c.value!=NULL)
    AND (a.type=VAR) AND (c.type=CONST)),
    {drawRect(a.name, a.value), drawRect(c.value),
    put(opr.name), put('?')}

```

Listing 2.5: Spezifikation von Visualisierungsregeln in Alma.

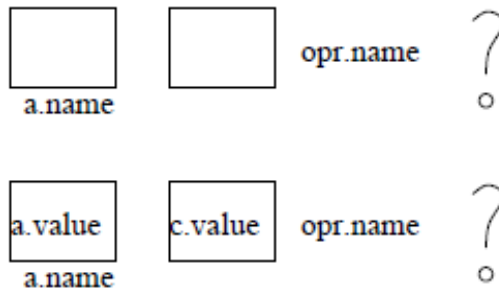


Abbildung 2.13: Visualisierung in Alma

```

p8: IF : cond actions actions
p9:   | cond actions

rule(p8) = <if,op,a,b>,
    (op.value=true),
    <p9:if,op,a>,
    {}

rule(p8) = <if,op,a,b>,
    (op.value=false),
    <p9:if,op,b>,
    {}

```

Listing 2.6: Spezifikation von Ersetzungsregeln in Alma.

Das Alma System erlaubt eine Programmvisualisierung für Sprachen, die über eine abstrakte Syntax verfügen und in LISA implementiert sind. Es ist keine Code-Annotation notwendig. Allerdings müssen zur Visualisierung bestimmte Grafikroutinen bekannt sein. Außerdem kann es schwierig sein aus einer abstrakten Syntax eine Animation herzuleiten, da diese, wie z. B. in Ausdrucksgrammatiken, oft sehr stark von der konkreten Syntax abweicht. Hier ist es wohl notwendig, dass Sprach- und Animationsentwerfer dasselbe Wissen über die Sprache haben.

## 2.7 Generatorsysteme

In den folgenden Abschnitten werde ich verwandte Generatorsysteme für Entwicklungsumgebungen für visuelle Sprachen vorstellen. Ich werde dabei insbesondere auf den Aspekt der Simulation und Animation eingehen. Die Entwicklung der eigentlichen visuellen Sprache steht dabei eher im Hintergrund. In [104] werden einige der Systeme jedoch genauer beschrieben.

### 2.7.1 MetaEdit+

MetaEdit+ [21] ist ein kommerzielles System der Firma MetaCase und gehört zur Klasse der CASE-Werkzeuge<sup>1</sup>. Im Gegensatz zu den weiter unten beschriebenen Systemen basiert es nicht darauf Bäume als Grundlage zur Berechnung der graphischen Repräsentation zu nutzen. Stattdessen wird das so genannte GOPRR-Modell verwendet, was für Graph-Object-Property-Relationship-Role steht. Die einzelnen Begriffe bezeichnen Methoden zur Berechnung einzelner Aspekte einer Sprache. So werden durch das Object-Kalkül Konstrukte modelliert, die mit anderen Konstrukten Beziehungen eingehen können. Das Relationship-Kalkül modelliert Beziehungen zwischen Objekten. Durch das Property-Kalkül können Eigenschaften von Konstrukten modelliert werden.

Der GOPRR Ansatz ist stark auf graphartige Darstellungen spezialisiert, wie sie in CASE-Werkzeugen häufig vorkommen. Für andere und komplexere visuelle Darstellungen ist er nicht gut geeignet. Allerdings wurde beim Entwurf des Systems ein besonderes Augenmerk auf einfache Benutzbarkeit gelegt, deshalb existieren für alle Modellierungsaspekte graphische Werkzeuge. Außerdem unterstützt MetaEdit+ die Teamentwicklung, da alle Sprachkonstrukte in einem globalen Repository abgelegt werden können.

MetaEdit+ erlaubt ebenfalls die Simulation von Programmen. Diese jedoch nur

---

<sup>1</sup>CASE steht für Computer Aided Software Engineering

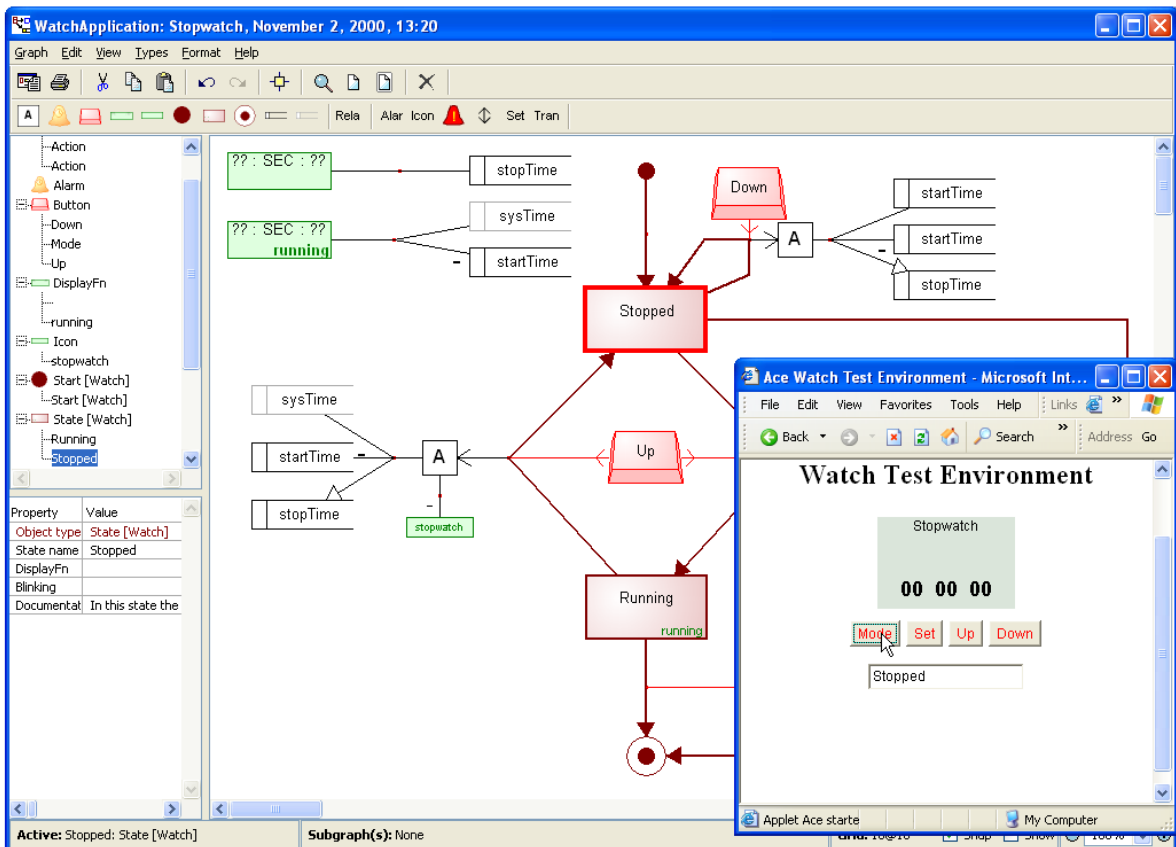


Abbildung 2.14: MetaEdit+

über den Umweg der *Source-to-Source*-Übersetzung. Dazu spezifiziert der Sprachdesigner für seine visuelle Sprache eine Zielcodetransformation. MetaEdit+ kann durch vorgefertigte Module Callback-Funktionen implementieren, die, während das Programm ausgeführt wird, wiederum auf die MetaEdit+ Umgebung zugreifen können. In der MetaEdit+ Umgebung wird dann das entsprechende Sprachkonstrukt, das aktiv ist oder eine Zustandsänderung auslöst, graphisch hervorgehoben. Die Callback-Funktionen sind über eine SOAP Schnittstelle realisiert. MetaEdit+ erreicht so relativ schnell eine einfache Simulation, jedoch mit der Einschränkung, dass erst Code generiert und dieser annotiert werden muss. Die in dieser Arbeit vorgeschlagene Simulationslösung ist eine Ebene direkter. Der Umweg über die Codegenerierung fällt weg.

Abbildung 2.14 zeigt die Ausführung einer Spezifikation, die eine Stoppuhr realisiert. In der rechten Bildhälfte im kleinen Fenster ist das ausgeführte Programm zu sehen. Die Stoppuhr ist im Zustand *Stopped*. Dieser Zustand ist auch in der MetaEdit+ Umgebung selektiert, dargestellt durch einen roten Rand.



## 2.7.2 GenGed/Tiger

Das Spezifikationskonzept von GenGed [6] beruht auf Graphgrammatiken. Der abstrakte Syntaxgraph (ASG) repräsentiert die logische Struktur des Programms. Der konkrete Syntaxgraph ordnet Konstrukten des ASG eine konkrete Repräsentation zu, siehe Abbildung 2.15a. Die graphische Darstellung wird dann durch einen Constraintsolver - im Falle von GenGed wird Parcon eingesetzt - durchgeführt. Dabei wird die graphische Darstellung als eine Menge von Bedingungen aufgefasst, die durch Gleichungen repräsentiert wird. Diese Gleichungen müssen dann vom Constraintsolver gelöst werden. Nachteilig ist hierbei, dass die Gleichungssysteme schnell sehr groß werden können und somit nicht mehr schnell genug vom Constraintsolver gelöst werden können. Dies führt häufig dazu, dass constraintbasierte Generatoren für graphische Struktureditoren nicht so ausdrucksstark sind.

Korrekte Instanzen eines graphischen Programms werden durch visuelle Syntaxregeln ausgedrückt, vgl. Abbildung 2.15b. GenGed generiert somit auch Umgebungen für strukturiertes Editieren.

Die Simulations- und Animationsdefinition in GenGed [32] basiert hauptsächlich auf der Basis von Graphtransformation. Die Basis hierfür bildet AGG [119]. AGG ist eine Entwicklungsumgebung für algebraische Graph-Transformation. Die Transformation basiert dabei auf einem Parsing Prozess des Graphen mittels Backtracking, der exponentielle Zeitkomplexität hat. Um dies zu verbessern wird *Critical Pair Analysis* eingesetzt.

In GenGed können im Layout der Sprache Verhaltensregeln angegeben werden um Simulation zu spezifizieren. Die Regeln bestehen aus einer Vorbedingung, die im Syntaxgraph gematched werden muss und einer Ersetzung des entsprechenden Konstrukts. Um automatisch Regeln anzuwenden, können so genannte *Transformationseinheiten* definiert werden. Diese erlauben die Spezifikation von beliebig vielen Regelanwendungen in einer bestimmten Sequenz. Um eine allgemeine Simulation in ein domänenspezifisches Layout zu transformieren, existieren Sichttransmutationsregeln. Teile des Syntaxgraphen werden durch ein konkretes Layout ersetzt. Abbildung 2.15d zeigt sechs der Regeln, um ein allgemeines Petri-Netz in ein konkretes Layout "Mutter kocht für ihr Kind" zu transformieren. Falls die Transformation erfolgreich ist, wird die entsprechende Animationsfunktion, z. B. "genProd1" aufgerufen.

Der Nachfolger von GenGed namens Tiger basiert auf dem Eclipse-Framework [13] und der Erweiterung dem *Graphical Editing Framework* (GEF)<sup>2</sup>. Die konkrete Repräsentation (in Tiger VL-Alphabet genannt) wird dabei erstellt, indem dem Modellelementen GEF-Graphikprimitiven zugeordnet werden. Die Syntax der

<sup>2</sup>Ein Vergleich des Spezifikationsansatzes von Eclipse mit DEViL ist in [17] zu finden.

Sprache wird durch Graphgrammatikregeln, definiert in AGG, festgelegt. Diese schränken die Standardeditieroperationen von GEF ein. Die Simulation der Sprache wird dabei ebenfalls durch Regeln definiert. Die Animation in Tiger wird durch parametrisierbare Zeichnungen erreicht: in [9] wird dazu das Spiel Rubicks Clock animiert. Die Zeiger der Uhren sind durch ihre Position parametrisiert. Das zeitliche Weiterschalten der Uhren kann so flüssig animiert werden.<sup>3</sup>

GenGed setzt auf die konsequente Anwendung von Graphtransformation und den daraus resultierende formalen Ansatz. Aus Simulationsregeln können so systematisch Animationen abgeleitet werden. Die Spezifikation erfolgt dabei immer visuell und im Layout der entsprechenden Domäne. Problematisch ist die Beschränkung auf Constraint-basiertes Layout, was zu einer Einschränkung hinsichtlich der graphischen Ausdruckskraft führt. Die Simulationsspezifikationen bilden nur konkrete Muster im Syntaxgraph ab - eine Modellierung von Zufall oder eine Gruppierung von Simulationsobjekten ist nicht möglich. Ebenso wenig existieren Mechanismen für eine weiche Animation oder eine Animation von Objekten, die nur indirekt von Modifikationen betroffen sind.

### 2.7.3 DiaGen/DiaMeta

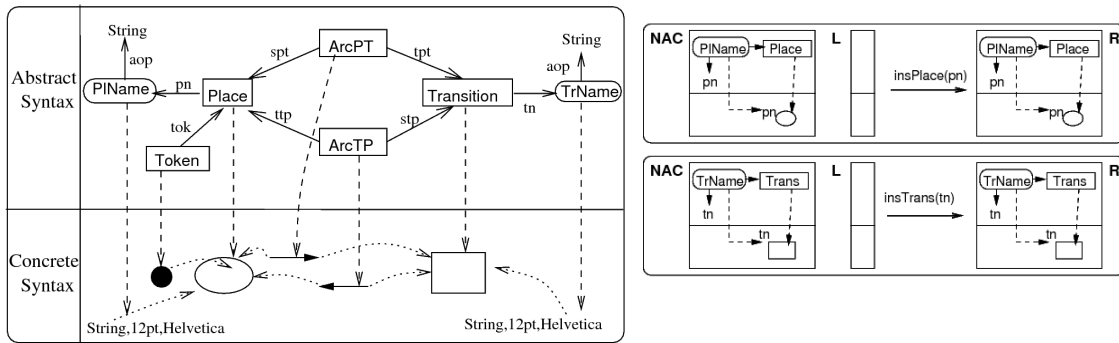
DiaGen [67] ist ein Generatorsystem, das basierend auf einer internen Hypergraphdarstellung wahlweise freie, syntax-gesteuerte oder hybride Editoren generiert. Freie Editoren sind bei der Konstruktion von visuellen Programmen nicht so restriktiv wie syntax-gesteuerte Editoren. Sie erlauben auch zeitweilige Syntaxfehler in der Darstellung und schränken den Benutzer beim Entwurf eines Programms nicht ein. Dies ist besonders vorteilhaft bei großen Programminstanzen. Andererseits helfen syntax-gerichtete Editoren dem Anwender indem sie die Auswahl an Sprachkonstrukten bei jedem Editiervorgang vorgeben.

DiaGen benutzt eine Hypergraphgrammatik um die Syntax der Sprache zu spezifizieren. Hypergraphgrammatiken werden ähnlich wie Grammatiken für textuelle Sprachen definiert. Sie bestehen aus sogenannten terminalen oder nichtterminalen Hyperkanten. Eine Hyperkante hat einen Typ und kann beliebig viele Knoten verbinden. Zusätzlich zur Hypergraphgrammatik werden Constraints angegeben, die das Layout für die Knoten angeben.

Da Hypergraph Repräsentationen bereits für kleine Programme sehr groß werden, wird dieser Graph zunächst durch einen so genannten *Reducer* in ein reduziertes Hypergraph Modell transformiert. Dies ist vergleichbar mit einer abstrakten Syntax, die aus einer konkreten Syntax abgeleitet wird. Im Anschluss wird durch den *Parser* die maximale korrekte Teilmenge des Programms bestimmt. Unkorrekte Teile

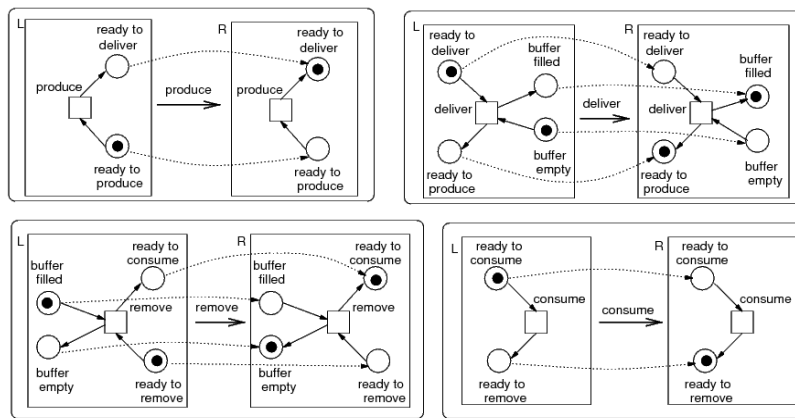
---

<sup>3</sup>Die Animationskomponente in DEViL wird dies automatisch durch das Morphen von Sprachkonstrukten leisten.

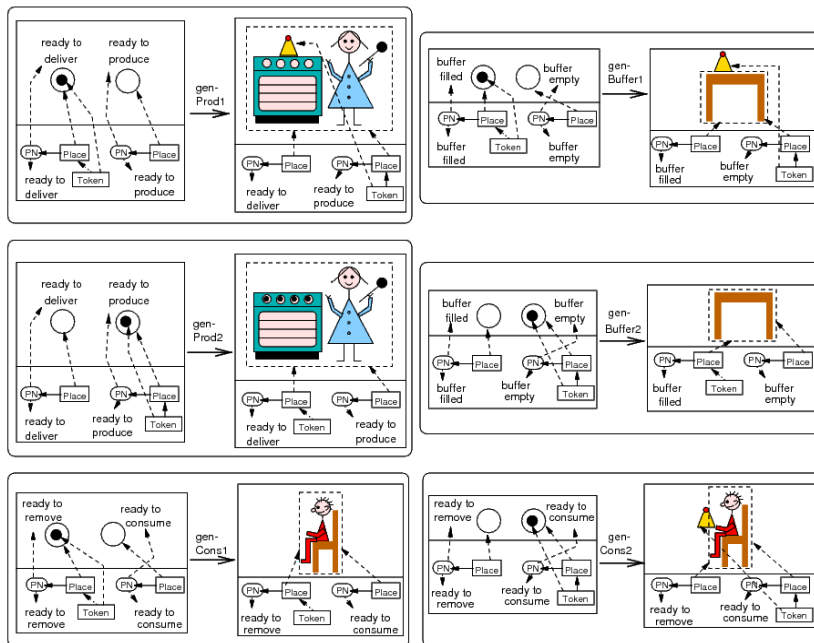


a) Definition des Alphabets

b) Definition der Syntax



c) Verhaltensgrammatik



d) Sichttransaktionsregeln

Abbildung 2.15: Spezifikation in GenGed

werden entsprechend visuell markiert.

Durch einen Attributauswerter werden die korrekten Teile des Diagramms in eine semantische Repräsentation des Programms überführt. Durch einen optionalen *Layouter* wird das Diagramm mittels Constraints in eine optimale Darstellung hinsichtlich Platzbedarf und Optik überführt. Um auch strukturiertes Editieren zu erlauben, können Hypergraph-Transformationsregeln hinzugefügt werden, die den Hypergraph entsprechend modifizieren und auch auf das reduzierte Hypergraphmodell sowie die Ableitungsstruktur zugreifen können.

Die neuere Variante von DiaGen - DiaMeta [68] - basiert auf dem Eclipse Framework. DiaMeta setzt zur Spezifikation des Sprachmodells nicht mehr, wie DiaGen, auf Grammatiken, sondern auf Meta-Modelle.

Es ist damit das erste System, das freie Editoren auf Basis von Meta-Modellen erzeugt. Die Spezifikation von Simulation wird in DiaMeta durch Graphersetzungsregeln auf Basis der abstrakten Syntax erreicht [116, 117]. Dies geschieht (noch) in einer textuellen Sprache. Das Graphmodell wird dabei durch Animationskanten,<sup>4</sup> sowie ein Attribut, das die Animationszeit darstellt, erweitert. So repräsentiert der Graph die sich aktuell verändernde Modellinstanz. Ausgewählt werden die Regeln durch einen Zustandsübergangsautomaten.

Abbildung 2.16 zeigt die Spezifikation von Simulation und Animation auf Basis des Hypergraphen an einem Ausschnitt, der das Spiel Avalanche spezifiziert. Die Darstellung ist hier graphisch, die Spezifikation erfolgt jedoch eigentlich in textueller Form. Die Kanten `switching_to` bzw. `blocked` definieren die diskreten Zustandsübergänge. Das Attribut `tc` repräsentiert die Animationszeit. Die Position von Sprachkonstrukten während der Animation kann durch Formeln angegeben werden (*Time*-Abschnitt im Bild unten). Die Animationskanten geben an, zu welchem Zeitpunkt die Animation gestartet wurde. So kann aufgrund der aktuellen Systemzeit und der aktuellen Graphinstanz eine konkrete Animation auf das Graphmodell zurückgeführt werden.

Animationszustände werden also als Zustände des Graphtransformationssystems angesehen, d. h. jeder Animationszustand lässt sich auf das zu Grunde liegende Graphmodell abbilden. Dies hat den Vorteil, dass auch während der Animation Ereignisse durch den Benutzer ausgelöst werden können, allerdings nur gültige im Sinne der Simulationsspezifikation. DiaMeta berechnet dazu alle gültigen internen Ereignisse, die Zustandsübergänge auslösen können, als auch alle momentan gültigen externen Ereignisse, die der Benutzer während der Animation auslösen kann. Dieser Mechanismus erinnert an das DEVS Verfahren von Vangheluwe und

---

<sup>4</sup>Die Kanten heißen in der Tat Animationskanten, obwohl sie in der Sprache zur Beschreibung von Zustandsübergängen auftauchen.

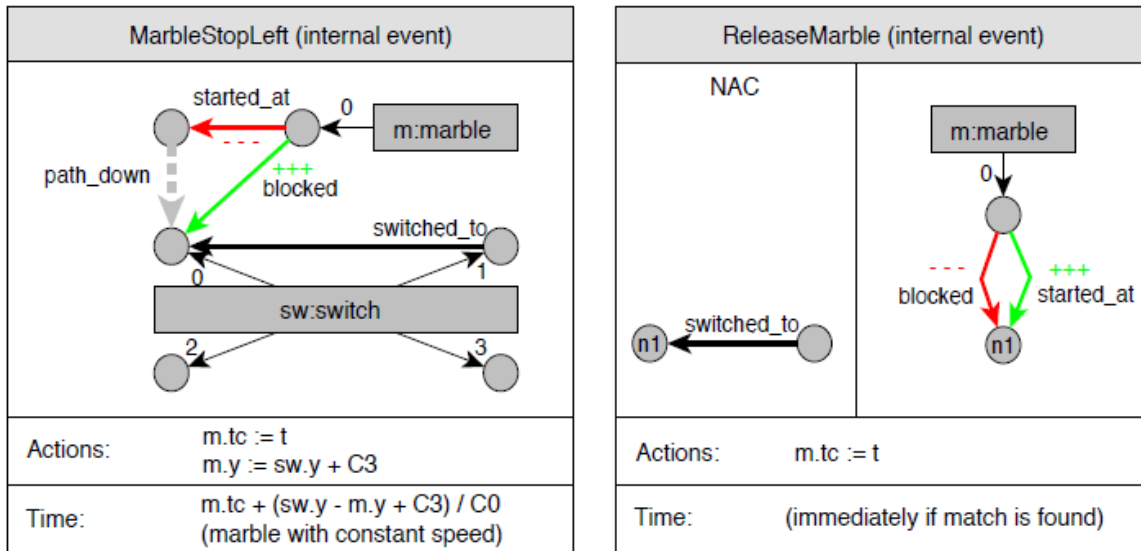


Abbildung 2.16: Spezifikation von Simulation und Animation in DiaMeta (aus [116])

Syriani [118], das auch im ATOM<sup>3</sup>-System (vgl. Abschnitt 2.7.5) zum Einsatz kommt. Des Weiteren lassen sich so recht einfach kontinuierliche Simulationen entwerfen.

Sowohl Simulation und Animation werden in DiaMeta auf Basis des Graphmodells spezifiziert, das durch dedizierte Attribute bzw. Kanten erweitert werden muss. Komplexere Animationen bzw. Animationen auf Objekten, die durch Seiteneffekte betroffen sind, können durch den Layouter ebenfalls berechnet werden.

Mit DiaMeta wurden bereits einige Editoren mit Animationsunterstützung implementiert. Darunter *Alligator Eggs*, eine Variante des Lambda Kalküls [115] sowie die Implementierung des Spiels *Avalanche* [116] und eine Verkehrssimulation.

## 2.7.4 Moses Toolsuite

Die Moses Toolsuite [33] generiert aus Spezifikationen frei editierbare Editoren. Die Definition einer visuellen Sprache ist sehr einfach. Es müssen lediglich Strukturobjekte sowie deren Attribute definiert werden (Abb. 2.17). Durch so genannte Prädikate werden Einschränkungen für gültige Sätze der visuellen Sprache definiert.

Das konkrete Aussehen der Knoten wird durch die Zuordnung von einfachen Formen aus einer Bibliothek erreicht. Benutzerdefinierte Formen können selbst programmiert werden. Ein Pendant zu DEViLs Editor für generische Zeichnungen ist nicht vorhanden. Formen sind jedoch stark parametrisierbar.

```

1 graph type PetriNet {
2   vertex type Place(integer InitialTokens)
3     graphics (Shape = "Oval", AnimationDecorator = "TokenDecorator",
4       ExtentX = 24, ExtentY = 24).
5   vertex type Transition()
6     graphics (Shape = "Rectangle", AnimationDecorator = "StateDecorator",
7       ExtentX = 8, ExtentY = 24).
8   edge type Arc(integer Weight)
9     graphics (Head = "ClosedTriangle").
10
11   predicate "Arc weights must be positive."
12     forall a ∈ Arc : a("Weight") ≠ null ⇒ a("Weight") > 0 end
13   predicate "Arcs from places must end at transitions."
14     forall a ∈ Arc : src(a) ∈ Place ⇒ dst(a) ∈ Transition end
15   predicate "Arcs from transitions must end at places."
16     forall a ∈ Arc : src(a) ∈ Transition ⇒ dst(a) ∈ Place end
17 }
18

```

Abbildung 2.17: Definition einer VL in Moses

Aus den Spezifikationen wird intern dann ein attributierter Graph generiert. Besonderes Augenmerk legen die Autoren von Moses auf die Komposition von Strukturen. So können Formen aus Unterstrukturen bestehen, die beliebig ein- oder ausgeblendet werden können. Dies entspricht einer Verflachung von 2,5-dimensionalen Editoren. Von Moses generierte Editoren sind stark auf graphartige Strukturen festgelegt. Muster wie Matrizen, Tabellen oder Bäume existieren nicht.

Die Moses Toolsuite erlaubt die Spezifikation von Simulationen mittels einer Variante der Abstract State Machines [43] (ASM). ASMs sind eine recht einfache Art operationale Semantik auszudrücken. Transformationen werden durch Regeln ausgedrückt, Sprachkonstrukte können über einfache Bedingungen eingeschränkt werden. Abbildung 2.18 zeigt eine ASM Definition für das Ausführen von Petri-Netzen. In der Definition für die visuelle Sprache kann über das Attribut "AnimationDecorator" eine Klasse zur Animation angegeben werden. Diese Klasse muss jedoch von Hand ausprogrammiert werden. Sprachkonstrukte werden dann über einen Listener über Änderungen benachrichtigt. In der Decorator-Klasse kann dann entsprechend reagiert werden. Moses verfügt mit ASMs über eine mächtige Simulationssprache, die jedoch auch sehr formal ist. Für die Animation hingegen gibt es keine direkte Werkzeugunterstützung, Animationen müssen ausprogrammiert werden. Ebenso schränkt Moses die generierten Sprachen auf Sprachen mit inhärentem Zustand und Zustandsübergängen ein [50].

### 2.7.5 AToM<sup>3</sup>

AToM<sup>3</sup> [27] ist ein visuelles Meta-Modellierungs Werkzeug, das ER-Diagramme mit OCL [125] zur Spezifikation von Meta-Modellen einsetzt. Das Besondere an AToM<sup>3</sup> ist, dass mehrere Sichten auf ein Modell spezifiziert werden können. Dies ist bei

```

1 class PNInterpreter[tm, G] is
2   define
3      $P = \{v \mid v \in \text{vertices}(G), v(\text{"type"}) = \text{"Place"}\},$ 
4      $T = \{v \mid v \in \text{vertices}(G), v(\text{"type"}) = \text{"Transition"}\};$ 
5   input {"fire"} : step;
6   output {"schedule"};
7
8   function M arity 1;
9
10  initialize :
11  once
12    [OutputPorts(this)("schedule")]  $\leftarrow \perp@tm,$ 
13    do forall p  $\in P$  :
14       $M(v) := v(\text{"initialTokens"})$ 
15    end
16  end
17
18  rule step[p, now, v] :
19  once
20    choose t  $\in \{t \mid t \in T,$ 
21       $\forall e \in \text{edges}(G) :$ 
22       $(\text{dst}(e) = t \Rightarrow e(\text{"Weight"}) \leq M(\text{src}(e)))\} :$ 
23
24      do forall e  $\in \text{edges}(G) :$ 
25        if t = dst(e) then
26           $M(\text{src}(e)) := M(\text{src}(e)) - e(\text{"Weight"})$ 
27        end.
28        if t = src(e) then
29           $M(\text{dst}(e)) := M(\text{dst}(e)) + e(\text{"Weight"})$ 
30        end
31      end.
32    [OutputPorts(this)("schedule")]  $\leftarrow \perp@now$ 
33  end
34 end
35 end

```

Abbildung 2.18: Definition eines Petri-Netz Interpreters in Moses

GenGed und DiaGen nicht der Fall. Die Spezifikation weiterer Sichten wird durch Graphersetzungsgesetze auf dem Ausgangsmodell erreicht. Dieser Mechanismus ist vergleichbar mit dem Konzept der gekoppelten Sichten in DEViL.

Neben der Modellierung von Modellen in ER-Notation können Elementen primitive, bereits voreingebaute typisierte Attribute, wie `Integer` oder `String`, mitgegeben werden oder zusammengesetzte Attribute, die wiederum als Sub-Modelle aufzufassen sind.

Simulation ist in AToM<sup>3</sup> ebenfalls durch Graphersetzungsgesetze möglich, die die Autoren als einen "natürlichen, deklarativen und allgemeinen" Ansatz sehen. Dazu wird in AToM<sup>3</sup> das DEVS-Modell [118] verwendet. Ein atomares DEVS-Modell wird dabei zusammengesetzt aus  $\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$ .  $S$  ist eine Menge sequentieller Zustände,  $X$  ist eine Menge erlaubter Eingabeereignisse,  $Y$  eine Menge erlaubter Ausgabeereignisse.  $\delta_{int}$  bzw.  $\delta_{ext}$  ist die interne bzw. externe Transitionsfunktion.  $\tau$  ist die Zeitfortschaltungsfunktion,  $\lambda$  ist die Ausgabefunktion.

Das Modell startet in einem Initialzustand. Es bleibt in einem beliebigen Zustand, solange es die Zeitfortschaltungsfunktion vorschreibt und kein Eingabeereignis stattfindet. Falls kein Eingabeereignis stattfindet, wird ein Ausgabeereignis erzeugt, sobald die Zeitfortschaltungsfunktion für den momentanen Zustand abgelaufen ist. Danach wird in den neuen Zustand gesprungen. Findet ein Eingabeereignis statt, dann wird die externe Transitionsfunktion angewendet.

Mit sogenannten gekoppelten DEVS-Modellen lässt sich ein DEVS-Modell noch weiter hierarchisch strukturieren. Dabei beschreiben Transfer-Funktionen die Ausgabe-zu-Eingabe Beziehungen zwischen Komponenten. Abbildung 2.19a zeigt ein gekoppeltes DEVS-Modell in AToM<sup>3</sup> zur Simulation von Modellen. Der Controller wählt die im Block `GGRule` definierten Regeln aus und wendet sie an. Die Regeln selbst sind ein hierarchischer DEVS-Block, der in Abbildung 2.19b dargestellt ist. Regeln sind priorisiert und in Layern angeordnet. Ein Synchronisationsblock entscheidet über die Anwendung von Regeln, falls mehrere in demselben Prioritätslayer matchen. Die Linien zwischen den Blöcken stellen den Datenfluss dar. Die Daten an sich sind in AToM<sup>3</sup> Graphen.

Ein Vorgehen zur Herleitung einer Animation ist nicht bekannt.

Konsequent wird in AToM<sup>3</sup> der Modellbegriff eingesetzt - die Transformation zwischen Modellen wird als ein Hauptziel angesehen. AToM<sup>3</sup> ist damit im Gebiet der *Multi-Paradigmen Modellierung* [121] anzusiedeln. Der Vorteil, des DEVS-Ansatzes in AToM<sup>3</sup> liegt in der Modularität. DEVS-Blöcke können als Black-boxes angesehen werden, die leicht zu erweitern sind. Kontinuierliche Simulation, genauso wie Benutzerinteraktion sind ebenso leicht modellierbar, indem z.B. der `GGRule`-Block



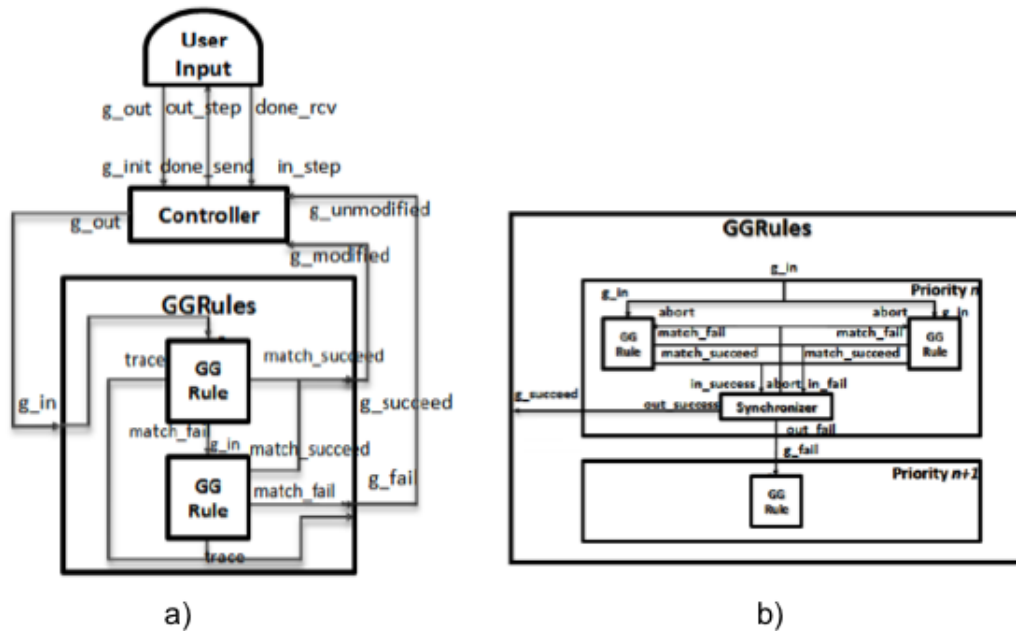


Abbildung 2.19: Gekoppeltes DEVS-Modell in AToM<sup>3</sup>

in Abbildung 2.19b nochmals unterteilt wird um autonome und benutzerdefinierte Regeln zu unterstützen.

## 2.8 Relevanz im Kontext dieser Arbeit

Die bisherigen Betrachtungen beinhalten bereits einige Anforderungen an das zu entwerfene Gesamtkonzept. Die Klassifizierung der Programmvisualisierung von Price et al. sowie die Darstellung von Cox zeigen bereits einige generelle Aspekte für eine Simulation und Animation einer visuellen Sprache. Die Animation sollte eine nahezu beliebige Abbildung von der zu Grunde liegenden Simulation zulassen. Große Visualisierungen sollten geeignet aufbereitet werden, um ein hohes Maß an Verständnis zu erreichen. Hier ist es wichtig Maßnahmen zu treffen um eine Animation hinsichtlich zeitlichem Ablauf, Strukturierung und Art der Visualisierung anzupassen. Auch ist es wichtig das Sprachspektrum nicht einzuschränken. Alle Editortypen, die sich mit DEViL generieren lassen, sollen auch mit einer Simulations- und Animationskomponente ausgestattet werden können, falls dies gewünscht ist.

Die Animation soll nicht nur direkt auf der Simulation aufbauen, sondern dazu benutzt werden um das Verständnis der Simulation zu fördern. Im weiteren Verlauf der Arbeit wird sich zeigen, dass dazu auch unkonventionelle Mittel wie z. B. *“Exaggeration”* und *“Anticipation”* aus der Comicanimation eingesetzt werden können. Der Einsatz von mehreren gleichzeitig arbeitenden animierten Sichten

erhöht nicht nur die Skalierbarkeit der Sprache. Es können auch verschiedene Varianten der simulierten Sprachinstanz gleichzeitig betrachtet werden. Hier erlaubt das Sichten- und das Strukturkopplungskonzept von DEViL bereits ausgefeilte Darstellungs- und Strukturierungsvarianten.

Die betrachteten Algorithmenanimationssysteme sind besonders interessant im Hinblick auf die Animationskomponente. Sie stellen Basisanimationstypen bereit, wie beispielsweise das Bewegen von graphischen Objekten. TANGOs Path-Transition Paradigma erlaubt es einfach Animationspfade zu spezifizieren. Auch dies wird sich in dieser Arbeit wiederfinden, wenn es darum geht, Objekte im Strukturmodell entlang beliebiger Positionen zu bewegen. Pictorial Janus' kontinuierliche Animation zeigt, dass es für eine gute Animation wichtig ist, Zustandsübergänge als eine weiche Animation darzustellen.

Häufig ist die Darstellung der simulierten visuellen Sprache von der ihr zugrundeliegenden DSL weit entfernt. Zum Beispiel werden Roboter im Fahrzeugbau mit einer Spezialsprache programmiert. Zum Testen dieses Programms möchte man sich jedoch der visuellen Domäne der Roboter bedienen, anstatt eine Art Debugging im Robotercode durchzuführen. Dieser strukturelle Unterschied zwischen Simulations- und Animationsstruktur sollte ebenfalls einfach spezifizierbar sein.

Auf Seiten der Simulation zeigen die Simulationssprachen und -umgebungen zum einen den Umgang mit der Simulationszeit und zum anderen den Umgang mit Simulationsobjekten. Die Simulationszeit sollte sich möglichst feingranular einstellen lassen und ein "Springen" in der Simulationszeit sollte ebenso ermöglicht werden. Nicht nur Komprimierung von Zeit, sondern auch das beliebige Zurückspringen und das Einplanen von Aktionen in der Zukunft sind wichtig. Hierbei ist eine ereignisbasierte Simulation mit einer prioritätenbasierten Warteschlange am flexibelsten. Es wird sich in den nächsten Kapiteln zeigen, dass nur *eine* Ereignis-Warteschlange nicht immer ausreichend ist. Insbesondere wenn Daten des Simulationsmodells gleichzeitig geschrieben und gelesen werden. Dies führt zu "Race-Conditions", die ebenfalls geeignet behandelt werden müssen. Lang laufende Simulationen sollten interessante Zwischenergebnisse speichern bzw. an vom Benutzer festgelegten Haltepunkten, die vielleicht eine konkrete Ausprägung der Simulation darstellen, stoppen.

Alle Simulationssprachen und -umgebungen haben gemein, dass Simulationsobjekte erstellt werden und durch das Simulationsmodell wandern, wobei sie manipuliert werden. Die Simulationsobjekte sind sich ihrer Umgebung "bewusst", d. h. je nachdem in welchem Kontext sie sich befinden, werden entsprechende Aktionen ausgeführt, z. B. die Verzögerung beim Verladen in einer Transportsimulation. Diese Erkenntnis führt zu kontext-sensitiven Simulationsfunktionen (vgl. Abschnitt 4.3.1 auf Seite 100). Des Weiteren sollte eine Simulationssprache das einfache Erstellen bzw. Gruppieren von Simulationsobjekten erlauben. Der Zugriff

auf Simulationsobjekte einer bestimmten Art oder mit bestimmten Eigenschaften ist insbesondere bei visuellen Sprachen wichtig (z. B. "alle Stellen eines Petri Netzes, die Vorbedingung einer bestimmten Transition sind"). Die Simulationsobjekte kapseln dabei Eigenschaften, sind also im Sinne der objektorientierten Programmierung Instanzen von Klassen. Diese Art der Gruppierung und Neuordnung von Objekten führt zum Begriff des Simulationsmodells.

Um aussagekräftige Simulationen zu erreichen, ist ein Simulationsmodell, das Zufallsvariablen diverser Verteilung bietet von Bedeutung. Alle Simulationssprachen bieten hierfür spezialisierte Konstrukte. Auch ist das Protokollieren von Variablen über die Zeit (so genannte Tracevariablen) wichtig, um Aussagen zum Simulationsmodell zu treffen. Hier zeigt sich, dass die Simulationsumgebungen ausgefeilte Mechanismen zur Analyse mitbringen. Dies können Report Mechanismen oder die Überprüfung von Eigenschaften des Simulationsmodells sein. In der vorliegenden Arbeit wird dieser Aspekt besonders berücksichtigt. Der hier entworfene Simulator wird Mechanismen zur Analyse bereitstellen, die über die reine Animation hinausgehen. Es werden dem Sprachanwender und auch dem Sprachspezifizierer weitere Werkzeuge zur Überprüfung der Sprachinstanz mitgegeben.

Viele Simulationsumgebungen sind auf die Simulation von Fertigungsstraßen spezialisiert. Hier gibt es diverse Sprachkonstrukte um den Fluss von Werkstücken zu spezifizieren. Die Simulationsobjekte sind sich ihrer Umgebung "bewusst", d. h. je nachdem in welchem Kontext sie sich befinden, werden entsprechende Aktionen ausgeführt, z. B. die Verzögerung beim Verladen in einer Transportsimulation. Diese Erkenntnis führt zu kontext-sensitiven Simulationsfunktionen (vgl. Abschnitt 4.3.1 auf Seite 100).

Des Weiteren ist, wenn die Spezifikation von Simulation betrachtet wird, ein Nachteil in den Algorithmenanimationssystemen erkennbar: Beim Konzept der "interesting Events" muss der Animator den Algorithmus kennen um animieren zu können. Der Simulationsalgorithmus wird durch Animationsaufrufe erweitert. Simulation und Animation sind somit nicht mehr voneinander entkoppelt, was dazu führen kann, dass interessante Ereignisse verpasst werden bzw. eigentlich gar nicht wirklich von Belang sind. Letztlich muss der Animator genauso viel über den Simulationsalgorithmus kennen, wie der Simulationsentwerfer.

Wenn die generierten Simulationsumgebungen betrachtet werden, fällt auf, dass es sehr nützlich sein kann, wenn der Benutzer direkt in die Animation eingreifen und diese unmittelbar verändern kann. Dies hat dann direkte Auswirkungen auf die Simulation. Der Forms/3 Ansatz zeigt das deutlich.

Die vorgestellten Generatorsysteme zur Spezifikation von Struktureditoren haben zum Teil schon Mechanismen eingebaut, um Simulationen und Animationen herzustellen. Zum einen sind Systeme auf Basis von Graphtransformation zu

nennen, die Simulation über Ersetzungsregeln definieren. Der Ansatz ist sehr formal jedoch deswegen auch problematisch. Der Spezifizierer muss sich zunächst mit einem formalen Kalkül auseinandersetzen. Insbesondere beim GenGed System ist dies schwierig, da recht viele neue Ebenen für die Simulation und Animation eingeführt werden. Von Vorteil ist, dass Graphersetzungssysteme wie GenGed Simulation auf der konkreten Syntax definieren können. Einschränkend ist zu sagen, dass Graphersetzungregeln Zusammenhänge zwischen entfernten Objekten sowie Objekten, die *irgendwo* im Modell zu finden sind nur schlecht beschreiben können [58]. Zudem ist es schwierig, immer alle möglichen Fälle abzudecken. Dies führt häufig zu einer Explosion der Regelmenge und auch zu Problemen, falls mehrere Regeln zutreffen. Meist wird dann eine Regel nicht-deterministisch ausgewählt.

Eine weitere Schwierigkeit ist das Speichern von Zuständen, damit Regelsequenzen ausgeführt werden können. Hier müssen Regeln künstlich um Attribute, die den Zustand speichern, erweitert werden.

## 2.9 Das DEViL-System

Das Generatorsystem DEViL (Development Environment for Visual Languages) ist eine Entwicklungsumgebung für visuelle Struktureditoren. Es basiert auf dem Vorgängersystem VL-Eli von Jung [54] und wurde von Schmidt weiterentwickelt [101]. Aus Spezifikationen eines hohen Abstraktionsniveaus erzeugt DEViL vollständige graphische Editoren mit Codegenerierung. Die Spezifikationen beschreiben dabei das Modell der Sprache, sowie Repräsentation und Transformation der visuellen Sprache. Von DEViL generierte Editoren basieren auf dem Model-View-Controller Paradigma, wobei die so genannte abstrakte Struktur das Modell der Sprache ist und verschiedene graphische Repräsentationen, so genannte Sichten, daran gekoppelt sind. Das Spezifikationskonzept ist in Abbildung 2.20 dargestellt. Es besteht im Wesentlichen aus der Spezifikation der abstrakten Struktur mittels DSSL (DEViL Structure Specification Language) und der Dekoration dieser abstrakten Struktur mit visuellen Mustern (VP).

Die visuellen Muster beschreiben deklarativ, wie Teilbäume der Struktur graphisch repräsentiert werden sollen. Dazu steht eine breite Bibliothek an vorgefertigten Mustern zur Verfügung. So kann beispielsweise ausgedrückt werden, dass Teilbäume als Listen, Mengen oder Matrizen dargestellt werden sollen. Zusätzlich kann ein Codegenerator spezifiziert werden, um aus Instanzen der visuellen Sprache Zielcode zu generieren. Dazu stehen alle Werkzeuge des Eli-Systems [41], auf dem DEViL basiert, zur Verfügung. DEViL benutzt Generatoren zur Sprachimplementierung des Eli Systems sowie Wodan, ein eigenes Herstellungsprozesssystem. Mit Dot zum Layout von Graphen und AspectC++ zur Implementierung von aspekt-orientiertem Code kann der Generator Struktureditoren generieren, die eine eigene Multi-

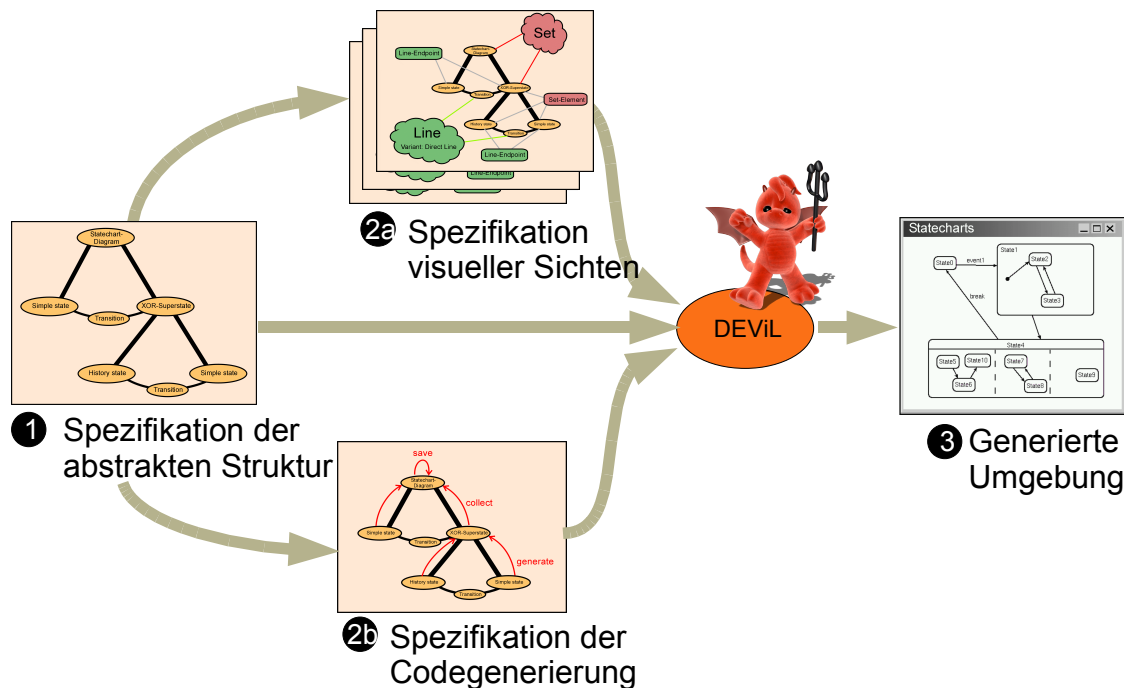


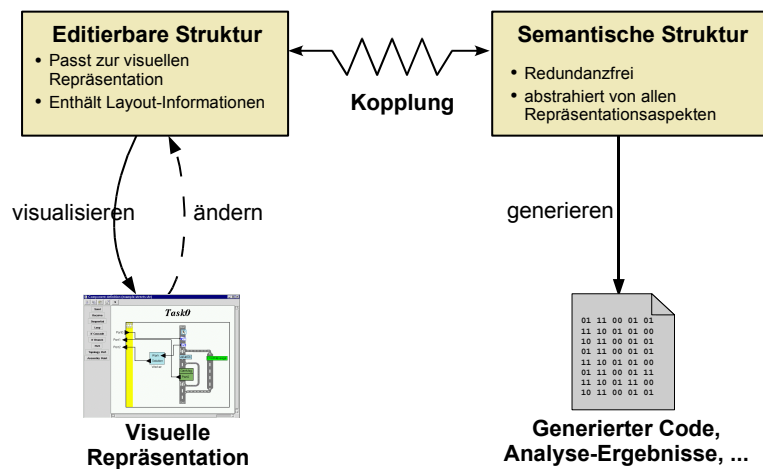
Abbildung 2.20: Das Spezifikationskonzept von DEViL

Fenster-Umgebung mit selbstdefinierten Datentypen unterstützen. Zur visuellen Spezifikation graphischer Struktureditoren kann der DEViL-Designer [23] benutzt werden, der Eingaben für alle unterliegenden Werkzeuge generiert.

Alle von DEViL generierten Editoren basieren auf einem Grundmodell: der Trennung von editierbarer und semantischer Struktur (Abb. 2.21). Die abstrakte Struktur einer visuellen Sprache ist mit der abstrakten Syntax einer kontextfreien Sprache in Übersetzern vergleichbar. Sie bildet das Fundament für semantische und editierbare Struktur. Die semantische Struktur enthält die für das Programm semantisch wichtigen Konstrukte und ist frei von Attributen, die nur der konkreten Repräsentation dienen. Die editierbare Struktur ist Grundlage für die konkrete graphische Repräsentation der Darstellung. Sie enthält vor allem zusätzliche Layout-Informationen.

### 2.9.1 Spezifikation der abstrakten Struktur

Zur Spezifikation der abstrakten Struktur wird in DEViL *DSSL* benutzt. *DSSL* basiert auf objektorientierten Modellierungskonzepten wie Klassen, Attributen und Vererbung. Jede Instanz einer nicht-abstrakten Klasse repräsentiert ein graphisches Objekt in der Darstellung. Innerhalb von Klassen existieren drei Knotentypen:



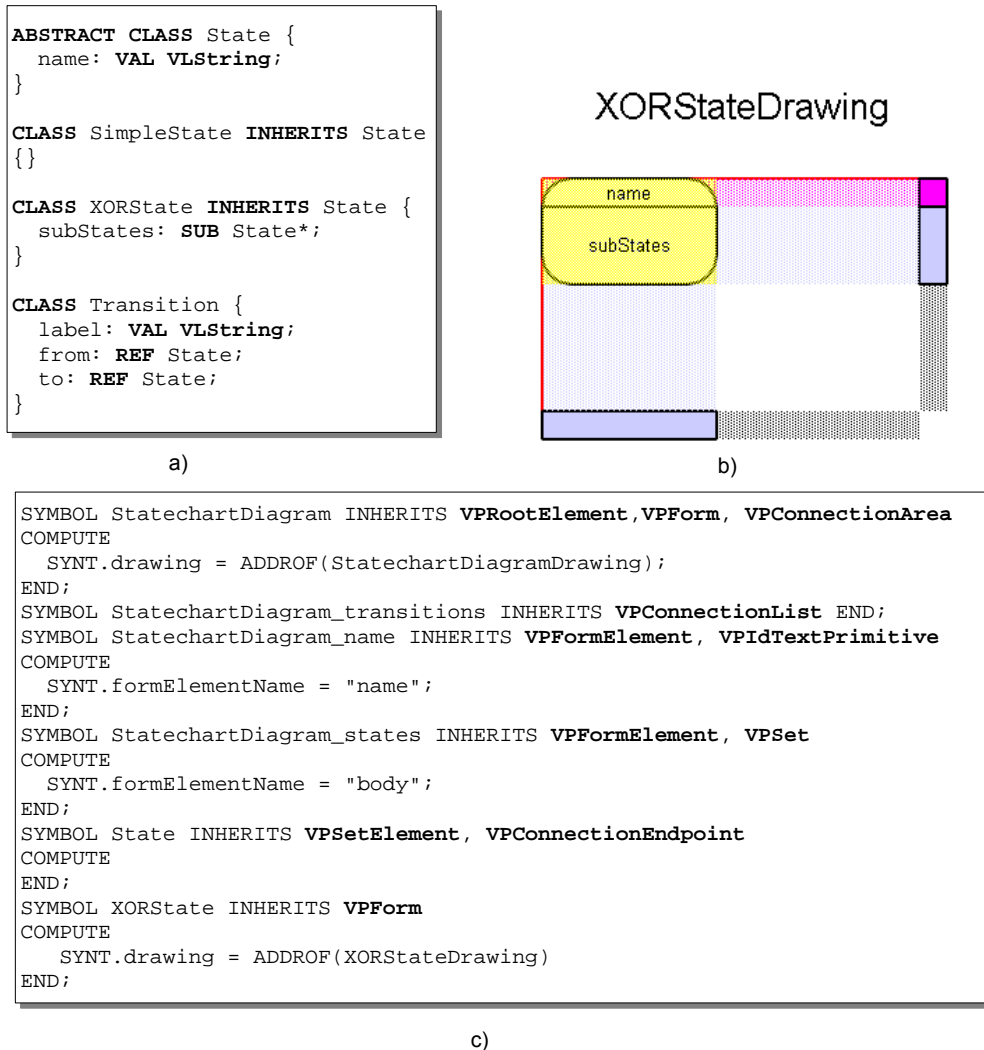
**Abbildung 2.21:** Die Trennung von editierbarer und semantischer Struktur in DEViL (aus [101])

- Knoten vom Typ VAL. Sie speichern primitive Attribute und können diverse Datentypen annehmen, z. B. VLInt zum Speichern von Integer-Werten oder VLString zum Speichern von Zeichenketten. Durch ein Fragezeichen hinter dem Knoten kann spezifiziert werden, ob ein VAL-Attribut zwischenzeitlich auch leer sein darf.
- Knoten vom Typ REF. Sie speichern Referenzen zu anderen Objekten im Strukturbaum. Sie modellieren also die Querbeziehungen im Baum.
- Knoten vom Typ SUB. Sie speichern Unterstrukturen vom angegebenen Typ. Sie modellieren also die Baumkanten der abstrakten Struktur. SUB-Knoten dürfen nur Objekte von Untertypen der angegebenen Klasse enthalten. Hinter der Typangabe darf eine Kardinalität angegeben werden. So können auch Knoten modelliert werden, die nur genau einen Unterbaum speichern.

In der Arbeit werden Knoten vom Typ VAL bzw. REF auch *Attributknoten* genannt. Knoten vom Typ SUB werden *Aggregationsknoten* genannt. Wenn alle Knoten angesprochen werden sollen, wird in dieser Arbeit die Bezeichnung *Sprachkonstruktknoten* verwendet.

Abbildung 2.22a zeigt einen kleinen Ausschnitt aus einer Spezifikation für Statecharts. Dabei erben die Klassen SimpleState und XORState von der abstrakten Klasse State. In XORState können im Attribut subStates weitere Unterobjekte des State Knotens enthalten sein. Die Klasse Transition enthält neben einem Attribut für den Namen, zwei Referenzen auf State Knoten.

**Stärken und Schwächen** DSSL besticht durch die Einfachheit der Beschreibung von strukturellen Eigenschaften. Es kommt mit wenigen orthogonalen Sprachkonstrukten aus, die eine Beschreibung von Struktur und persistenten Daten ausdrücken können. Somit ist die Sprache sehr übersichtlich und schlicht gehalten, aber trotzdem mächtig genug um beliebige Baumstrukturen beschreiben zu können. Aus einer Spezifikation der abstrakten Syntax kann bereits ein baumbasierter und voll funktionsfähiger Struktureditor generiert werden. Allerdings können in DSSL keine Transformationen auf Basis eines Automaten ausgedrückt werden. Es gibt keine direkte Unterstützung zur Gruppierung von Sprachkonstruktinstanzen.



**Abbildung 2.22:** Spezifikation von Struktur und Repräsentation: a) semantische Struktur, b) eine generische Zeichnung, die einem Sprachkonstrukt seine konkrete graphische Darstellung verleiht und c) Beschreibung der graphischen Repräsentation durch Anwendung visueller Muster.



## 2.9.2 Spezifikation der graphischen Darstellung

Die Sprachkonstruktknoten der abstrakten Struktur werden, um eine konkrete graphische Repräsentation zu erreichen, mit visuellen Mustern attribuiert (Abb. 2.22c). Wie zu sehen ist, erben die Sprachkonstruktknoten dabei bestimmte Rollen der visuellen Muster, z. B. `VPSet` zur Darstellung einer Menge. DEViL stellt eine große Bibliothek an vorgefertigten Mustern bereit. Dies sind beispielsweise Muster zur Darstellung von Listen, Mengen, Matrizen, Tabellen und vieles mehr. Die visuellen Muster können durch Kontrollattribute geeignet parametrisiert werden um das Layout und die graphische Darstellung individuell anzupassen. In [103] und [104] werden die visuellen Muster genauer beschrieben.

Das visuelle Muster `VPForm` hat eine herausragende Rolle: Es hat die Eigenschaft, Sprachkonstrukten ein spezifisches Aussehen zu verleihen. So kann diesem Muster durch ein Kontrollattribut `SYNT.drawing` eine so genannte generische Zeichnung zugeordnet werden. Abbildung 2.22b zeigt eine solche Zeichnung für das visuelle Element `XORState` der Statecharts. Generische Zeichnungen ordnen Sprachkonstrukten ihr konkretes Aussehen zu. Sie sind als abstraktes Konzept zu verstehen und kapseln neben graphischen Primitiven, wie Linien, Rechtecken oder Kreisen auch Ausdehnungseigenschaften. Innerhalb von generischen Zeichnungen können so genannte *Container* benutzt werden. Container können Unterbäume aufnehmen und dehnen die generische Zeichnung entsprechend der Größe der aggregierten Elemente aus. In Abb. 2.22b sind die beiden Container `subStates` und `name` enthalten. Container können innerhalb der Sichtdefinition mit dem Kontrollattribut `formElementName` angesprochen werden (vgl. Abb. 2.22c). Das Besondere an generischen Zeichnungen ist, dass sie automatisch den Platzbedarf für aggregierte Objekte berechnen und sich die Zeichnung beliebig ausdehnen kann. Ist so beispielsweise der horizontale Platzbedarf nicht ausreichend, so wird die Zeichnung gedehnt, indem quasi an den Enden "gezogen" wird. Diese Ausdehnung kann mit Eigenschaften der Container angepasst werden, d. h. sie ist insbesondere nicht linear!<sup>5</sup>

Die Musteranwendung erfolgt in der funktionalen Sprache LIDO [53]. In der Regel müssen nur wenige Kontrollattribute überschrieben werden. Falls dies nicht ausreicht, stehen jedoch alle Mechanismen des zu Grunde liegenden Eli Systems zur Verfügung. Auch benutzerdefinierter Code in Form von C- oder Tcl-Funktionen kann zur Berechnung benutzt werden.

**Stärken und Schwächen** Der Ansatz zur Spezifikation einer graphischen Darstellung ist vollständig deklarativ. Durch die große visuelle Musterbibliothek und die Kombinierbarkeit der Muster kann nahezu jede graphische Darstellung erreicht werden. Die Modellierung des Zusammenhangs zwischen Struktur und Darstellung

<sup>5</sup>Dies führt später in der Arbeit zum Konzept des *Morphens* von generischen Zeichnungen.

ist klar: Ein Sprachkonstrukt wird in einer Sicht durch eine Musterberechnung dargestellt<sup>6</sup>.

Für eine Animation kann dieser direkte Bezug auch ein Nachteil sein. Objekte, die keinen Repräsentanten im Modell haben, z. B. flüchtige Objekte, können zunächst nicht definiert werden. Die Spezifikation der Repräsentation ist vollständig von der Strukturdefinition entkoppelt, was zu einer Erleichterung bei der Entwicklung in Teamarbeit führt. Das Konzept der Spezifikation der graphischen Darstellung mittels generischer Zeichnungen überzeugt durch den visuellen Ansatz. Nachteilig im Hinblick auf die hier entworfene Animationserweiterung wirkt sich aus, dass zwar Strukturobjekten eindeutig graphische Primitive zugeordnet werden, dies aber nicht konsequent bei Attributen gemacht wird. D. h. Attribute haben in der graphischen Darstellung keine Identität mehr. Dies ist problematisch, wenn auf einem Attribut eine Änderung in der Simulation ausgelöst wird, dieses Attribut aber in der graphischen Repräsentation nicht mehr "gefunden" werden kann. Hier zeigt sich ein generelles Problem der Implementierung von DEViL: Der Weg von der Struktur zur graphischen Repräsentation ist durch Abbildungen recht einfach, diese sind jedoch nicht bijektiv, sodass der Rückweg entsprechend schwierig ist.

Der Algorithmus zur Berechnung der graphischen Darstellung mittels generischer Zeichnungen ist sehr komplex und wirkt sich zum Teil unvorhersehbar auf die graphische Darstellung aus. Dies führt später in der Arbeit dazu, dass die Repräsentation von zwei aufeinanderfolgenden Simulationszuständen nur durch einen "Trockenlauf" der Attributauswerter und dem Zwischenspeichern aller(!) graphischer Objekte erreicht werden kann.

### **Grammatikabbildung**

Um attributierte Grammatiken als Kalkül zur Berechnung der Darstellung verwenden zu können, muss die semantische Struktur, spezifiziert durch DSSL, auf eine kontextfreie Grammatik abgebildet werden. Dieses Verfahren beruht darauf, dass die Abbildung eine neue Struktur schafft, die für LIGA [53] geeignet ist. Allgemein wird dabei jede nicht-abstrakte DSSL Klasse auf eine Grammatikregel abgebildet. Auf der linken Seite steht ein Nichtterminal, das sich aus dem Namen der Klasse ableitet. Auf der rechten Seite steht die Attributhülle der Klasse. Diese besteht aus der Menge der direkt definierten Attribute, vereinigt mit den direkt oder indirekt geerbten Attributen.

Die Abbildung der DSSL Sprachkonstrukte auf die kontextfreie Grammatik geschieht automatisch, der Anwender braucht hier in der Regel nicht einzugreifen.

---

<sup>6</sup>Dies ist auch ein Nachteil, da innerhalb einer Sicht ein Sprachkonstrukt an allen Stellen dieselbe Musteranwendung bekommt. Dies resultiert aus der Spezifikationssprache LIDO, die keine mehrfachen Aufkommen von gleichbenannten Symbolberechnungen erlaubt.

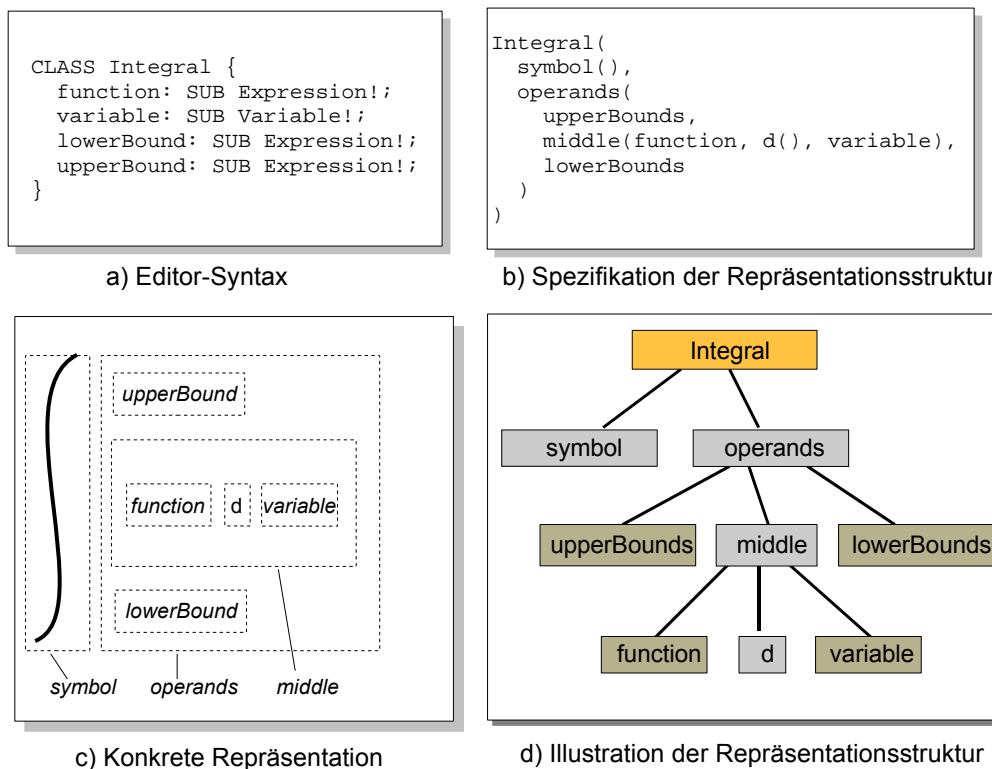


Abbildung 2.23: Das Konzept der Grammatikabbildung (aus [101])

Allerdings gibt es Fälle, in denen es sinnvoll ist, die automatische Grammatikabbildung zu überschreiben. Meistens kommt dies vor, wenn editierbare und semantische Struktur sich leicht unterscheiden und wenn eine abweichende Strukturierung sinnvoller erscheint. Abbildung 2.23a zeigt die Modellierung eines Integrals in DSSL und die graphische Repräsentation 2.23c dieser Klasse. Wie zu erkennen ist, besteht die graphische Darstellung aus einer Reihe von geschachtelten Rechtecken, in denen die Bestandteile des Integrals eingefügt werden sollen. Die Standardabbildung von DSSL auf die Grammatik würde hier zu einer Baumstruktur führen, bei der die Attribute der Klasse `Integral` aus Abbildung 2.23a alle auf einer Ebene dargestellt werden. Die nochmalige Schachtelung, um eine Darstellung wie in Abbildung 2.23c zu erreichen, wäre nicht möglich. Um diese Strukturierung dennoch zu erhalten, kann eine textuelle Spezifikation wie in Abbildung 2.23b benutzt werden. Es entsteht dann ein Strukturbaum wie in 2.23d. Die Wurzel `Integral` ist der einzige Sprachkonstrukt-knoten der Spezifikation. Die grau hinterlegten Knoten stellen neue Zwischenknoten dar. Sie können beliebig benannt werden und beschreiben neu eingefügte Zwischenproduktionen der Grammatik, die dann auch benutzt werden können um Rollen visueller Muster anzuwenden.

In der textuellen Spezifikation in Abb. 2.23b werden Unterknoten in Klammern hinter ihrem Vaterknoten notiert. Zwischenknoten lassen sich von Attributknoten daran unterscheiden, dass hinter Zwischenknoten noch ein evtl. leeres Klammernpaar folgt.

### 2.9.3 Zusammenhang zur Repräsentationsgraphik

Editierbare und semantische Struktur sind in der Regel nahezu identisch. In einigen Spezialfällen können sich beide Strukturen allerdings unterscheiden. Dies ist beispielsweise der Fall, wenn es von einem semantischen Sprachkonstrukt in unterschiedlichen Sichten mehrere Repräsentanten gibt. Dies kommt in UML Diagrammen vor. Hier kann eine Klasse in mehreren Diagrammen enthalten sein. Die editierbare Struktur würde im zugehörigen Editor also noch einen zusätzlichen Repräsentanten der Klasse bereitstellen. Häufig möchte man auch Sprachen spezifizieren, die einen bestimmten Teil der semantischen Struktur mehrfach, jedoch in unterschiedlichen Layoutvarianten, darstellt. Weitere Anforderungen für die Trennung von semantischer und editierbarer Struktur nennt Schmidt in [101] (Seite 106 ff).

Um auch Sprachen mit unterschiedlichem Niveau von semantischer und editierbarer Struktur spezifizieren zu können, steht in DEViL ein so genannter asymmetrischer Kopplungsmechanismus zur Verfügung. Er erlaubt es, beide Strukturen aneinander anzupassen. Dazu stehen diverse Anpassungsschemata bereit, die flexibel bestimmte Sprachkonstrukt-knoten aneinander binden können. Asymmetrisch ist die Kopplung, da es möglich ist, beide Richtungen der Kopplung unterschiedlich zu behandeln. So kann beispielsweise spezifiziert werden, dass sich das Löschen eines Repräsentanten in der editierbaren Struktur nicht auf sein Pendant in der semantischen Struktur auswirkt. Schmidt beschreibt in [101] sieben Anpassungsschemata um die Kopplung zu definieren. In [127] wurde eine regelbasierte Spezifikationssprache zur Kopplung von Strukturen in DEViL entworfen.

Listing 2.7 zeigt einen Ausschnitt der Basisstruktur für einen Petri-Netz Editor, der das *Dining-Philosophers* Problem modelliert. Die Basisstruktur modelliert einfache Petri-Netze. Die abgeleitete Struktur aus Listing 2.8 koppelt an jede `PetriNet` Instanz eine `diningPhilosophers::PetriNet`-Instanz. Sie enthält genau die Attribute aus Listing 2.8 und überschreibt somit die entsprechende Klasse aus der Basisstruktur. Durch das Anpassungsschema `syncVal` wird sichergestellt, dass

das Größenattribut `setSize` in der abgeleiteten Struktur nicht gekoppelt wird.<sup>7</sup>

```

CLASS Root {
  petriNets: SUB PetriNet%;
  diningPhilosophers: SUB
    diningPhilosophers::PetriNet*;
}

CLASS PetriNet {
  nodes: SUB Node*;
  connections: SUB Connection*;
  setSize: VAL VLPoint? EDITWITH "None";
  name: VAL VLString;
}

ABSTRACT CLASS Node {
  position: VAL VLPoint? EDITWITH "None";
  name: VAL VLString;
}

...

```

**Listing 2.7:** Ausschnitt aus der Basisstruktur

```

DERIVED diningPhilosophers ROOT PetriNet {
  CLASS PetriNet {
    philosoph1: SUB Philosoph?;
    philosoph2: SUB Philosoph?;
    philosoph3: SUB Philosoph?;
    philosoph4: SUB Philosoph?;
    philosoph5: SUB Philosoph?;

    setSize: VAL VLPoint? INIT "500 500"
      EDITWITH "None";
    name: VAL VLString;
  }

  CLASS Philosoph {
    name: VAL VLInt;
    dishRef: REF Dish EDITWITH "None";
  }
  ...
  PetriNet.setSize.syncVal = 0;
}

```

**Listing 2.8:** Eine abgeleitete Struktur

Abbildung 2.24 zeigt, wie in DEViL sukzessive die graphische Darstellung aus der editierbaren Struktur berechnet wird. Auf der linken Seite sind dazu die notwendigen Transformationsschritte dargestellt. Dies führt letztendlich zur Rastergraphik, also der Darstellung, die durch den Anwender manipuliert werden kann. Rechts ist dargestellt, wie intern graphische Primitive in der Zeichenfläche auf Strukturobjekte abgebildet werden.

Die editierbare Struktur wird zunächst auf die Repräsentationsstruktur abgebildet. Diese enthält zusätzlich Methoden und Daten für die weiteren Transformationen. In DEViL sind dies die visuellen Muster. Die abstrakte Repräsentation beschreibt die wesentlichen Darstellungseigenschaften in einem konzeptionell sprachunabhängigen Kalkül. In der Layoutphase werden den Layoutattributen der graphischen Objekte ihre konkreten Werte zugewiesen. In der Regel sind dies Angaben über Positionen und Farben. Durch das Rendering wird durch das Betriebssystem die Darstellung auf dem Monitor erzeugt.

Wie auf der rechten Seite zu sehen ist, werden den Strukturobjekten des semantischen Modells eindeutige Kennungen zugeordnet, sodass Benutzerinteraktion (z. B. das Löschen eines Objekts) in der Rastergraphik auf das korrekte Strukturobjekt zurückgeführt werden kann. Dabei kann ein Strukturobjekt aus vielen graphischen Primitiven, z. B. mehreren Linien (spezifiziert im Editor für generische Zeichnungen) zusammengesetzt werden. Die Primitive haben dann keine eindeutige Kennung mehr, sie erben die Kennung des zugeordneten Strukturobjekts. Auch Attributkno-

<sup>7</sup>Die Simulation des Dining-Philosophers Problems ist in Abschnitt 7.4.1 auf Seite 156 zu finden.

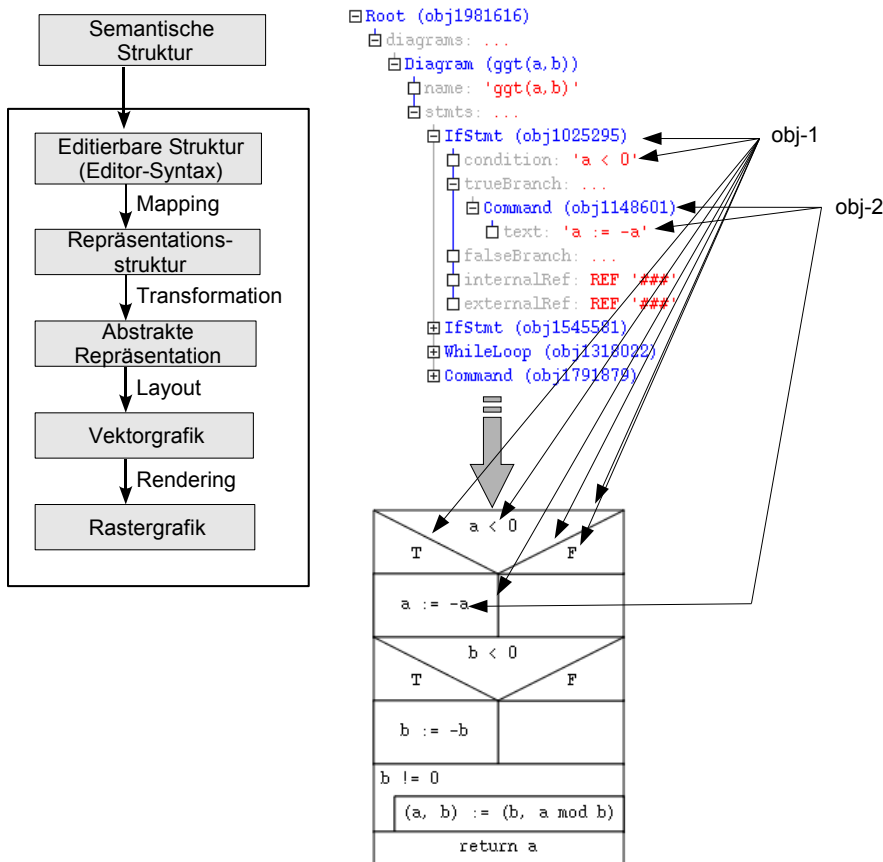


Abbildung 2.24: Abbildung auf die Repräsentationsstruktur

ten erben ihre Kennung von dem Knoten in dem sie definiert wurden. Prinzipiell ist es in von DEViL generierten Struktureditoren schwierig die Rückrichtung zu berechnen, da die Transformationen nicht bijektiv sind. Dies ist von Nachteil bei der Berechnung einer Animation, die alle Primitive eindeutig zuordnen muss, um Änderungen zu erkennen. In dieser Arbeit musste deswegen ein Mechanismus geschaffen werden um trotzdem Primitive eindeutig identifizieren zu können.

### 2.9.4 Das Klassenmodell und Pfadausdrücke in DEViL

Häufig ist es nötig, dass im Strukturbaum eines generierten Editors bestimmte Attribute aufgesammelt werden müssen, z. B. um eine Analyse zu bewerkstelligen.

Um im Strukturbaum der visuellen Sprache zu navigieren und Objekte aufzusammeln bzw. Attributeigenschaften zu lesen, stehen die so genannten *Pfadausdrücke* zur

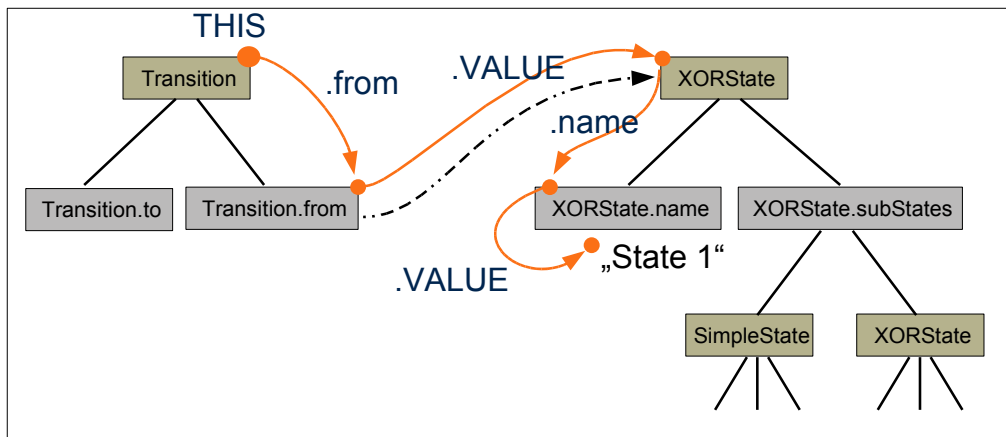


Abbildung 2.25: Pfadausdruck für `THIS.from.VALUE.name.VALUE`

Verfügung. Sie beschreiben in einer Spezialsprache, die sich an XPath orientiert, Bewegungen im Strukturbaum. Somit kann durch den Baum gewandert und Attribute bzw. Substrukturen abgefragt werden. In Abbildung 2.25 ist eine mögliche Instanz eines Strukturbaums der Spezifikation aus Abb. 2.22 zu sehen. Hier wird, ausgehend von einem `Transition`-Knoten auf den Namen des zugeordneten XOR-Knotens zugegriffen. Die Pfadausdrucksyntax besteht aus Konstrukten um Werte zu lesen und um auf Mengen von Objekten zuzugreifen. Eine genauere Beschreibung ist [101] zu entnehmen.

### 2.9.5 Eigenschaften generierter Umgebungen

Mit dem DEViL Generator lassen sich eine Vielzahl an unterschiedlichen Arten von Struktureditoren für visuellen Sprachen generieren. Struktureditoren mit 2,5-dimensionalen Darstellungen [40], die es erlauben z. B. Detailsichten zu einem Sprachobjekt zu öffnen und untereinander verbundene Diagramme zu spezifizieren, sind in DEViL eingeschränkt generierbar. Editoren mit dreidimensionalen Darstellungen lassen sich hingegen nicht spezifizieren.

Interaktionsbasierte Repräsentationen, deren wichtiges Layoutmittel die Exploration des Programms durch den Benutzer ist, können ebenfalls generiert werden. Dazu gehören Baumdarstellungen, deren Knoten durch den Anwender auf- bzw. zugeklappt werden können oder beispielsweise Sprechblasen, die sich öffnen, wenn der Benutzer mit der Maus über ein Sprachkonstrukt fährt.

Von DEViL generierte Editoren bieten dem Benutzer sowohl diskrete als auch kontinuierliche Layoutfreiheiten. Diskrete Layoutfreiheiten sind z. B. der Zeilenumbruch in einem textuellen Editor, der vom Benutzer bestimmt werden kann. Eine

kontinuierliche Layoutfreiheit ist die Anordnung von Mengenelementen, um für den Benutzer eine zusätzliche Semantik durch sekundäre Notation zu erreichen.

Die Interaktion zwischen Benutzer und DEViL-Editoren basiert darauf, dass Sprachobjekte in DEViL immer nur abhängig vom Kontext eingefügt werden können. DEViL bietet also je nach Editiersituation nur Sprachkonstrukte an, die auch laut Syntax der Sprache eingefügt werden dürfen. Dies hat den Vorteil, dass syntaktisch inkonsistente Strukturen nicht entstehen können. Daraus ergibt sich jedoch auch der Nachteil, dass temporär inkonsistente Strukturen nicht entworfen werden können, was zu einem erhöhten Aufwand bei der Erstellung einer Sprachinstanz durch einen Anwender führt. Editoren mit Parsing Techniken umgehen dieses Problem. Zusätzlich können Sprachkonstrukte mit der Maus "angefasst" und direkt manipuliert werden (*direct-manipulation*). Über Tastaturkürzel und Eigenschaften, die auch von Texteditoren bekannt sind, wie "Kopieren, Einfügen" oder "Ausschneiden" kann der versierte Benutzer die Editoren zügig benutzen.

Semantische und editierbare Struktur der Editoren haben in DEViL das gleiche Niveau. Unterschiede können durch Anpassung des Kopplungsmechanismus zwischen den beiden Strukturen ausgeglichen werden. Dadurch ist es möglich, ein Sprachkonstrukt in mehreren Sichten modifiziert darzustellen. In UML Diagrammen kann somit beispielsweise eine Klasse in mehreren Diagrammen (Sichten) dargestellt werden. Es existieren also mehrere Sichtrepräsentanten in der editierbaren Struktur für ein semantisches Objekt.

Die Schärfe der Editorsyntax beschreibt den Abstand zwischen syntaktisch und semantisch korrekten Programmen. In Struktureditoren können per Definition nur syntaktisch korrekte Programme erstellt werden. Eine allzu scharfe Syntax würde zu einem benutzerunfreundlichen Editor führen. Die Schärfe der Syntax in von DEViL generierten Editoren ist durch DSSL in etwa mit denen von UML Diagrammen vergleichbar. So können durch Baumstrukturen, Querbeziehungen und Kardinalitäten strukturelle Constraints definiert werden.

Die von DEViL generierten Editoren sind, bedingt durch den nicht-parsing basierten Ansatz, sehr ausdrucksstark. Eine vollständige Evaluation der Produktebene hinsichtlich der Bedienbarkeit findet sich in [101] und in [102].

**Relevanz im Kontext dieser Arbeit** DEViL erlaubt es mit klar getrennten Konzepten Struktur, graphische Repräsentation und Analyse zu beschreiben. DSSL zur Strukturspezifikation bietet sich in ähnlichen Konzepten auch zur Definition eines Simulationsmodells an. Es können Aggregation, Vererbung und Attributierung wiederverwendet werden. Die Pfadausdrücke, mit denen Strukturobjekte bestimmter Eigenschaften "gesammelt" werden können, bieten sich ebenfalls an um Simulations-



objekte mit gleichen Eigenschaften zu kapseln. Auf struktureller Seite fehlen DSSL Eigenschaften wie Funktionen und ein zufallsbasierter Baumaufbau.

Zur Spezifikation der graphischen Darstellung bietet DEViL eine mächtige parametrisierbare Musterbibliothek. Diese kann einfach (durch Kontrollattribute) aber auch beliebig angepasst werden. Das deklarative Konzept ist bewährt [102]. Die direkte 1:1 Beziehung zwischen Struktur und Repräsentation sowie die komplexen Algorithmen zur Darstellung stellen jedoch hohe Anforderungen im Hinblick auf die Implementierung einer Animationskomponente.



# 3 Simulier- und Animierbarkeit visueller Sprachen

## Inhalt

---

3.1	Struktur und Daten . . . . .	70
3.2	Zustand und Zustandsübergänge . . . . .	73
3.3	Repräsentation . . . . .	76
3.4	Klassifikation visueller Sprachen . . . . .	77

---

Voraussetzung für eine Simulation oder eine Animation ist zunächst, dass sie für eine gegebene Sprache auch sinnvoll ist und einen echten Mehrwert an Erfahrung bringt. Dies hängt stark von der Sprache, dem Entwickler- und Anwenderkreis sowie dem Teilgebiet ab, dem die Sprache zuzuordnen ist (vgl. auch Abschnitt 2.4). Ist dieser Aspekt geklärt, kann untersucht werden, ob und wie die Sprache mit Simulations- bzw. Animationsunterstützung ausgestattet werden kann.

In diesem Kapitel werden visuelle Sprachen untersucht, und es soll geklärt werden, welche Bedingungen für eine Sprache gelten, die simulier- bzw. animierbar ist. Dabei werden die Sprachen insbesondere auf ihre interne Struktur hin untersucht, d. h. es wird betrachtet, wie sich das Modell der Sprache von der Zustandsrepräsentation unterscheidet oder inwieweit beides miteinander verwoben ist. Es soll dabei im Folgenden herausgefunden werden, wie eine visuelle Sprache mit Simulations- und Animationsunterstützung beschaffen ist und unter welchen Bedingungen eine Instanz einer visuellen Sprache simuliert werden kann. Dazu werde ich den Begriff "Programm" genauer charakterisieren und ausweiten. Außerdem wird auf Besonderheiten einer Animation hingewiesen: z. B. können dies zusätzliche Effekte oder spezielle Animationsrepräsentationen sein.

### 3.1 Struktur und Daten

Im herkömmlichen Sinne der Programmiersprachen ist ein Programm "die Formulierung eines Algorithmus und der zugehörigen Datenbereiche sowie Darstellung von Prozessen und Objekten in einer Programmiersprache" [64]. Diese Definition sagt noch nichts über die Simulierbarkeit eines visuellen Programms aus. Insbesondere kann ein Programm im herkömmlichen Sinne noch nicht unmittelbar simuliert werden, da noch nichts über den Zustand und die Zustandsübergänge bekannt ist. Ein Programm ist lediglich eine konkrete Ausprägung der Syntax einer Sprache. In einer visuellen Sprache wird ein Programm durch eine Menge von visuellen Konstrukten beschrieben, die durch eine gegebene Syntax entsprechend angeordnet werden. Visuelle Konstrukte sind dabei vergleichbar mit Konstrukten aus textuellen Sprachen, wie z. B. bedingte Ausdrücke, Befehle etc.. Auch sie werden entsprechend einer gegebenen Syntax und Semantik angeordnet. Die Ausführungssemantik könnte z. B. durch eine *denotationale Semantik* formal definiert werden. Eine denotationale Semantik ordnet den Konstrukten der Sprache eine Wirkung zu. Auch in der Simulation einer visuellen Sprachinstanz wird den einzelnen Sprachkonstrukten durch eine Verhaltensbeschreibung eine Wirkung zugeordnet.

Der Fokus der Simulation, also die Frage, *was* simuliert und animiert werden soll, wirkt sich auf das strukturelle Level der Sprache aus. Ein Beispiel ist die Simulation eines Bubblesort-Sortieralgorithmus: Sollen die elementaren Tauschoperationen innerhalb eines Arrays flüssig animiert werden oder stellt man den Kontrollfluss

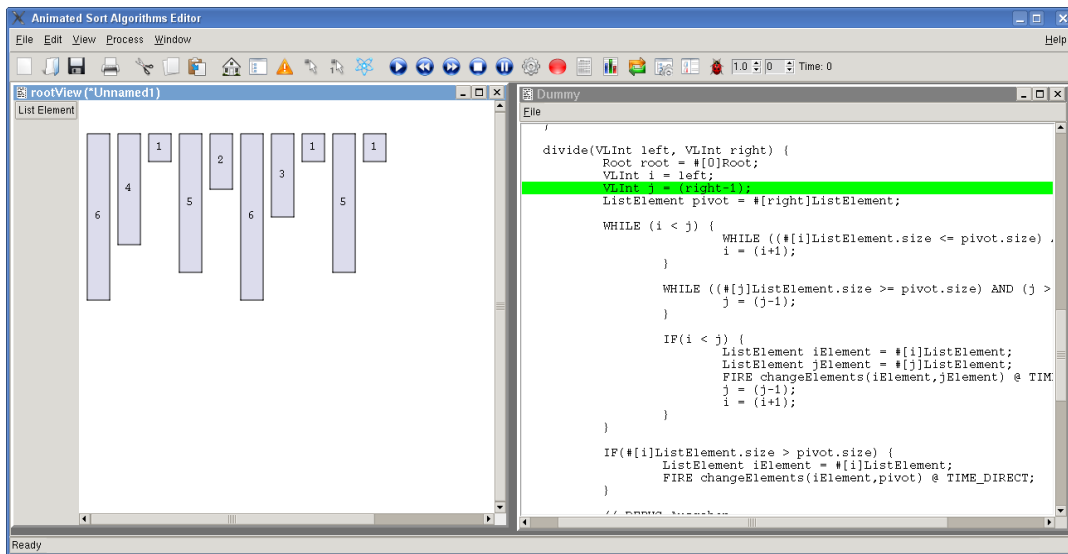
mitsamt der aktuellen Variablenbelegung im Quellcode des Algorithmus dar?

Ersteres würde bedeuten: Man benötigt eine Struktur, die verschieden große Datenelemente enthält, die wiederum visualisierbar sein müssen, z. B. als verschieden große Rechtecke. Die Animation zeigt dann den Tauschvorgang zweier benachbarter Elemente als weiche Animation. Die Programmbeschreibung hierfür - der Quellcode - ist in diesem Fall nicht von Interesse. Einzige Bedingung ist, dass auch wirklich Bubblesort implementiert wird.

Eine andere Sichtweise auf die Bubblesort-Simulation wäre die Code-orientierte: Der Fokus liegt dabei auf der Hervorhebung einer entsprechenden Zeile des Programmcodes, der Bubblesort mit einer beliebigen Eingabe simuliert. Dies ähnelt der Sichtweise eines Debuggers in einer imperativen Programmiersprache. In diesem Fall würde die Struktur des Programms aus Sprachkonstrukten bestehen, die Pseudocodeelemente darstellen. Die einzelnen Pseudocode-Sprachkonstrukte könnten entsprechend zu einem Programmtext zusammengestellt werden, der Bubblesort beschreibt. Wenn den einzelnen Programmkonstrukten eine Ausführungssemantik zugeordnet wird, könnte jedoch auch jedes andere beliebige Programm im Rahmen der Pseudocodesprache konstruiert werden. Als Eingabe würde der Algorithmus beispielsweise eine Reihe von Zahlen "konsumieren". Abbildung 3.1 verdeutlicht die Struktur- und Darstellungsunterschiede beider Varianten. Das Beispiel zeigt, dass Struktur und Daten, je nach Anwendungsfall, sehr unterschiedlich sein können. Liegt der Fokus auf den Tauschoperationen, die weich animiert werden sollen, so kann die zu Grunde liegende Sprachstruktur sehr flach sein. Lediglich die Eingabedaten (die zu tauschenden Elemente) werden modelliert.

Bei der Algorithmenvisualisierung ist die Struktur größer und wesentlich tiefer. Sprachkonstrukte haben eine eigene Ausführungssemantik, die vom Simulationsspezifizierer festgelegt wird. Der Benutzer kann diese selbst kombinieren und auf eine Eingabe anwenden lassen. Die Eingabe wird zur Laufzeit, z. B. durch einen modalen Dialog in die Simulationsstruktur geladen.

Die Simulation wird komplexer, wenn der Benutzer *beliebige* Sprachkonstrukte (mit Ausführungssemantik) in einem visuellen Programm kombinieren kann und diese dann auf einen anderen dedizierten Teilbaum angewendet werden, der als Eingabe dient und den Programmzustand speichert. Als Beispiel ist hier eine Verkehrssimulation zu nennen, bei der der Benutzer selbst innerhalb des Editors Verkehrsregeln, wie "rechts vor links" definieren kann, die dann auf das Verkehrsgeschehen angewendet werden. Das Verkehrsgeschehen stellt dann den aktuellen Systemzustand dar. Hier müssen Strukturen und Daten in einem Teilbaum des Programms erkannt werden. Dies entspricht in gewisser Weise einer recht aufwändigen Mustererkennung und wird in *Programming-by-Demonstration*-Systemen wie *Agentsheets* [92] angewendet. In *Agentsheets* kann die Programmierung visuell durch Vorher-/Nachher-Regeln erfolgen. Im Hintergrund arbeitet dann ein System



**Abbildung 3.1:** Ein Struktureditor zur Simulation des Quicksort-Algorithmus. Links die datenorientierte Sicht. Eine Animation zeigt Vertauschungsoperationen zwischen Elementen. Rechts die codeorientierte Variante, die den Fokus auf den Algorithmus legt. Beide Ansichten sind in von DEViL generierten Editoren möglich.

zur Auswertung dieser Regeln. Der Simulator muss also nicht mehr von Hand entworfen werden, da er auf das regelbasierte Kachelparadigma spezialisiert ist.

Wie weit Simulation und Animation strukturell voneinander entfernt sind, wirkt sich ebenfalls auf das Spezifikationskalkül aus. In einigen Sprachen fallen Simulations- und Animationsstruktur zusammen. Zustandsübergänge sind in der Sprache definiert. Hier sind wieder Petri-Netze zu nennen, wobei das Programm aus den Daten, repräsentiert durch Marken, zusammen mit der Struktur, d. h. den Stellen, Transitionen und Verbindungen, modelliert wird. Eine Animation ist dann die kontinuierliche Bewegung der Marken.

Betrachtet man nun eine Sprache, die Verkehrsampeln einer Kreuzung durch ein gegebenes Petri-Netz simuliert, so möchte man evtl. in einem ersten Schritt die Stellen des Petri-Netzes in den unterschiedlichen Farben der Verkehrsampel färben. Dies ist eine rudimentäre Darstellung, die jedoch noch kein domänenspezifisches Layout beinhaltet. Ein domänenspezifisches Layout wäre die Darstellung der Kreuzung mit den Ampeln. Durch das bereits vorgestellte Kopplungskonzept von DEViL (Seite 62) kann eine Struktur, die eine Verkehrsampel darstellt, an Teile des Petri-Netzes gekoppelt werden. Markenbewegungen des Petri-Netzes führen dann gleichzeitig zu einer Änderung der Ampel-Darstellung. Somit kann für die Struktur einer visuellen Sprache mit Simulations- und Animationskomponente gesagt werden:

*Simulation und Animation fallen im einfachen Fall zusammen, in komplexen Fällen können gekoppelte Strukturen eingesetzt werden.*

Petri-Netze haben eine Modellierungseigenschaft, d. h. mit ihnen können andere Systeme formal spezifiziert werden. Vorteilhaft ist hier, dass die Domäne sehr eingeschränkt ist, da nur vier unterschiedliche Sprachkonstrukte benutzt werden können. Problematisch wird es, wenn diese Einschränkung nicht mehr gilt, d. h. wenn Sprachen mit *allgemeinen* Modellierungseigenschaften betrachtet werden, z. B. Java. Allgemein bedeutet, dass jede beliebige Anwendung durch ein Programm dieser Sprache ausgedrückt werden kann. Ein beliebiges Java-Programm (oder eine andere GPL) und dessen Ausführung sind somit das andere Ende des Unterschiedes zwischen Simulations- und Animationsstruktur. Die Darstellung, also der Java-Quelltext, unterscheidet sich erheblich von dem ausgeführten Programm. Allerdings ist Java auch keine domänenspezifische Sprache mehr und fällt somit auch nicht in die Kategorie der Sprachen, die hier simuliert und animiert werden sollen.

Wie eine visuelle Sprache strukturell implementiert wird, liegt häufig auch an den Vorlieben des Designers. Ein Unterschied liegt in der Regel darin, ob eher flache oder tiefe Baumstrukturen bevorzugt werden. So kann etwa die Markenbelegung eines Petri-Netzes als Attribut der Stelle modelliert werden oder als eigener Strukturknoten, der in der Stelle aggregiert enthalten ist. Die Simulation würde bei der ersten Variante das Attribut der Stelle entsprechend in- oder dekrementieren. Bei der zweiten Variante müsste das Sprachkonstrukt strukturell bewegt werden. Dies hat direkte Konsequenzen für die Animation: Im ersten Fall muss ein Animationsobjekt in Form einer Marke erstellt und bewegt werden, das keinem Sprachkonstrukt der Simulationsstruktur zugeordnet ist. Im letzteren Fall stellt die Animation direkt die Bewegung der Marke dar. Um dem Entwickler freie Wahl bei der Implementierung zu lassen, sollten beide Fälle spezifizierbar sein und eine Animation sollte auch in beiden Fällen gleich aussehen. Den ersten Fall, bei dem keine Sprachkonstrukte bewegt werden, nenne ich im Folgenden *attributorientiert*, die Simulation für tiefe Strukturen nenne ich *strukturorientiert*. Struktur- und attributorientierte Simulation sind natürlich auch kombinierbar.

### 3.2 Zustand und Zustandsübergänge

Um ein Programm einer visuellen Sprache zu animieren, benötigt man einen Zustandsübergangsautomaten, der Zustände des Programms erkennt und Zustandsübergänge auslöst. Dies ist nur möglich, wenn die Sprache eine wohldefinierte Ausführungssemantik hat. Sie beschreibt, wie ein gegebener Programmzustand in einen anderen überführt werden soll. Diese Transformation ist bei den meisten Sprachen diskret, d. h. es gibt für einen Zustandsübergang nur zwei adjazente

Zustände, zwischen denen das Programm in einem diskreten Schritt überführt wird. Die Animation zeigt dann eine Interpolation zwischen den beiden Zuständen an. Während dieser Interpolation ist das Programm quasi in einem ungültigen Zwischenzustand. In manchen Editoren (vgl. DiaMeta, Abschnitt 2.7.3 auf Seite 44) korrespondiert dieser Zustand jedoch auch mit dem Zustand des zu Grunde liegenden Graphtransformationssystem. Ein Animationszustand ist dann auf eine Graphinstanz abbildbar. Diese Systeme eignen sich dann nicht nur für diskrete Simulationen, sondern auch für kontinuierlicher Modelle. In visuellen Sprachen kommen solche kontinuierlichen Darstellungen vor allem bei der Positionierung von Mengenelementen vor. So kann durch räumliche Nähe Beziehungen zwischen verwandten Elementen ausgedrückt werden (sekundäre Notation).

Die obige Betrachtung zur Struktur visueller Sprachen zeigt, dass Zustände innerhalb einer Sprache ganz unterschiedlich definiert sein können. Sie können Teil der visuellen Sprache sein, in einem separaten Teilbaum gespeichert werden und auch strukturell unterschiedlich komplexe Ausprägungen haben.

Sprachen, die dynamische Systeme modellieren, definieren den Zustand häufig schon innerhalb des Programms selbst. Petri-Netze repräsentieren den Zustand durch Marken. Pictorial Janus (vgl. Abschnitt 2.6.7 auf Seite 37) repräsentiert den Zustand durch Listenelemente unterschiedlicher Länge und ein Aufblähen der aktuellen Vorbedingung. Eine Spielesprache wie Pac-man<sup>1</sup> beschreibt den aktuellen Spielzustand über die Position und Darstellung der einzelnen Spielfiguren. Zustände sind hier häufig relativ einfache Sprachkonstrukte. Diese lassen sich gut darstellen und haben eine eingeschränkte Bedeutung.

Eine Sprache wie UML-Statecharts modelliert ebenfalls dynamische Systeme, sie definiert jedoch nur ihre Zustandsübergänge durch benannte Kanten. Der aktuelle Zustand kann einer Programminstanz nicht angesehen werden. Die so genannte *Zustandsnotation* kann hier jedoch leicht nachgerüstet werden, indem jedem Knoten des Statecharts ein Attribut zugeordnet wird, das angibt, ob der Knoten aktiv oder inaktiv ist. Zustandsübergänge werden hier durch externe (Tastatur-)Ereignisse geschaltet. Aus einer Programminstanz lässt sich also nicht unmittelbar der Folgezustand wie bei Petri-Netzen oder Pictorial-Janus-Programmen berechnen. Hier kann zunächst nur eine mögliche Menge von Folgezuständen angegeben werden. Der Benutzer des Programms wählt aus dieser Menge dann direkt einen Folgezustand aus. Eine andere Variante wäre, Zustandsübergänge nicht durch den Benutzer schalten zu lassen, sondern durch eine nicht-deterministische Auswahl eines möglichen Folgeereignisses.

Die Simulation einer visuellen Sprache kann entweder *universell* für alle Programminstanzen oder *modellspezifisch* sein, d. h. nur eine Teilmenge aller möglichen

---

<sup>1</sup>Pac-man ist eingetragenes Warenzeichen der Firma Namco.



Programme der Sprache kann simuliert werden. Für den Zustandsübergangsautomaten bedeutet dies, dass im letzteren Fall nur Übergänge ausgelöst werden, wenn eine ganz bestimmte Programminstanz erkannt wird. Eine Sprache mit universeller dynamischer Semantik hingegen kann für jede Programminstanz den Folgezustand bestimmen. In der Literatur wird dies auch als *Interpreter Semantik* beschrieben (vgl. [31]). Sprachen mit nicht-universeller Semantik werden auch als Sprachen mit *Übersetzer Semantik* bezeichnet. Auch hybride Sprachen sind durchaus anzutreffen. Beispiele sind Ampelsteuerungen oder das "Dining-Philosophers"-Problem, welche beide durch Petri-Netze modelliert werden können. Dazu kann ein Zustandsübergangsautomat für universelle Petri-Netze benutzt werden, der durch einen (Kopplungs-)Mechanismus Änderungen an der anwendungsspezifischen Sicht beschreibt.

Die Komplexität von Zustandsübergängen kann sehr unterschiedlich sein. Im einfachsten Fall sind Zustände im Programm repräsentiert und Zustandsübergänge universell definiert. Dies reicht bei komplexen Simulationen nicht aus und Berechnungen müssen zwischengespeichert werden. Auch hier kann im einfachen Fall die Simulationsstruktur durch das Einführen neuer persistenter Attribute angepasst werden. Dies ist manchmal auch nötig um den globalen Programmzustand zu speichern und Unterprogramme aufzurufen. In Graphtransformationssystemen muss hier die Graphgrammatik künstlich erweitert werden um kaskadierende Ersetzungsregeln aufzurufen.

Bei der Frage *wer* Zustandsübergänge beschreibt, handelt es sich also um den Anwender oder den Entwerfer der Sprache, gibt es mehrere Varianten. Neben universellen Zustandsübergängen können Zustandsübergangsdefinitionen sich auch in die Programminstanz selbst verlagern. Der Benutzer programmiert in diesem Fall selbst Zustandsübergänge. Dazu benutzt er Konstrukte der Sprache, die eine bestimmte dynamische Semantik haben und setzt sie zu einem Programm zusammen. Diese "höhere" Semantik ist wiederum durch den Sprachspezifizierer festgelegt worden. Logo ist so eine Sprache, bei der imperative Anweisungssequenzen auf ein spezielles Sprachkonstrukt (eine Schildkröte) angewendet werden. Dieses Sprachkonstrukt stellt den Zustand durch Position und Rotationswinkel dar. Es befindet sich in einem anderen Teilbaum des Programms der visuellen Sprache als die Anweisungssequenzen.

Die Simulation wird umso komplexer, je aufwändiger die Semantik der Regeln ist. Sind diese tief strukturiert und müssen sie auf eine Reihe anderer Sprachkonstrukte angewendet werden, so steigen die Anforderungen an die Simulationssprache.

Wie Zustandsübergänge ausgelöst werden, ist ein weiteres Unterscheidungsmerkmal. Für Sprachen mit universeller Ausführungssemantik kann der Folgezustand in der Regel ohne weiteres ermittelt werden. Nicht-Determinismus kann hier allerdings auch entstehen. Eine Methode diesen aufzulösen besteht darin, den Anwender zu

fragen. Bei Petri-Netzen kann dies passieren, wenn zwei Transitionen gleichzeitig schalten können. Dem Anwender wird dann ein modaler Auswahldialog angezeigt. Modal bedeutet hier, dass der Zustandsübergangsautomat zwingend eine Eingabe benötigt um weiter arbeiten zu können.

In anderen Fällen lösen Tastaturereignisse einen Zustandsübergang aus, z. B. durch die Betätigung einer Taste um in einem Statechart den nächsten Zustand zu aktivieren. Nicht modale Ereignisse werden somit auch nicht zwingend für eine laufende Simulation benötigt. Die Steuerung des Pac-man durch den Anwender ist z. B. so ein nicht-modales Ereignis. Ohne dieses läuft die Simulation trotzdem ab, die anderen Spielfiguren werden durch den Computer bewegt.

Ein Benutzerereignis kann andere Ereignisse verdrängen, evtl. weil eine Berechnung nicht mehr gültig ist oder aufgrund anderer Effekte nicht mehr ausgeführt werden kann. Anzumerken ist, dass Zustände als Daten in der Sprache oder als Eingabedaten durch den Benutzer definiert werden können.

### 3.3 Repräsentation

Die graphische Notation des Zustands ist neben wohldefinierten Zustandsübergängen nötig, um eine geeignete Animation zu finden. Die graphische Notation drückt den aktuellen Systemzustand visuell in der Sprache aus. Sprachen mit universell definierten Zustandsübergängen haben diesen meist bereits integriert. Aber auch bei Sprachen, die den Zustand nicht von Haus aus graphisch ausdrücken können, kann dies nachgerüstet werden. Dies liegt dabei häufig in der Kreativität des Animationsentwerfers. Aktive Zustände in Statecharts können so beispielsweise durch eine grüne Hintergrundfarbe dargestellt werden. Aber auch andere Repräsentationen sind hier denkbar, z. B. durch Pfeile, die auf den aktiven Zustand hindeuten.

Simulation und zugehörige Animation einer visuellen Sprache können auch eine völlig andere Darstellung haben. In Petri-Netzen kann dies die bekannte Markensimulation sein, aber auch ein domänenspezifische Layout ist denkbar. Dies könnte wie beim Produzent/Konsument Beispiel eine Mutter (Produzent) sein, die für ihr Kind (Konsument) einen Kuchen (Marke) backt.

In Generatorsystemen wie AToM<sup>3</sup> (vgl. Abschnitt 2.7.5 auf Seite 48) werden beide Layoutvarianten in einem Objekt-Repository zu einer verallgemeinerten Sprache vereint und darauf simuliert. In GenGed (vgl. Abschnitt 2.7.2 auf Seite 43) wird eine zusätzliche Transformationsgrammatik angegeben, die aus einer Instanz des Petri-Netzes ein konkretes DSL-Layout erstellt. Auf diesem können dann Simulationsregeln definiert werden. In DEViL leistet dies auch wieder das Kopplungskonzept. Diese Distanz zwischen visueller Sprache und Simulation

bzw. Animation kann jedoch noch größer sein. Beispiele hierfür sind strukturierter Programmtext, der in ein beliebiges ausführbares Programm übersetzt wird oder auch eine regelbasierte Spielesprache, die als visuelles Konstrukt nur eine Reihe von Vorher-/Nachher-Regeln kennt und diese dann auf die Zustandsrepräsentation, ein konkretes Spielfeld mitsamt Spielfiguren, anwendet.

Um die Distanz zwischen Programm und Programmausführung zu verringern setzen manche Animationsdesigner auch hinweisende Animationen ein, die sich nicht direkt auf die aktuelle Simulationssituation zurückführen lassen aber aus ihr abgeleitet werden können. Einfache Varianten davon sind schlichte Hinweistexte, die in einem Nachrichtenfenster erscheinen oder auch direkt in die Zeichenfläche der Animation eingebunden werden können. Anspruchsvoller sind kleine *Sub-Animationen*, wie sie in einem Editor vorstellbar wären, der den Datenpfad einer CPU simuliert: Hier werden Datenimpulse durch einen kleinen Ball dargestellt, der entlang einer Verbindung läuft. Das Besondere ist, dass diese Animation keinen Repräsentanten in der Struktur hat, d. h. alle Animationen wurden bisher durch das Verändern von Strukturobjekten, z. B. einer Bewegung, ausgelöst. Diese Verbindung ist bei solchen hinweisenden oder analytisch aus der Simulation berechneten Animation nicht mehr der Fall. Ein anderes Anwendungsszenario sind die bereits vorgestellten *Effect-Lines*, die Geschwindigkeit für sich bewegende Objekte verdeutlichen. Auch hier haben die einzelnen Linien keinen Strukturrepräsentanten.

Häufig basiert die Repräsentation einer Animation auch auf dem Kontext, in dem ihr zugehöriges Sprachkonstrukt eingebettet ist. Je nachdem wo sich ein Strukturobjekt innerhalb des Modells befindet, wird eine unterschiedliche Animation ausgelöst. Hier muss die Animation auf Daten des Modells zurückgreifen.

## 3.4 Klassifikation visueller Sprachen

Aus den obigen Überlegungen lassen sich Typen von visuellen Sprachen ableiten und eine Aussage darüber treffen, ob und wie sie simuliert bzw. animierbar sind. Im Folgenden werde ich eine Klassifizierung vorstellen und diese anhand konkreter Sprachen verdeutlichen. Durch diese Einordnung kann für andere Sprachen abgeschätzt werden, wie hoch der Aufwand einer zusätzlichen Simulation und Animation ist.

Generell kann gesagt werden, dass Sprachen, die simuliert und animiert werden sollen, Folgendes benötigen: eine **Notation des Zustands**, die zum einen nötig ist um den Zustand graphisch zu repräsentieren und zum anderen um den Folgezustand zu beschreiben. Entweder ist diese Zustandsnotation bereits vorhanden, wie bei Petri-Netzen durch die Markenbelegung oder die Zustandsnotation ist integrierbar durch zusätzliche graphische Elemente, wie bei Statecharts (Aktivität

eines Zustands wird repräsentiert durch die Hintergrundfarbe) bzw. durch neue Strukturen, wie beim Logo Interpreter (durch Hinzufügen einer Cursor-Struktur).

Des Weiteren werden **wohldefinierte Regeln für Zustandsübergänge** gebraucht. Diese sind entweder universell wie bei Petri-Netzen und Statecharts oder sie können durch den Benutzer durch die Definition von Regeln in der Sprache festgelegt werden (z. B. Logo-Anweisungssequenzen).

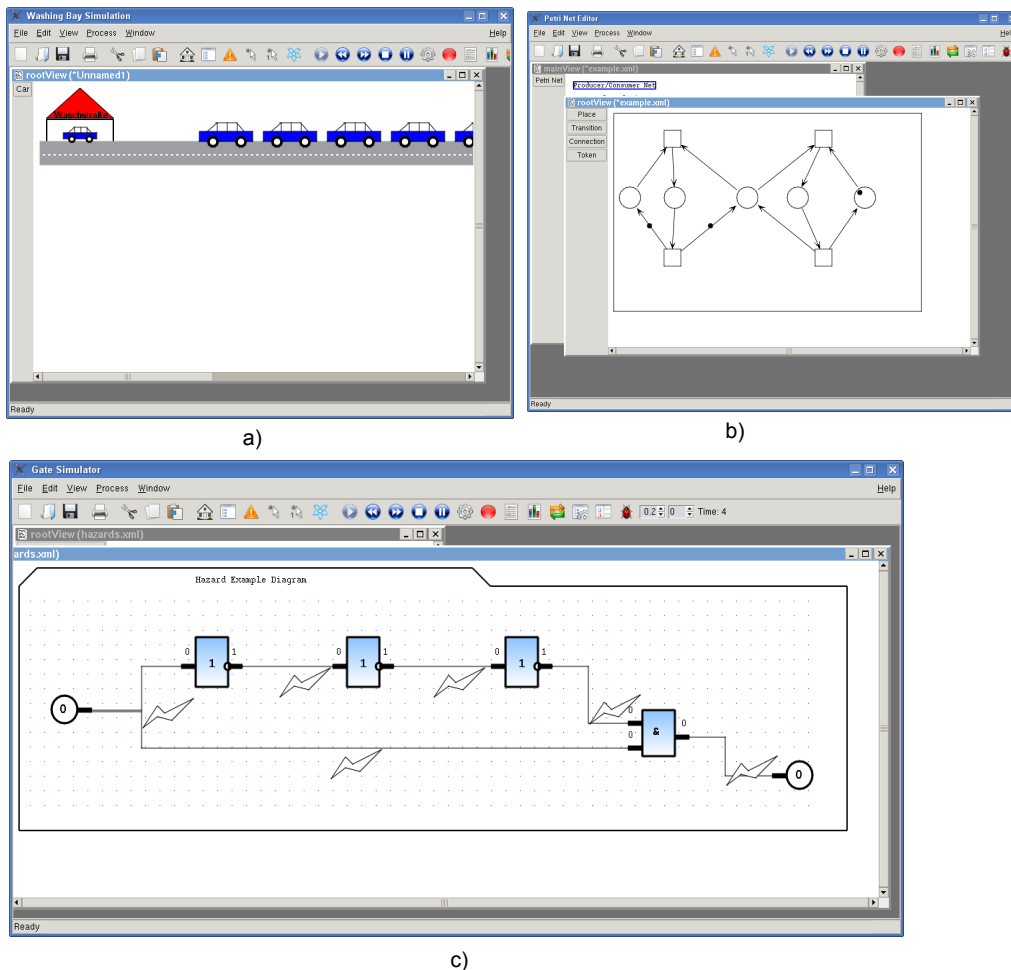
Insgesamt lassen sich durch die vorangegangenen Betrachtungen eine Klassifikation von Sprachen hinsichtlich ihrer Simulier- und Animierbarkeit erstellen. Die Einordnung sagt aus, wie weit sich Animation von der Simulation unterscheiden und welche Methodik angewendet werden muss, um eine anspruchsvolle Animation zu erstellen. Man kann folgende Kategorien unterscheiden:

1. Sprachen, die den Programmzustand inhärent definieren und beispielsweise universelle Zustandsübergänge haben, sowie zusätzlich bereits eine graphische Notation ihres Zustands mitbringen. Hier ist keine Erweiterung von Sprach- oder Simulationsstruktur notwendig.
2. Sprachen bei denen die Simulationsstruktur erweitert werden muss um Daten zwischenspeichern oder den Programmzustand zu merken sowie Sprachen, die eine Erweiterung der editierbaren Struktur benötigen um die graphische Notation des Zustands darzustellen.
3. Sprachen, bei denen eine erhebliche Distanz zwischen Simulations- und Animationsstruktur vorhanden ist. Die Animationsstruktur ist dabei mittels Strukturkopplung an die semantische Struktur gebunden sein. Die Animation kann hier in der Regel fast automatisch mit dem Strukturkopplungsprinzip hergeleitet werden.
4. Sprachen, deren Ausführungssemantik innerhalb der Sprache selbst definiert ist. Der Sprachanwender kann hier selbst ein Programm aus Konstrukten zusammensetzen, die Semantik tragen, häufig sind dies visuelle Regeln. Diese können entweder mit der Animationsstruktur zusammenfallen oder in einem anderen Teilbaum beheimatet sein.
5. Sprachen, die sich nicht simulieren bzw. animieren lassen, da Programm- und Programmausführung dieselbe Darstellung haben. Die Sprache definiert hier keine Zustandsübergänge. Außerdem gehören hierzu Sprachen bei denen Programm- und Programmausführung eine erhebliche Distanz haben, die nicht oder nur mit erheblichen Aufwand überbrückt werden kann.

Die *erste Kategorie* sind Sprachen mit universellen Zustandsübergängen, die graphische Notation ist ebenfalls integriert. Die Simulationsstruktur fällt hier mit der semantischen Struktur zusammen. Charakteristisch ist, dass Strukturobjekte durch die Simulationsstruktur "wandern". Mit Hilfe von Zufallsvariablen lassen sich so auch Warteschlangensimulationen einfach entwickeln. Für die visuelle Sprache ist eine Animation unmittelbar ersichtlich, so definieren Petri-Netze den Markenflug.

### 3.4. KLASSIFIKATION VISUELLER SPRACHEN

Die Simulation elektronischer Schaltkreise benutzt "fliegende Einsen" als Metapher um Stromfluss zu verdeutlichen. Die Animation kann selbstverständlich noch erweitert werden. So blinken die schaltenden Transitionen im hier entwickelten Editor für Petri-Netze. Für die Analyse ist die Animation in solchen Sprachen das primäre Mittel. Trotzdem sind auch Untersuchungen denkbar, die sich nicht direkt aus der Sprache ergeben und erst durch eine mehrmalige und zufallsbasierte Ausführung der Simulation ergeben. In der Waschstraßensimulation ist dies ein Auslastungsgraph der Warteschlange bzw. die durchschnittliche Wartezeit eines Autos wiedergibt. Mit der später vorgestellten Analysekomponente lässt sich dies leicht und fast automatisch realisieren. Abbildung 3.2 zeigt drei generierte Struktur-editoren dieser Kategorie.



**Abbildung 3.2:** Beispiele visueller Sprachen der *ersten Kategorie*. Eine zufallsbasierte WaschstraÙe (a), Petri-Netze (b) und ein Editor zum Modellieren elektronischer Schaltkreise (c)

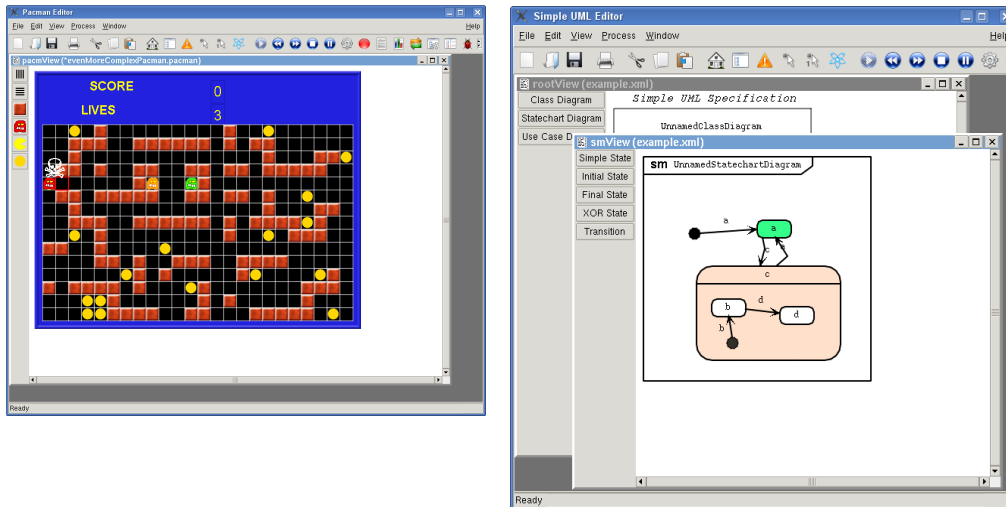
In der *zweiten Kategorie* können Zustandsübergänge nicht allein aus dem semantischen Sprachmodell berechnet werden. Es werden zusätzliche Attribute benötigt, die den Programmzustand zwischenspeichern. In der Quicksort-Implementierung sind dies die Zeiger auf die linke bzw. rechte Grenze sowie ein Zeiger auf das Pivotelement. Werden erweiterte Attribute nicht nur für die Simulation benötigt, sondern auch um die graphische Zustandsdarstellung zu berechnen, so können diese Attribute in die editierbare Struktur gezogen werden. Dies ist der Fall bei der Simulation des Statecharts. Die Aktivität/Inaktivität eines Zustands wird durch ein Boole'sches Attribut in der Simulationsstruktur ausgedrückt. Da aus diesem Attribut jedoch noch die Hintergrundfarbe berechnet wird, kann es in die editierbare Struktur gezogen werden.

In dieser Kategorie liegen Sprachen, die zur Simulation zusätzlich Benutzerinteraktion verwenden. Entweder modal oder kontinuierlich. Im generierten Pac-man Editor wird eine kontinuierliche Interaktion genutzt um den Pac-man zu steuern. Der Computer bewegt die restlichen Spielfiguren. Ähnlich verhält es sich bei der Implementierung des *"Mensch-ärger-Dich-nicht"*-Editors, der halbautomatisch abläuft. Der Computer steuert drei Spielfiguren, die vierte wird durch den Anwender bewegt. Dieser wählt, falls mehrere Figuren ziehen können, eine mögliche Spielfigur aus. Der Statechart-Editor erwartet explizit Tastatureingaben um Zustandsübergänge zu triggern. Somit muss die Eingabe als Teil der Daten einer Simulation betrachtet werden. Gegebenenfalls müssen Eingabedaten entsprechend transformiert werden, bevor sie verarbeitet werden können.

Da in diese Kategorie Sprachen mit komplexerer Simulation fallen, ist auch die Animation anspruchsvoller. Die Animation des Pac-man kann verbessert werden, indem er sich bei einer Richtungsänderung vor der Bewegung entsprechend dreht. Für Statecharts sind hier die Farbübergänge zu nennen, eine Startfarbe wird langsam in eine Zielfarbe überführt (siehe auch Abb. 3.3 rechts). Dies hat den Sinn, auf einen sich sonst sehr schnell auswirkenden Zustandsübergang hinzudeuten. Ebenfalls hinweisend wirken sich zusätzliche Animationen aus. In der Quicksort-Implementierung deuten Pfeile auf Pivotelement und Grenzen hin. In der Pac-man Animation wird ein sich bewegendes Totenkopf angezeigt, wenn der Pac-man gefangen wird. Diese Animationstypen haben keinen Repräsentanten in editierbarer oder Simulationsstruktur, lassen sich aber ähnlich wie die anderen Animationen spezifizieren und nutzen.

Die *dritte Kategorie* der Klassifizierung bilden Sprachen mit erheblicher Strukturdifferenz zwischen Simulation und Animation. In dieser Kategorie liegen Sprachen, die eine Modellierungseigenschaft besitzen: d. h. mit ihnen lässt sich die Semantik anderer komplexer Systeme beschreiben. Es ist also schon implizit klar, dass auch ein anwendungsspezifisches Layout möglich ist. Beispiel ist auch hier wieder das Petri-Netz, mit dem sich Ampelsteuerungen oder das bekannte *"Dining-Philosophers"*-Problem modellieren lassen (Abb. 3.5). Interessant ist hier, dass die

### 3.4. KLASSIFIKATION VISUELLER SPRACHEN



**Abbildung 3.3:** Beispiele visueller Sprachen der *zweiten Kategorie*. Links im Bild ein Pac-man Editor. Der Pac-man wurde gerade von einem roten Geist gefangen. Als dynamisches Animationsobjekt wird ein sich bewegender Totenkopf angezeigt. Das rechte Bild zeigt die Simulation eines Statecharts. Das Diagramm befindet sich gerade im Zustandsübergang von a nach c. Die Hintergrundfarbe von Zustand a wird langsam wieder weiß (inaktiv), Zustand c wird langsam grün (aktiv)

Sprache universelle Zustandsübergänge definiert. An ein konkretes Programm lässt sich dann eine anwendungsspezifische Sicht koppeln. Zustandsübergänge in der Basisstruktur triggern dann Änderungen in der gekoppelten Struktur. Diese lassen sich wiederum durch Animationen geeignet visualisieren. Die Animation kann mit DEViLs Strukturkopplungsprinzip automatisch hergeleitet werden.

Die *vierte Kategorie* bilden Sprachen, die es erlauben die dynamische Semantik als Teil des Programms zu definieren. Der Benutzer kann Sprachelemente mit Ausführungssemantik auswählen und zusammenstellen um selbst ablaufende Programme definieren. Ein schönes Beispiel ist die Lernsprache *Logo* [65] mit der durch imperative Anweisungen ein graphischer Cursor (meist in der Form einer Schildkröte) bewegt werden kann. Die Anweisungen in Form von Kommandos wie VORWÄRTS 100, RECHTS 90, ... ändern die Position und den Rotationszustand des Cursors. In der Sprache lassen sich keine Zuweisungen oder Variablen definieren. Die Anweisungen selbst sind Sprachkonstrukte, die in der Simulationsspezifikation eine hinterlegte dynamische Semantik haben.

Simulations- und Animationsstruktur können in zwei verschiedenen Teilbäumen beheimatet sein. Die Semantik der Sprachkonstrukte wird gewissermaßen

interpretiert und auf ein dediziertes Sprachkonstrukt angewendet. Einen Sonderfall in dieser Kategorie bildet *Pictorial Janus*: Animations- und Simulationsstruktur fallen zusammen. Regeln, mit denen PJ-Programme erstellt werden können, werden quasi auf sich selbst angewendet. Erschwerend kommt in *Pictorial Janus* noch die *Selbstähnlichkeit* hinzu. Subkonfigurationen sind innerhalb einer Regel prinzipiell beliebig tief enthalten, werden jedoch nur bis zu einer gewissen Tiefe visualisiert.

Interessant in dieser Kategorie ist der CPU-Datenpfad Editor (inspiriert durch [34]). Hier wandern Instruktionen durch das Simulationsmodell und je nach Position werden unterschiedliche Ereignisse ausgelöst. So bewirkt eine Additionsinstruktion im Decoder etwas anderes als in der arithmetischen Einheit. Dies führt später zum Konzept der *kontext-sensitiven Funktionen*.

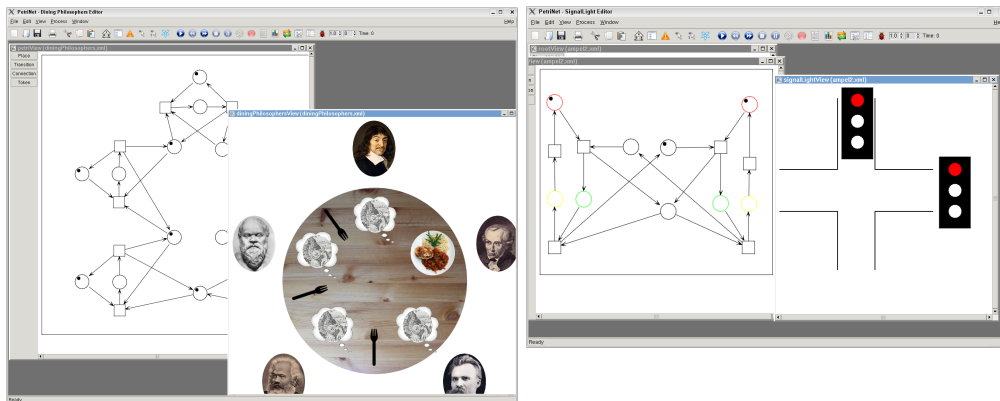
Komplizierter zu behandeln sind Sprachen, bei denen die für die Simulation wichtigen und Semantik tragenden Sprachkonstrukte nicht nur auf ein einzelnes dediziertes Sprachelement angewandt werden und dessen Zustand verändern, sondern wenn beliebig viele dieser Elemente existieren. Dann müssen in einem Teilmodell Sprachkonstrukte, die bestimmte Eigenschaften aufweisen, z. B. sich in einer ganz speziellen Umgebung befinden, identifiziert werden. Diese Mustererkennung ist aufwändig und nicht leicht abzubilden. Systeme wie *Agentsheets*, die sich auf diese Domäne spezialisiert haben, sind hier klar im Vorteil. Sie haben jedoch den Nachteil, keine anderen Systeme außer kachelbasierten simulieren zu können.

Für die Simulationssprache ergibt sich hieraus die Konsequenz, Mechanismen einzubauen um Sprachkonstrukte mit spezifischen Eigenschaften, wie Attributbelegungen oder Kontext, zu identifizieren. Die in DEViL integrierten Pfadausdrücke arbeiten hier eher strukturell. Deswegen wurde zusätzlich ein XPath-Auswerter [129] integriert um auch komplexere Muster und konkrete Wertbelegungen zu erkennen.

Für die Animation lassen sich in dieser Kategorie einige interessante Beobachtungen machen. In der Logo-Animation fallen die Spurlinien auf, die die Schildkröte während der Bewegung hinterlässt. Diese sind nicht flüchtig, sondern überdauern die Simulationsschritte. Vergleichbar ist dies mit den bereits erwähnten *Effect-Lines* zur Darstellung von Geschwindigkeit in Comic-Animationen. Sie werden nicht, wie der Ball, der Datenimpulse in der CPU-Simulation repräsentiert, bewegt oder rotiert, sondern statisch auf die Zeichenfläche gelegt. In dieselbe Kategorie fallen hinweisende Texte, die die aktuell ausgeführte Instruktion textuell anzeigen, jedoch nur ein oder zwei Simulationsschritte überdauern.

In der Animation des Datenpfad-Editors sind Bewegungen entlang von Wegpunkten wichtig, in diesem Fall Linien, die Verbindungen repräsentieren. Auch in der Verkehrssimulation wird erst durch eine korrekte Bewegung der Autos eine anspruchsvolle Animation erreicht. Dies wird erschwert, da die Bewegung eines

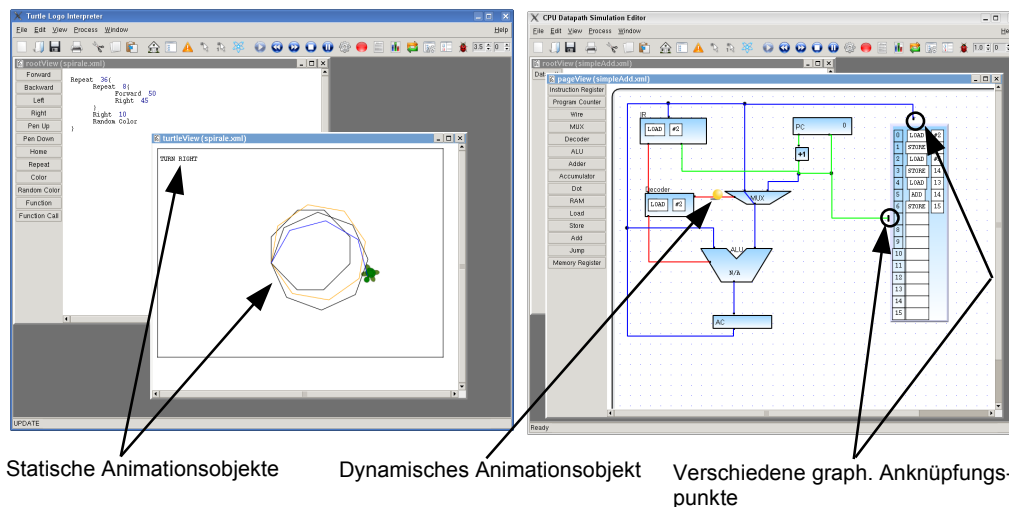




**Abbildung 3.4:** Beispiele visueller Sprachen der *dritten Kategorie*, die Gebrauch von Strukturkopplung machen. Die anwendungsspezifischen Sichten (Ampel- bzw. Dining-Philosophers-View) sind an das Petri-Netz gekoppelt. Änderungen in der Basisstruktur leiten Animationen in der gekoppelten Struktur ein.

Autos in jedem Simulationsschritt in so genannte *Mikroschritte* unterteilt wird. Dies ist notwendig um ein realistisches Verhalten der Autos zu simulieren, das dem Nagel-Schreckenberg Modell [20] nachempfunden ist. Innerhalb eines Mikroschrittes kann ein Auto nacheinander auf mehrere Straßenkacheln bewegt werden. Dies hat zur Folge, dass vor und nach einem Simulationsschritt nur Start- und Endposition eines Autos bekannt sind. Wurde das Auto während eines Simulationsschrittes über eine Kurve bewegt, so soll nicht der direkte Weg in der Animation benutzt werden, sondern das Auto soll sich realistisch am Straßenverlauf orientieren und sich auch entsprechend in Kurven zur Fahrtrichtung drehen. Hier zeigt sich, dass zwischen Simulation und Animation eine Informationslücke klafft. Besonders deutlich wird dies bei Petri-Netzen, wobei eine Marke in der Nachbedingung erstellt wird, aber zwei unterschiedliche Transitionen hätten schalten können. In der Simulation kann dies nicht ohne Änderungen (dem Zwischenspeichern der schaltenden Transition) ausgedrückt werden. Somit fehlen der Animation Informationen. Der Animationskomponente müssen also Mechanismen bereitgestellt werden um auch diese Fälle abzudecken, ohne dabei die Simulationsspezifikation ändern zu müssen.

In der Datenpfad-Animation fällt außerdem auf, dass der Ball, der die Datenimpulse repräsentiert, an unterschiedlichen Stellen eines Sprachkonstrukts andockt. Dadurch, dass DEViL syntax-basierte Structureditoren generiert, ist die Ausdehnung eines Sprachkonstrukts nur zur Laufzeit zu ermitteln. Um trotzdem komfortabel Positionen innerhalb eines Sprachkonstrukts angeben zu können (z. B. der out1-Ausgang des RAM), wurde das Konzept der *graphischen Anknüpfungspunkte* mittels so genannter *Crosslines* entwickelt. In Abbildung 3.5 sind im Datenpfad-Editor

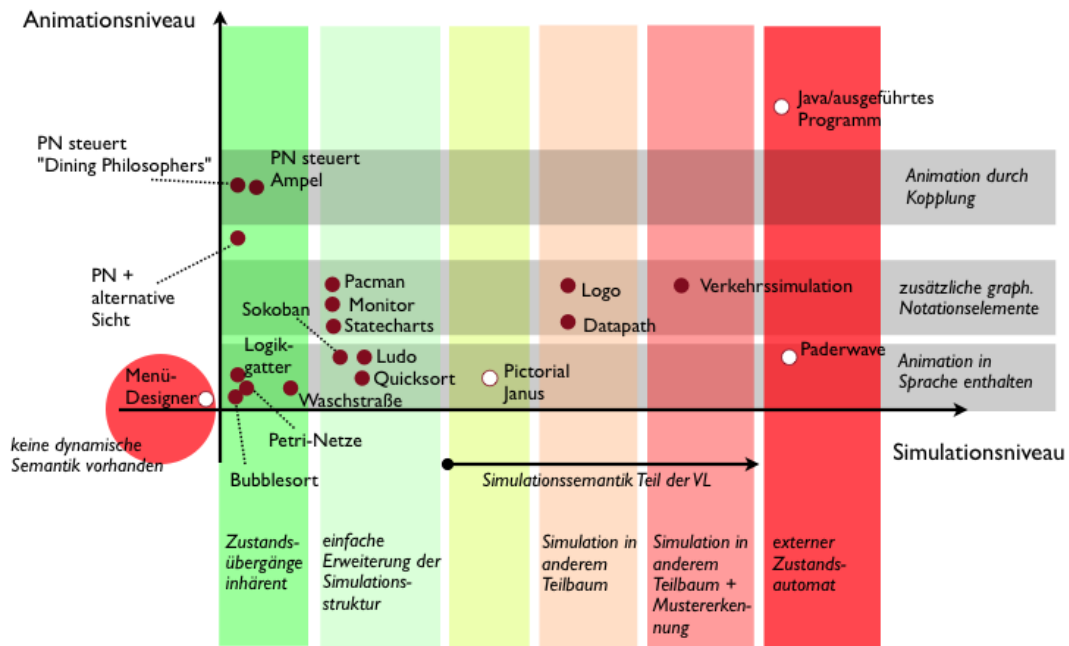


**Abbildung 3.5:** Beispiele visueller Sprachen der *vierten Kategorie*. Links im Bild der Logo-Editor, rechts der Datenpfad-Editor. Zu sehen ist der Unterschied zwischen statischen und dynamischen Animationsobjekten. Statische überdauern Simulationsschritte und werden nicht bewegt. Dynamische Animationsobjekte werden nur innerhalb einer Animationssequenz benutzt.

zwei solcher Anknüpfungspunkte zu sehen. Im Editor für generische Zeichnungen können bestimmte benannte Stellen eines Sprachkonstrukts markiert werden, die dann zur Laufzeit in absolute Positionen umgerechnet werden und referenziert werden können.

In die *fünfte Kategorie* fallen Sprachen, wie z. B. der DEViL-Menü-Designer. Mit ihm lassen sich visuell Menüstrukturen entwerfen, wie sie sich auch später in von DEViL generierten Editoren finden. Programm- und Programmausführung haben dieselbe Darstellung. Das andere Ende des Spektrums von Sprachen dieser Kategorie bilden VLs wie PaderWAVE [105]. Mit PaderWAVE lassen sich dynamische Webseiten visuell modellieren. Dabei werden neben den Verlinkungen und der visuellen Definition statischer Webseiten auch Skripte und Datenbankabfragen modelliert. Aus einer visuellen Spezifikation werden PHP-Skripte und HTML-Code generiert, die zusammen auf einem Webserver ausgeführt werden können. PaderWAVE als VL definiert selbst keine Ausführungsemantik innerhalb der Sprache. Der generierte Code, der auf dem Webserver ausgeführt wird, erhält seine Daten durch den Benutzer, der Aktionen auf der Webseite ausführt. Die Ausführung ist durch quasi beliebige PHP-Skripte sehr komplex und auch nicht in der Sprache definiert. Eine simulierte Variante von PaderWAVE könnte also höchstens darin bestehen, Callback-Methoden in den generierten PHP-Code zu integrieren, die ausgeführt werden, wenn die Webseite besucht wird und bestimmte Animationsaktionen im

### 3.4. KLASSIFIKATION VISUELLER SPRACHEN



**Abbildung 3.6:** Klassifikation visueller Sprachen und Einordnung der Beispielimplementierungen.

generierten Editor triggern. Diesen indirekten Ansatz verfolgt MetaEdit+ (vgl. Abschnitt 2.7.1 auf Seite 41). Er soll hier aber nicht Teil der Betrachtung sein. Eine simulierte PaderWAVE-Variante erscheint im Übrigen aber auch nicht besonders sinnvoll.

Abbildung 3.6 zeigt einen Überblick auf die Sprachklassen und eine Einordnung der Beispielimplementierungen. Die X-Achse repräsentiert das Simulationsniveau. Es unterscheidet sich von Stufe zu Stufe vom semantischen Modell der Sprache. Ab Stufe drei ist die Simulationssemantik Teil der visuellen Sprache. Hier ist ein deutlicher Komplexitätssprung in der Realisierung zu verzeichnen. Sprachen mit externem Zustandsautomaten oder Sprachen ohne dynamische Semantik (links der Y-Achse) sind nicht simulierbar.

Auf der Y-Achse ist das Animationsniveau abgebildet. Es gibt an, wie weit sich Animations- von Simulationsstruktur unterscheiden. Hier ist anzumerken, dass ein höheres Animationsniveau nicht zwingend mit einer schwierigeren Spezifikation einhergeht. Sprachen auf der höchsten Ebene zeichnen sich hier durch Strukturkopplung aus. Hieraus lassen sich wiederum automatisch Animationen ableiten. Dies ist recht einfach in DEViL zu realisieren. Innerhalb einer Sprachklasse gibt es keine Rangordnung zwischen den einzelnen Sprachen. Der Schwierigkeitsgrad ist hier nur schwer abzuschätzen. Ich habe dennoch versucht dem Spezifikationsaufwand der einzelnen Sprachen gerecht zu werden. So ergibt sich beispielsweise für die

Pac-man-Implementierung ein erhöhter Spezifikationsaufwand für die Animation als für die der Statecharts, da noch dynamische Animationsobjekte eingesetzt werden. Trotzdem lässt sich auch dies recht einfach realisieren, wie die Evaluation zeigen wird.

# 4 Simulationskonzept

## Inhalt

---

<b>4.1</b>	<b>Der Simulator</b> . . . . .	<b>88</b>
4.1.1	Der erweiterte Spezifikationsprozess . . . . .	88
4.1.2	Anforderungen an den generierten Simulator . . . . .	90
4.1.3	Einbettung des Simulators in die Werkzeugkette . . . . .	92
4.1.4	Zeitliches Verhalten des Simulators . . . . .	93
<b>4.2</b>	<b>Entwurf der Simulationssprache</b> . . . . .	<b>96</b>
4.2.1	Imperativ vs. regelbasiert . . . . .	99
<b>4.3</b>	<b>Die Simulationsspezifikationssprache DSIM</b> . . . . .	<b>99</b>
4.3.1	Spezifikation der Simulationsstruktur . . . . .	100
4.3.2	Spezifikation des Simulationsverhaltens . . . . .	102
4.3.3	Spezifikation von Ereignissen . . . . .	104
4.3.4	Spezielle Funktionalitäten . . . . .	106
4.3.5	Textuell vs. visuell . . . . .	110
<b>4.4</b>	<b>Verwandte Arbeiten</b> . . . . .	<b>111</b>

---

In diesem Kapitel soll das hier entworfene Simulationskonzept vorgestellt werden. Es bildet die Grundlage für die Animation. Dazu beschreibe ich zunächst den aus einer Spezifikation generierten Simulator; die Anforderungen, die an den Simulator gestellt werden und dessen Einbettung in das Gesamtsystem. Im Anschluss werden Anforderungen zur entworfenen Simulationsspezifikationsprache *DSIM* diskutiert und diese dann an ausgewählten Beispielen vorgestellt.

### 4.1 Der Simulator

Im Rahmen dieser Arbeit wird die Simulation als Hilfsmittel zur Animation visueller Sprachen betrachtet. Die Begriffe *Simulator* bzw. *Simulation* habe ich gewählt, da die Ausführung visueller Sprachen analysiert werden soll. Durch das gewählte *ereignisbasierte* Simulationskonzept lassen sich sehr einfach Sprachen entwerfen und untersuchen, die auf stochastischen Modellen basieren. Das Konzept der Ereignisse und der Ereigniswarteschlange wird in der Literatur immer in Zusammenhang mit Simulatoren gebracht (vgl. Abschnitt 2.5). Generatorsysteme, wie GenGed oder DiaMeta, führen ebenfalls den Begriff der Simulation in Zusammenhang mit der Ausführung von VLs ein. Den Simulator betrachte ich in meiner Arbeit als einen Automat, der den Simulationsablauf koordiniert. Er arbeitet dabei nicht auf einer Zwischensprache, wie z.B. eine *virtuelle Maschine*, sondern modifiziert direkt eine Instanz der Simulationsstruktur.

Der entworfene Simulator soll nicht als Simulator im herkömmlichen Sinne betrachtet werden, der in der Regel für Fertigungssysteme oder rechenintensive Simulationen gedacht ist. Deshalb soll er auch nicht mit einem existierenden Simulator direkt verglichen werden, da die Ansätze verschieden sind.

Es sollen die in Kapitel 3 untersuchten Sprachen der Kategorien eins bis vier der Klassifikation simuliert werden. Neben der reinen Simulation wird der Simulator eine Analysekomponente enthalten, sodass eine Auswertung auch von komplexen Simulationen unterstützt wird. Weitere Eigenschaften werden in den folgenden Abschnitten diskutiert.

Zunächst soll auf den nun erweiterten allgemeinen Spezifikationsprozess für Struktureditoren in DEViL eingegangen werden.

#### 4.1.1 Der erweiterte Spezifikationsprozess

Die Simulations- und Animationsspezifikation erweitert das allgemeine DEViL Spezifikationskonzept um einen weiteren Zweig, wie in Abbildung 4.1 zu sehen ist. Die Simulationskomponente wird dabei nur optional in die generierten Umgebungen integriert - und auch nur, falls eine Simulationsspezifikation vorhanden ist. Somit kann

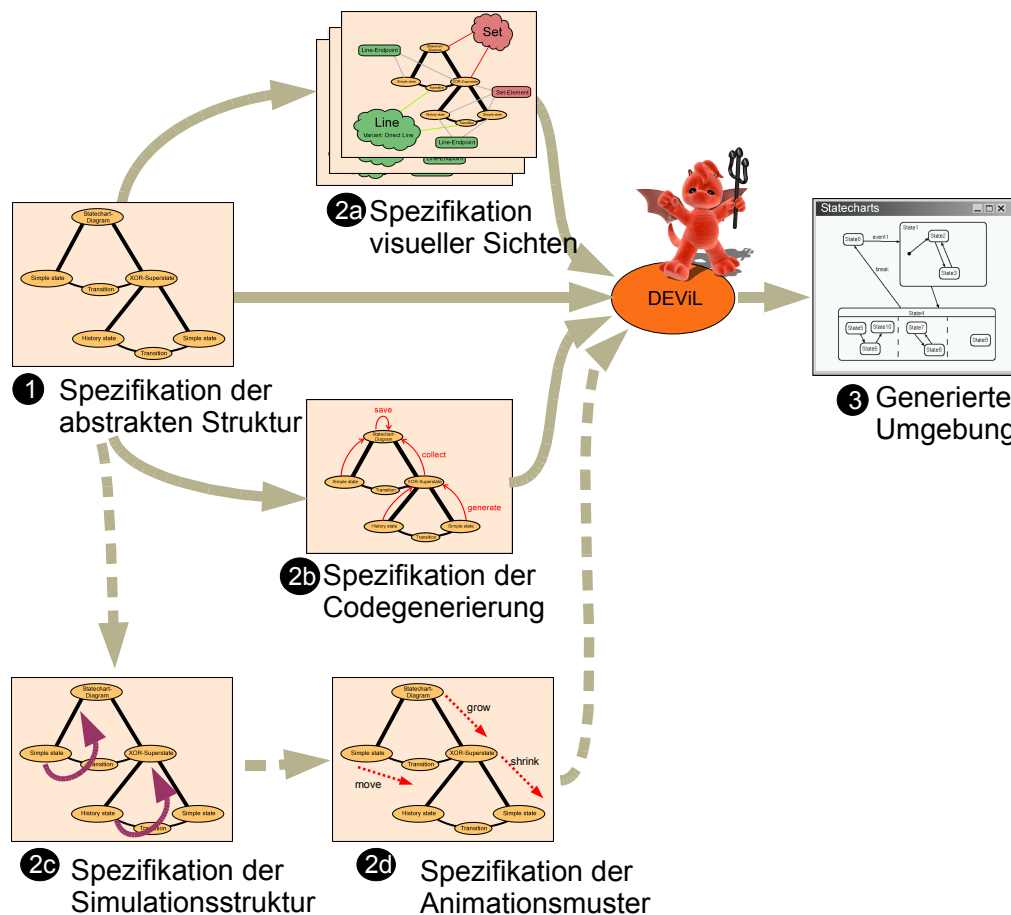


Abbildung 4.1: Erweiterung des DEViL Spezifikationsprozesses

erreicht werden, dass bereits vorhandene Spezifikationen ohne Simulationsunterstützung nicht durch Seiteneffekte beschädigt werden. Ebenso ist zu sehen, dass eine Simulationsspezifikation nicht auf vorhandenen Spezifikationen aufbaut. So ist es auch denkbar, ausschließlich einen Structureditor mit Simulationsunterstützung zu generieren und erst später die graphische Repräsentation hinzuzufügen. Somit können Simulationsspezifikationen auf DSSL aufbauen, müssen dies aber nicht. Wie später noch zu sehen ist, wird aus einer Simulationsspezifikation automatisch eine Animationsstruktur generiert, die standardisierte Animationen aus Grundoperationen der Simulation generiert. Diese Animationen können vom Spezifizierer noch flexibel angepasst werden (Schritt 2d).

In der generierten Umgebung wird durch eine Simulationsspezifikation ein ereignisbasierter Simulator integriert, der auf einer eigenen Simulationsstruktur arbeitet. Die Wahl des ereignisbasierten Konzeptes ist aufgrund der Verbreitung bei den Simula-

tionssprachen und der Flexibilität der ereignisorientierten Simulation gefallen, wie sich bei der Untersuchung anderer Werkzeuge herausgestellt hat (vgl. Abschnitt 2.5).

### 4.1.2 Anforderungen an den generierten Simulator

Bevor die Spezifikationsmethoden der Simulation für visuelle Sprachen erläutert werden, wird an dieser Stelle zunächst der Fokus auf den aus der Spezifikation generierten Simulator gelegt.

Vorweg sei erwähnt, dass der hier generierte Simulator auf die Anforderungen visueller Sprachen maßgeschneidert ist. Er soll nicht in Konkurrenz zu Simulatoren stehen, die komplexe wissenschaftliche Aufgaben lösen, wie z. B. die Simulation des Klimas und dazu Aufgaben auf einen Rechnercluster verteilen. Vielmehr soll der generierte Simulator sich nahtlos in das DEViL-System einpassen. Trotzdem soll er korrekte Ergebnisse liefern und Mechanismen anbieten, die auch in etablierten Simulatoren bereitgestellt werden. Dazu gehört unter anderem die Generierung von Zufallszahlen bestimmter Wahrscheinlichkeitsverteilungen.<sup>1</sup>

Die konkreten Anforderungen an den Simulator sind:

1. Arbeit auf einer eigenen maßgeschneiderten Simulationsstruktur.
2. Entkopplung vom semantischen Modell der visuellen Sprache.
3. Abbildung des vollständigen semantischen Modells auf eine geeignete Simulationsstruktur.
4. Realisierung von langlaufenden Simulationen.
5. Spezielle Anforderungen: Zufallszahlen, Pfadausdrücke, Mustererkennung und kontext-sensitive Funktionen.
6. Anpassung an den LIGA Attributauswertergenerator.
7. Analysefähigkeiten.
8. Erweiterbarkeit.
9. Schmale Schnittstelle zur Animation durch standardisierte Änderungsoperationen.
10. Flexibilität bei der Bedienung.

Die Arbeit auf einer eigenen Simulationsstruktur verdeutlicht eines der wichtigsten Ziele bei der Gestaltung von Modellen für die Simulation. Die Simulationsstruktur muss in ihrer Komplexität möglichst minimal sein, um keine unnötigen oder langwierigen Berechnungen zu provozieren, die nicht der Problemlösung dienen. Trotzdem müssen geeignete Schnittstellen enthalten sein, die es erlauben, die Simulationsstruktur zu attributieren um neue Eigenschaften hinzufügen zu können. Des

---

<sup>1</sup>Intern werden die Zufallszahlengeneratoren der etablierten Boost C++ Bibliothek verwendet [26].



Weiteren muss es die Simulationsstruktur ermöglichen, einfach auf Strukturobjekte mit bestimmten Eigenschaften zugreifen zu können. Außerdem muss es möglich sein, Strukturobjekte im Baum zu verschieben. Dies geschieht im Hinblick auf transportorientierte Simulationen, bei denen Simulationsobjekte im Modell "wandern".

Die vollständige Entkopplung des Simulators und der Simulationsstruktur vom semantischen Modell der Sprache bedeutet, dass bestehende Spezifikationen nicht beeinflusst werden. Dies stellt die besondere Bedeutung eines Simulators im Kontext eines Generatorframeworks für VLs heraus. Dies ist wichtig, da mit DEViL bereits viele DSLs spezifiziert wurden. Des Weiteren erlaubt es die Entkopplung, den Simulator leichter wart- und erweiterbar zu machen. So ist eine spätere Erweiterung denkbar, um Rechenlast auf mehrere Rechner zu verteilen. Dazu müssten lediglich im Simulator entsprechende Erweiterungsschnittstellen geschaffen werden. Ein weiterer Aspekt der Entkopplung wird sich in der Sprache zeigen: die Verwendung von Zugriffsfunktionen auf Typen von Objekten. Dadurch lässt sich später sehr einfach die visuelle Darstellung anpassen ohne die Simulationsspezifikation modifizieren zu müssen. Des Weiteren erlaubt die Separierung von semantischer Struktur und Simulationsstruktur, beide Modelle auch in so genannten Erstellungs- und Initialisierungsfunktionen zu unterscheiden. Dies sind Funktionen, die automatisch aufgerufen werden, wenn ein neues Sprachkonstrukt erstellt wird oder an einen anderen Ort verschoben wird. Beide Strukturen können somit unabhängig voneinander modifiziert werden.

Die Abbildbarkeit von semantischer und Simulationsstruktur besagt, dass alle Konstrukte der semantischen Struktur der VL auch im Simulator abgebildet und manipuliert werden können.

Zufallsbasierte Simulationen liefern häufig erst sehr spät in der Simulation Ergebnisse oder es sind, um ein korrektes Resultat zu erzielen, mehrere Simulationsläufe nötig. Ein Mechanismus muss vorhanden sein, um die Animation von Simulationsschritten zu unterbinden. Auch können die Editoren so durch ein Skript automatisiert gestartet werden.

Besondere Konstrukte wie Pfadausdrücke, Mustererkennung oder Zufallszahlen sollen ebenfalls spezifiziert werden können. Diese Anforderung wurde direkt aus den untersuchten Simulationssprachen abgeleitet.

Die Anpassung an LIGA ist wichtig zur späteren Berechnung der dynamischen graphischen Darstellung durch den Simulator. DEViL benutzt zur Berechnung der Animation den Attributauswertergenerator LIGA [53] des Eli-Systems. Die Simulationsstruktur wird dabei so transformiert, dass sie von LIGA durchlaufen werden kann. Dies ist jedoch transparent für den Benutzer und schränkt ihn in keiner Weise bei der Spezifikation ein. Hier zeigt sich der Nutzen der Designentscheidung, Vorhandenes und bereits Bewährtes aus dem DEViL-System zu adaptieren und

wiederzuverwenden.

Neben der Animation als Analysewerkzeug kann es auch noch andere Mechanismen geben, um visuelle Programme zu analysieren. Dies kann die schrittweise Ausführung von Simulationen, das Loggen von Ereignissen oder auch die Verifikation von Eigenschaften der Struktur sein (vgl. Kapitel 6).

Erweiterbarkeit ist insbesondere wichtig für die Analyse. Sprachentwickler sollen den Simulator durch eigene oder fremde Bibliotheken erweitern können. Dies geschieht bereits durch den Einsatz der C++ Boost-Bibliotheken um Zufallszahlen zu erzeugen. Aber auch eigene Funktionen, die nicht so allgemein sind, dass sie in das Generatorframework einfließen können, sollen sich einbinden lassen.

Die schmale Schnittstelle verweist bereits auf die Animation. Durch einige wenige standardisierte Operationen, die die Simulationsstruktur ändern, kann eine Animation abgeleitet werden. Somit kann die Animation als formale Abbildung der Simulation betrachtet werden.

Der letzte Punkt zielt auf die Bedienbarkeit und die Flexibilität des Simulators ab. Es soll möglich sein Simulationsschritte einzeln ablaufen zu lassen, die Simulation zu stoppen und mit veränderten Eingaben wieder zu starten. In der Literatur wird dies oft auch als *Reaktivität* bezeichnet. Auch eine Rückwärtssimulation mit Animation im Sinne des Forms/3 Systems [16] sollte möglich sein.

### 4.1.3 Einbettung des Simulators in die Werkzeugkette

Der Simulator kann als eine Umgebung verstanden werden, die aus einer Reihe von Werkzeugen besteht (siehe Abb. 4.2). Es wird ein zeitdiskreter ereignisbasierter Simulator generiert, der dem "next-event to time advance"-Ansatz folgt. Das Zeitmodell unterteilt die Simulation in ihre kleinsten Einheiten und kann vom Benutzer des Simulators flexibel angewendet werden. Neben einer Warteschlange, in die Ereignisse eingeordnet werden können, gibt es noch die Option Ereignisse direkt auszuführen. Dies vermeidet race-conditions, da sich Änderungen in der Struktur direkt auswirken.

Der Simulator ist das Kernstück der generierten Simulationsumgebung. Er simuliert auf der Simulationsstrukturinstanz. Diese wird aus dem semantischen Modell der Sprache hergeleitet. D. h. es können alle Konstrukte des semantischen Modells in die Simulationsstruktur übernommen werden. Man kann jedoch auch Teilbäume, die nicht für die Simulation von Belang sind, weggelassen. Dies kann für Teilbäume gelten, die nur der Darstellung dienen. Die so entstandene Simulationsstruktur kann dann noch um weitere optionale persistente Attribute, Zufallszahlen, Pfadausdrücke oder kontext-sensitive Funktionen erweitert werden. Pfadausdrücke können so zur

Laufzeit Gruppen von Objekten aufsammeln, die bestimmte Eigenschaften erfüllen, z. B. die Vorbedingung einer Transition in einer Petri-Netz-Simulation.

Der Simulator überprüft während der Simulation in einer Endlosschleife die Simulationsstruktur und plant Ereignisse ein, die die Simulationsstrukturinstanz ändern (explizite Darstellung ist in der Abbildung 4.2 weggelassen). Änderungen an der Simulationsstruktur, die sich auf die semantische Struktur der VL zurückführen lassen, werden ebenfalls vom Simulator geschrieben. Außerdem schreibt der Simulator in die Analysekomponente und protokolliert die Veränderungen in einer Historie. Simulationsschritte können so vollständig rückgängig gemacht werden.

Eine weitere Eigenschaft des Simulators ist, dass die Instanz einer Simulationsstruktur während einer schrittweisen Simulation durch den Anwender modifiziert werden kann. Hier ist jedoch Vorsicht geboten, da auch das Löschen von Programmkonstrukten erlaubt ist und so auch ungültige Simulationszustände produziert werden können. Trotzdem wird hier eine optimistische Variante benutzt, die den Eingriff des Anwenders in die Simulationsstrukturinstanz jederzeit erlaubt. So wird eine besonders hohe Flexibilität erreicht. Eine Filterung von potentiell gefährlichen Modifikationen könnte eine sinnvolle Erweiterung sein. In der Literatur wird diese Eigenschaft als eine *“in-place”*-Änderung bezeichnet [88].

Der so genannte *Change-Propagation-Handler* registriert Änderungen an der Simulationsstruktur und stößt ggf. Sicht-Updates an. Dies kann bei jedem Simulationsschritt passieren, was eine Animation auslösen würde oder, wenn im *Simulation-only* Modus gearbeitet wird, auch erst nach beispielsweise 1000 Simulationsschritten. Weiterhin wichtig für Sprachen, bei denen sich Simulations- und Animationsstruktur stark unterscheiden und eine Spezifikation durch Kopplung realisiert ist, ist der Zusammenhang zwischen editierbarer Struktur und Simulator, verdeutlicht durch den gestrichelten Pfeil. Änderungen an der semantischen Struktur können nämlich Änderungen in gekoppelten editierbaren Strukturen auslösen. Der Simulator erkennt dies und schreibt für die gekoppelten Strukturen ebenfalls Modifikationsoperationen in die Schnittstelle zur Animation. Somit kann auch hier einfach eine Animation hergeleitet werden.

### 4.1.4 Zeitliches Verhalten des Simulators

Das Zeitmodell kann durch den Spezifizierer einer Simulation flexibel angepasst werden. Da ich den ereignisbasierten Ansatz zur Simulation gewählt habe, existiert eine Ereigniswarteschlange, in die Ereignisse zu beliebigen Zeitpunkten eingeordnet werden können. Zu einem Simulationszeitpunkt können beliebig viele Ereignisse ausgeführt werden, die wiederum aus beliebig vielen Änderungsoperationen bestehen. Eine Ordnung über Ereignisse, die zum selben Zeitpunkt ausgeführt werden, existiert nicht. Eine Notwendigkeit für solche Ordnungen hat sich bei den bisher

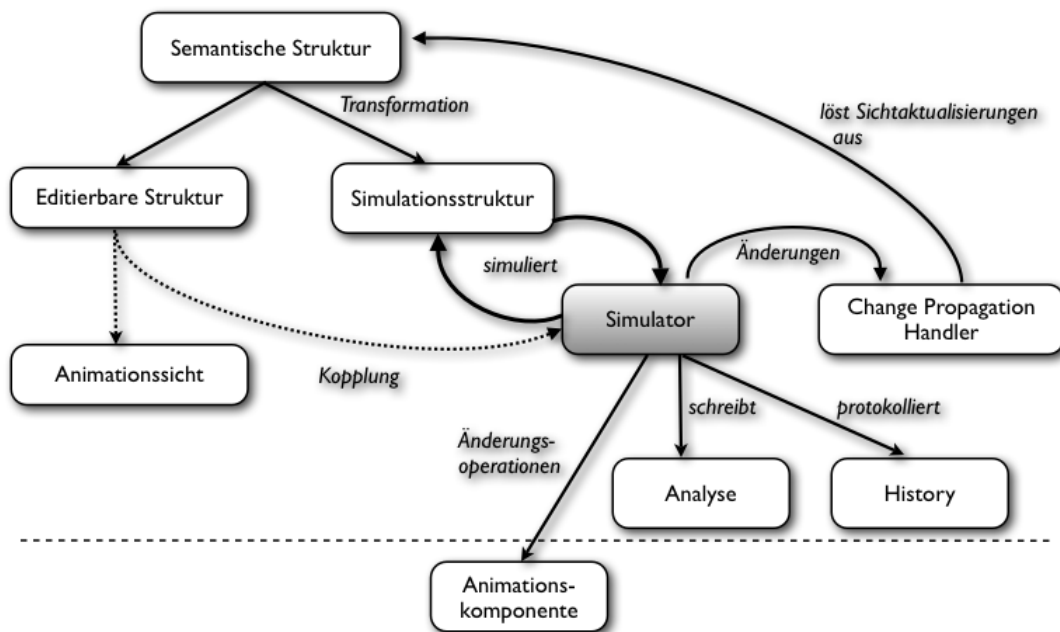


Abbildung 4.2: Einbettung des Simulators in das Simulationsframework

implementierten Sprachen noch nicht gezeigt. Dies könnte jedoch leicht nachgerüstet werden. Zeitpunkte zu denen kein Ereignis eingeplant ist, werden übersprungen. Dies ist der *“next-event to time advance”*-Ansatz. Um anzudeuten, dass zu einem Zeitpunkt kein Ereignis getriggert wurde, kann im generierten Editor eine Stoppuhr angezeigt werden. Besonders in Warteschlangensimulationen kann der Anwender Inaktivität so sehr leicht erkennen.

Neben der Warteschlange für Ereignisse existiert noch die weitere Möglichkeit Ereignisse direkt auszuführen: Die Ereigniswarteschlange wird dabei umgangen. Mit diesem Mechanismus können race-conditions verhindert werden. Diese können in ereignisbasierten Systemen eintreten, wenn mehrere Ereignisse zu unterschiedlichen Zeitpunkten auf dieselben Daten zugreifen. Dies kommt beispielsweise in Petri-Netzen vor, wenn zwei Transitionen dieselbe Stelle als Vorbedingung haben. Ist dort eine Marke vorhanden, so stellt der Simulator in der Simulationsschleife fest, dass beide Transitionen schalten können, obwohl dies laut Petri-Netz Semantik nur für eine Transition geschehen darf. Wird die Marke direkt durch ein Ereignis entfernt, so erkennt der Simulator dies und plant kein weiteres Ereignis mehr ein.

Abbildung 4.3 zeigt exemplarisch das zeitliche Verhalten des Simulators. Prinzipiell lässt sich ein Simulationsschritt dabei in zwei Phasen unterteilen. In der ersten Phase (lila farbene Blöcke auf der X-Achse) arbeitet der Simulator die *Simulatorschleife* ab. In ereignisbasierten Simulatoren wird dabei das Simulationsmodell überprüft und Ereignisse werden in die Warteschlange eingeordnet. In einer zweiten Phase

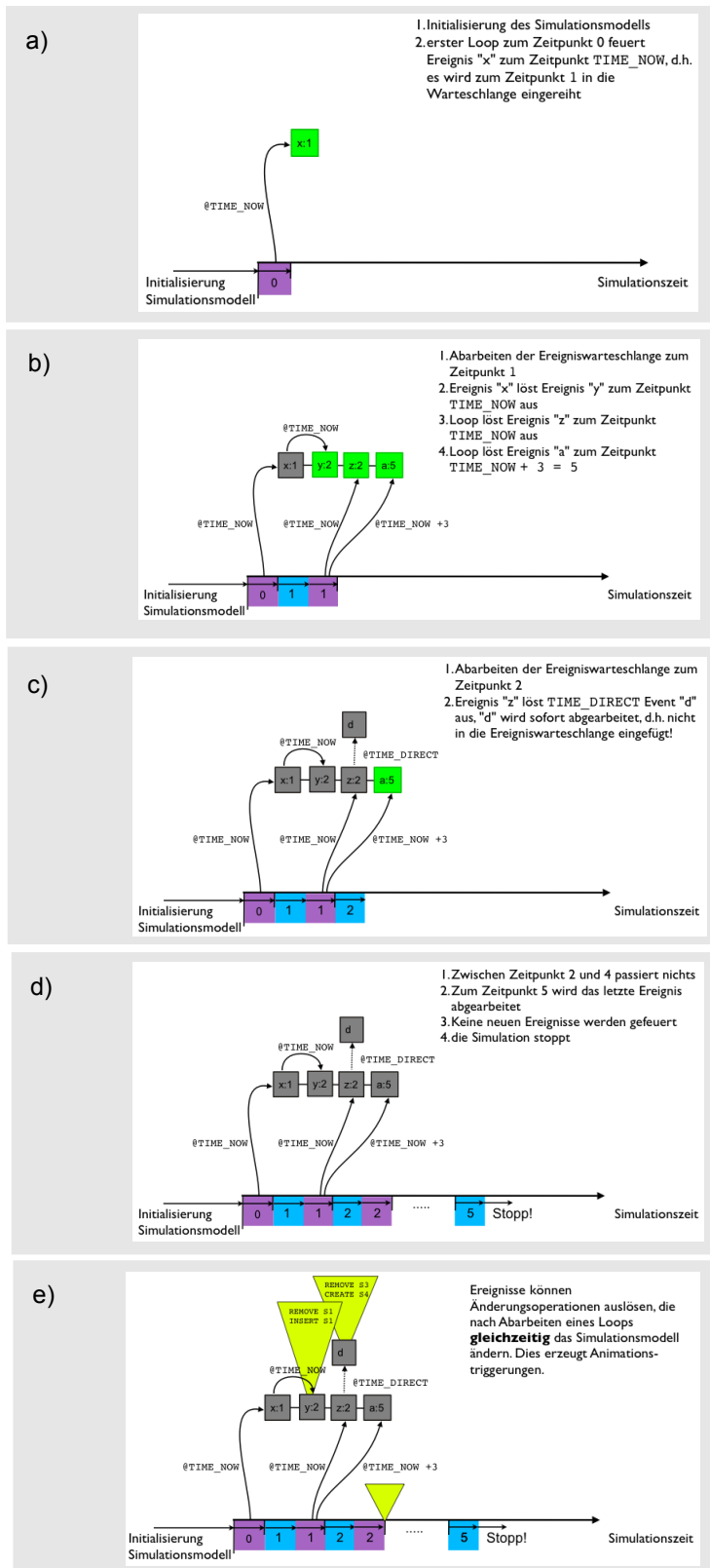


Abbildung 4.3: Zeitliches Verhalten des Simulators

werden die Ereignisse zum selben Simulationszeitpunkt abgearbeitet.

Abb. 4.3a zeigt, dass im ersten Schleifendurchlauf ein Ereignis  $x$  zum Zeitpunkt 1 in die Warteschlange eingeordnet wird (grünes Kästchen).

Abb. 4.3b zeigt die Warteschlange zum Zeitpunkt 1. Das Ereignis  $x$  (graues Kästchen) ist abgearbeitet, hat jedoch ein weiteres Ereignis  $y$  in die Warteschlange eingeordnet. In der Simulatorschleife zum Zeitpunkt 1 wurden die Ereignisse  $z$  und  $a$  eingefügt. Die Zeitpunkte der Ausführung sind 2 bzw. 5.

Abb. 4.3c zeigt die Warteschlange nach Abarbeiten der Ereignisse zum Zeitpunkt 2. Das Ereignis  $z$  wurde bearbeitet und hat ein weiteres Ereignis  $d$  ausgelöst, das jedoch nicht in die Warteschlange eingeordnet, sondern direkt bearbeitet wurde. Das momentan einzig unbearbeitete Ereignis ist  $a$ . Zwischen Zeitpunkt 2 und 5 passiert weiter nichts; die Simulation würde hier zur Verdeutlichung die Stoppuhr anzeigen. Zum Zeitpunkt 5 wird  $a$  ausgeführt. Da Ereignis  $a$  keine weiteren Ereignisse feuert und die Warteschlange leer ist, stoppt die Simulation.

Abb. 4.3e zeigt, dass jedes Ereignis beliebig viele Simulationsmodifikationsoperationen auslösen kann, deren Animationen jeweils nach Abarbeiten einer Simulationsschleife erfolgt.

## 4.2 Entwurf der Simulationssprache

Der Entwurf einer Simulation auf Basis einer visuellen Sprache zeigt, dass die semantische Struktur der visuellen Sprache nicht unbedingt optimal zur Simulationsstruktur passt. Das ist der Fall, wenn sie Klassen, Attribute oder Referenzen enthält, die als Zugeständnis an die editierbare Struktur, also der visuellen Darstellung der visuellen Sprache hinzugefügt wurden. Es werden aber bestimmte Objekte der visuellen Sprache in der Simulationsstruktur benötigt. Häufig muss die Simulationsstruktur auch neue Funktionalitäten wie das Gruppieren von Objekten oder das Ausführen von Funktionen bereitstellen.

Die Simulationsstruktur einer visuellen Sprache kann also Gebrauch vom semantischen Modell der visuellen Sprache machen, d. h. in der Simulationsstruktur kann auf Objekte des semantischen Modells zugegriffen werden. Diese Simulationsobjekte haben gegenüber Objekten des semantischen Modells der visuellen Sprache einige zusätzliche Eigenschaften. Bei der Initialisierung sind sie genaue Kopien ihrer Pendanten der visuellen Sprache, werden in der Simulationsstruktur jedoch erweitert oder beschnitten.

Der erwähnte Zugriff auf Objekte der visuellen Sprache erfolgt durch die bereits

eingeführten Pfadausdrücke. Diese erlauben es, einfach durch den Strukturbaum zu wandern und Objekte, die bestimmte Eigenschaften erfüllen, aufzusammeln (z. B. die Vorbedingung einer Transition eines Petri-Netzes).

Folgende Eigenschaften von Strukturspezifikation und Simulationsstruktur sind zu nennen:

1. Ähnlichkeit der Spezifikationsprache zu DSSL und anderen Spezifikationsmethoden in DEViL.
2. Erweiterung der Struktur durch weitere (so genannte erweiterte) Attribute.
3. Erweiterung der Struktur durch kontext-sensitive Funktionen.
4. Spezialkonstrukte wie Zufallszahlen.
5. Zugriff auf Mengen von Objekten.

Die Ähnlichkeit zu vorhandenen Spezifikationsmethoden ist eine der wichtigsten Beiträge dieser Arbeit. Die Ähnlichkeit zu existierenden und bewährten Methoden ermöglicht es, Simulation und Animation von visuellen Sprachen als einfache Erweiterung des bestehenden Konzepts aufzufassen. Simulationsspezifikationen können leicht gelesen und verstanden werden. Durch die Wiederverwendung müssen nicht neue Spezifikationstechniken erlernt werden.

Die Struktur Erweiterung ist wichtig, wenn Simulationen spezifiziert werden sollen, die strukturelle Eigenschaften besitzen, die im semantischen Modell der Sprache nicht vorhanden sind. Außerdem erleichtert dies die Teamarbeit und die Trennung von Simulation und Strukturdefinition in DSSL. Ein Simulationsspezifizierer kann eine Simulation entwerfen, ohne Rücksicht auf die zu Grunde liegende semantische Modellstruktur zu nehmen. Bestehende Klassen des semantischen Modells können erweitert werden, z. B. auch durch neue Teilbäume. Dies ist nützlich, wenn die Simulation einige zusätzliche Attribute benötigt um Simulationseigenschaften graphisch zu repräsentieren. In der Statechartssimulation ist dies beispielsweise das Attribut, das die Aktivität eines Zustands angibt. Aber auch neue Unterstrukturen sind denkbar, um beispielsweise Strategien für Routen von Autos in einer Verkehrssimulation zu beschreiben und zu speichern.

In objektorientierten Simulationen kommt es häufig vor, dass Simulationsobjekte durch die Simulationsstruktur "wandern" und an bestimmten Stationen modifiziert, z. B. verzögert werden. Das Simulationsobjekt weiß selbst, was an jeder Station passiert. Es kennt also seinen Kontext und führt entsprechende Funktionen aus. Die Erweiterung durch neue Objekte oder auch Spezialisierung ist deshalb sehr einfach.

Die Generierung und Benutzung von Zufallszahlen ist ein wichtiger Aspekt in Simulationen. Nur so lassen sich aussagekräftige Ergebnisse erzielen. Mit Hilfe der Zufallszahlen, erweiterter Attribute und der Auslastungsanalyse können sehr

einfach Warteschlangensysteme spezifiziert werden.

Der letzte Punkt besagt, dass der Zugriff auf bestimmte Mengen von Objekten ermöglicht werden soll. Dies wird, wie ich später aufzeigen werde, durch Quantoren, die auf Objekte eines Typs zugreifen, realisiert. Somit ist es leicht möglich, die Struktur der Sprache später anzupassen, z. B. um sie zu verschieben oder tiefer zu schachteln, ohne die Simulationsspezifikation zu ändern.

Insgesamt ergibt sich nun für die Simulationsstruktur das Tripel  $\mathcal{S} = (\mathcal{C}_{\text{sim}}, \mathcal{A}_{\text{sim}}, \mathcal{M})$ , wobei  $\mathcal{C}_{\text{sim}}$  die Menge der Klassen des visuellen semantischen Sprachmodells ist und  $\mathcal{A}_{\text{sim}} = \mathcal{A}_{\text{semantic}} \cup \mathcal{A}_{\text{extended}}$  die Menge der Attribute des semantischen Modells vereinigt mit der Menge der erweiterten Attribute der Simulationsstruktur ist.

$\mathcal{M} : \mathcal{C}_{\text{sim}} \times \mathcal{A}_{\text{sim}}$  ist eine Abbildung von Attributen auf Klassen.

Die Anforderungen an die Spezifikationssprache für die Simulation sind vielfältig: sie soll möglichst einfach sein, jedoch eine große Menge an Simulationen visueller Sprachen durch einfache Sprachkonstrukte abdecken. Der Sprachumfang der Spezifikation leitet sich dabei direkt aus den Sprachen ab, die simulierbar sind. Diese decken das Spektrum zufallsbasierter, transportorientierter und Simulationen mit in der Sprache verankerter Simulationssemantik ab. Ist die Ausdrucksfähigkeit der Simulationssprache an einer Stelle nicht ausreichend, so soll es möglich sein, die Simulationssprache durch vom Sprachanwender definierte Funktionen zu erweitern.

Die Schnittstelle zur Animation sollte eng und standardisiert werden. Nur so ist es möglich, die Animation als formale Abbildung der Simulation anzusehen und Standardanimationen aus einer Simulationsspezifikation zu generieren.

Ein weiterer wichtiger Aspekt beim Entwurf der Sprache war, dass die Fähigkeit zur Teamarbeit in DEViL erhalten bleibt. Die einzelnen Spezifikationsaspekte beim Entwurf einer Sprache sollten weiterhin getrennt bleiben, d. h. Simulation, Animation, Sprachmodell und Repräsentation müssen weiterhin separat spezifizierbar bleiben.

Auch der Zugriff auf Sprachkonstruktinstanzen ist einfach. Er kann auf Typenebene erfolgen. Visuelle Darstellungen können so einfach geändert werden ohne die Simulationsspezifikation zu beeinträchtigen.

Wiederverwendung ist ein weiteres Merkmal, durch das sich die Spezifikationssprache auszeichnet. Vorhandene Konzepte zur Strukturdefinition werden im Simulationsmodell wieder aufgegriffen und erweitert.



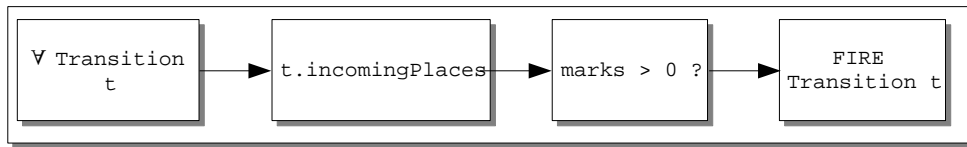


Abbildung 4.4: DSIM - funktional betrachtet

### 4.2.1 Imperativ vs. regelbasiert

Die Simulationsspezifikation folgt dem imperativen Programmierparadigma. Eine alternative Variante wäre das regelbasierte Programmierparadigma. Dagegen spricht jedoch, dass schon einfache Simulationen viele Regeln haben können. So werden in GenGed für die Simulation einer konkreten Ausprägung eines Petri-Netzes bereits vier Regeln zur Simulation und nochmal sechs Regeln für das Animationsmapping benötigt [32]. Regeln werden aufgrund von Seiteneffekten schnell unwartbar und unübersichtlich [111]. Fishwick beschreibt in [36], dass deklarative Modelle bei kontinuierlichen Systemen sowie bei Systemen, die aus einer Menge von Objekten bestehen (wie in unserem Fall), nicht gut geeignet sind. Außerdem können in regelbasierten Szenarien Abhängigkeiten zwischen entfernten Objekten nur schwer modelliert werden. Trotzdem ist es denkbar, die Sprache um regelbasierte Konstrukte zu erweitern, um eine vereinfachte Spezifikation für gewisse Anwendungszwecke zu erreichen.

Des Weiteren orientiert sich die Sprache DSIM am prozessorientierten Simulationsparadigma, wobei Simulationsobjekte bestimmten Typs aufgesammelt werden (durch Pfadausdrücke) und dann durch das Simulationsmodell wandern. Kontext-sensitive Funktionen unterstützen dieses Paradigma. Währenddessen werden Eigenschaften der Objekte durch Modifikation ihrer Attribute geändert. Diese Modifikationen können auch funktional als Hintereinanderschaltung von Funktionen betrachtet werden, die bestimmte Mengen von Objekten einschränken. Abbildung 4.4 zeigt diese funktionale Betrachtungsweise am Beispiel von Petri-Netzen. Die Menge aller Transitionen wird benutzt um die Menge aller eingehenden Stellen zu berechnen. Die berechneten Stellen werden dann wiederum gefiltert (`marks > 0`).

## 4.3 Die Simulationsspezifikationssprache DSIM

In den folgenden Abschnitten soll die Simulationsspezifikationssprache *DSIM* vorgestellt werden. Anhand einiger Ausschnitte aus Spezifikationen für bereits implemen-

tierte Editoren werde ich die einzelnen Teile des Spezifikationsprozesses in DSIM verdeutlichen.

### 4.3.1 Spezifikation der Simulationsstruktur

Die Spezifikation von Simulation in DSIM unterteilt sich in mindestens drei Abschnitte (ein vierter ist zur Konfiguration von globalen Einstellungen optional). Zunächst wird die Simulationsstruktur beschrieben. Hier können Klassen des semantischen Sprachmodells mit zusätzlichen persistenten Attributen, Pfadausdrücken oder kontext-sensitiven Funktionen ausgestattet werden. Im einfachsten Fall fällt die Simulationsstruktur mit dem semantischen Modell der Sprache zusammen. In diesem Fall wäre die Spezifikation hier leer, wie das beim Editor für das bekannte *Game-of-Life*-Szenario ist.

```
1 MODEL {
2   CLASS Transition {
3     SET incomingPlaces OF Place: "THIS.IVALUE[Connection.to].PARENT.from.VALUE[Place]";
4     SET outgoingPlaces OF Place: "THIS.IVALUE[Connection.from].PARENT.to.VALUE[Place]";
5   }
6 }
```

**Listing 4.1:** Spezifikation der Simulationsstruktur für Petri-Netze in DSIM

Listing 4.1 zeigt die Spezifikation der Simulationsstruktur für einfache Petri-Netze in DSIM. Der Ausschnitt zeigt, wie die Klasse `Transition` des semantischen Sprachmodells um zwei Pfadausdrücke erweitert wird. Die Pfadausdrücke berechnen *zur Laufzeit* die Vor- bzw. Nachbedingung einer Transition. Das Schlüsselwort `THIS` innerhalb des Pfadausdrucks wird dabei zur Laufzeit durch das aktuelle Objekt ersetzt, in diesem Fall ein `Transition`-Objekt.

Listing 4.2 zeigt einen weiteren Ausschnitt aus einer Simulationsstrukturspezifikation. Der Ausschnitt ist der Verhaltensspezifikation des *“Mensch-ärgere-Dich-nicht”*-Editors entnommen. Das Besondere ist, dass hier Klassen nicht nur um Pfadausdrücke erweitert werden, sondern auch durch persistente Attribute (Zeile 5 u. 6). Es können die gleichen Attributtypen wie in DSSL verwendet werden. So können unter Verwendung von SUB-Knoten aggregierte Unterbäume spezifiziert werden, die nur in der Simulation existieren. Interessant ist dies, wenn beispielsweise komplexe Strategien in Spielen modelliert werden sollen. Eine Spielfigur könnte so eine komplexe Strategie aggregiert enthalten. Das Weglassen von Teilbäumen, die für die Simulation irrelevant sind, kann ebenfalls spezifiziert werden. Im Beispiel wird der Teilbaum `ruleDescription`, der eine textuelle Beschreibung für Regeln des Spiels enthält, weggelassen (Zeile 37).

```

1 ...
2
3 CLASS PlayerFigure {
4   OBJECT getNode OF GameNode: "THIS.PARENT.PARENT";
5   isOnTurn: VAL VLBoolean INIT "0";
6   error: VAL VLBoolean INIT "0";
7
8   FUNCTION moveToNextNode(PlayerTable player) {
9     ON SimpleNode CONTEXT {
10      PlayerFigure figure = THIS;
11      GameNode node = figure.getNode;
12      GameNode nextNode = node.next;
13      GoalNode goalNode = node.goalNode;
14      IF (NOTNULL(goalNode)) {
15        IF(goalNode.color == figure.color) {
16          nextNode = goalNode;
17        }
18      }
19      FIRE moveFigureTo(figure,nextNode) @ TIME_DIRECT;
20    }
21
22    ON StartNode CONTEXT {
23      PlayerFigure figure = THIS;
24      GameNode node = figure.getNode;
25      GameNode nextNode = node.next;
26      GoalNode goalNode = node.goalNode;
27      IF (NOTNULL(goalNode)) {
28        IF(goalNode.color == figure.color) {
29          nextNode = goalNode;
30        }
31      }
32      FIRE moveFigureTo(figure,nextNode) @ TIME_DIRECT;
33    }
34  }
35
36 CLASS Root {
37   IGNORE SUBTREE (ruleDescription);
38 }
39 ...

```

Listing 4.2: Ausschnitt der Spezifikation der Simulationsstruktur für Ludo in DSIM

Eine weitere Besonderheit ist die Definition von *kontext-sensitiven Funktionen* (Zeile 8 - 34). Eine solche Funktion kann als Objektmethode einer Klasse angesehen werden, die aus der Verhaltensspezifikation (siehe nächster Abschnitt) aufgerufen werden kann. Innerhalb dieser Funktion können Blöcke definiert werden, die den strukturellen Modellkontext einer Klasseninstanz referenzieren. Im Beispiel sind mit `StartNode` und `SimpleNode` zwei Strukturknoten benannt, d. h. der Rumpf des Blocks wird nur ausgeführt, wenn sich das aktuelle Objekt direkt unterhalb eines `SimpleNode` bzw. eines `StartNode` befindet. So kann das Verhalten von Objekten in bestimmten Kontexten sehr einfach spezifiziert werden. Dies ist bei transportorientierten Simulationen häufig der Fall. Je nach Station (Kontext) werden unterschiedliche Aktionen ausgeführt. Der Kontext kann auch noch genauer angegeben werden, indem nicht nur ein Strukturobjekt, sondern zusätzlich noch ein Ag-

gregationsknotenkontext in der Form Strukturknoten. Aggregationsknoten angegeben werden kann.

Des Weiteren sind innerhalb der kontext-sensitiven Funktion bereits Teile der Verhaltensspezifikation zu sehen. Es können Variablen und Strukturobjekte zugewiesen werden (Zeilen 10-13 u. 23-26), außerdem werden mit dem Schlüsselwort `FIRE` bereits Ereignisse gefeuert. `TIME_DIRECT` ist dabei die Variante, die die Ereigniswarteschlange umgeht und Ereignisse direkt abarbeitet. Ein Ereignis entspricht dann einer Methode ohne Ergebnis.

### 4.3.2 Spezifikation des Simulationsverhaltens

Die Spezifikation des Simulationsverhaltens findet in der Regel im `LOOP`-Abschnitt der DSIM Spezifikation statt. Wie oben gesehen, können aber auch innerhalb von kontext-sensitiven Funktionen Verhaltensbeschreibungen stehen.

Der `LOOP`-Abschnitt der Spezifikation wird direkt in den Simulator generiert. Was hier spezifiziert wird, wird vom Simulator in einer Endlosschleife durchlaufen.<sup>2</sup> Dies ist eine direkte Konsequenz aus der Wahl eines ereignisbasierten Simulatorsystems.

Innerhalb einer DSIM-Spezifikation kann es mehrere benannte `LOOP`-Blöcke geben um unterschiedliche Varianten implementieren zu können. In der Simulation von Sortieralgorithmen gibt es so einen Quicksort- und einen Bubblesort-Block. Zu Beginn der Simulation fragt der Simulator interaktiv den Benutzer, welcher Block ausgeführt werden soll.

Zweck des `LOOP`-Abschnittes ist das Einplanen von Ereignissen in die Warteschlange mittels des `FIRE`-Konstrukts. Dazu kann die Simulationsstruktur durchlaufen und überprüft werden. Dies geschieht mit Zugriffen auf Sprachkonstrukt-knoten und deren Attribute.

Listing 4.3 zeigt mehrere Prinzipien in der Verhaltensspezifikation von DSIM. Der Zugriff auf Gruppen von Objekten soll so einfach wie möglich sein. Dazu gibt es eine Reihe von Quantoren. In der ersten Zeile ist das `FOREACH`-Konstrukt zu sehen. Es ermöglicht den Zugriff auf alle Objekte eines bestimmten Typs, in diesem Fall alle `Transition`-Objekte. Mit dem Schlüsselwort `RANDOM` in der Spezifikation werden die Objekte in zufälliger Reihenfolge zurückgegeben. Auch auf Unterklasseninstanzen von abstrakten Klassen kann zugegriffen werden. Ein Vorteil des typ-basierten

---

<sup>2</sup>Unterbrochen werden kann die Schleife nur, falls der Simulator feststellt, dass sich in der Simulationsstruktur nichts geändert hat und keine Ereignisse mehr abgearbeitet werden müssen oder falls der Benutzer selbst die Simulation unterbricht.

```

1 FOREACH t IN [Transition] RANDOM{
2   IF(t.canFire) {
3     IFEVERY t.incomingPlaces->tokenCount > 0 {
4       FIRE preTransitionFire(t2) @ TIME_DIRECT;
5       FIRE postTransitionFire(t2) @ TIME_NOW + 1;
6     }
7   }
8 }

```

**Listing 4.3:** Ausschnitt der Spezifikation des LOOP-Blocks für Petri-Netze in DSIM

Zugriffs ist, dass Teilbäume auch nach einer Simulationsspezifikation noch verschoben werden können. Dies ist häufig der Fall, wenn die Spezifikation für eine visuelle Sprache refaktoriert wird. Möchte der Sprachspezifizierer beispielsweise nicht nur ein Petri-Netz darstellen, sondern eine Menge von Petri-Netzen, so fügt er üblicherweise noch eine Sicht ein, die die einzelnen Netze mit Namen darstellt. Die Netze selbst wandern im Strukturbaum eine Ebene tiefer. Die Verhaltensspezifikation der Simulation bleibt unberührt.

Das Listing zeigt mit dem IFEVERY-Konstrukt einen weiteren Quantor. Dabei wird überprüft, ob die Markenanzahl größer als null ist. Mit dem Punkt-Operator wird dabei auf Attribute der Simulationsstruktur zugegriffen. Dies können erweiterte Attribute oder auch Attribute des semantischen Modell sein. In diesem Fall ist `incomingPlaces` ein Pfadausdruck (vgl. Listing 4.1), der die Vorbedingung der Transition berechnet. Der Pfadausdruck liefert eine Menge von Objekten, das IFEVERY-Konstrukt wertet das Attribut `tokenCount` dieser Menge aus.

Listing 4.3 zeigt, wie Ereignisse gefeuert werden. Das erste Ereignis wird mittels FIRE-Konstrukt direkt ausgeführt (`TIME_DIRECT`). Ereignisse diesen Typs wirken sich unmittelbar auf die Simulationsstruktur aus, d. h. Veränderungen an der Struktur sind bereits in der Zeile nach dem FIRE-Konstrukt wirksam. Dies verhindert, dass zwei Transitionen gleichzeitig feuern, die dieselbe Vorbedingung haben, jedoch zu wenig Marken auf der Vorbedingung. Es würde sich hier also um eine race-condition handeln. Die zweite Variante zeigt das Einordnen in die Ereigniswarteschlange - ebenfalls mit dem FIRE-Konstrukt. Als Zeitangabe steht hier `TIME_NOW`, was für den nächsten Simulationszeitpunkt steht - plus einem nicht-negativen Ausdruck. So können Ereignisse beliebig weit in der Zukunft eingeplant werden.

Listing 4.4 zeigt mit REPEAT und WHILE zwei weitere Schleifenkonstrukte zur Iteration. Besonderheit an diesem Ausschnitt ist der direkte Zugriff im gesamten Strukturbaum auf Sprachkonstruktinstanzen mit dem Offset-Operator `#[i]`. Besonders einfach gestalten sich damit Zugriffe auf Objekte, wenn bereits bekannt ist, wie viele

```
1 ...
2
3 WHILE ((#[i]ListElement.size <= pivot.size) AND (i < right)) {
4     i = i + 1;
5 }
6
7 REPEAT SIZE(#[0]Root.elements) {
8     ...
9 }
```

**Listing 4.4:** Ausschnitt einer Spezifikation eines LOOP-Blocks mit Schleifenvarianten und Ausdrücken

Objekte zur Laufzeit existieren. Es darf nur ein `Root`-Objekt im Programm existieren, somit kann mit `#[0]Root` auf das einzige Wurzelobjekt zugegriffen werden.

### 4.3.3 Spezifikation von Ereignissen

Im Ereignisabschnitt der Simulationsspezifikation wird definiert, wie die Simulationsstruktur verändert werden soll. Ereignisse werden benannt und haben eine beliebig lange formale Parameterliste. Als Parameter sind Sprachkonstrukt-knoten oder Attributwerte spezifizierbar. Innerhalb von Ereignissen können beliebig viele *Simulationsmodifikationsoperationen* definiert werden. Diese modifizieren, wenn das Ereignis ausgeführt wird, die Simulationsstruktur. Simulationsmodifikationsoperationen dürfen nur innerhalb von Ereignissen und kontext-sensitiven Funktionen

```
1 EVENTS {
2     preTransitionFire(Transition transition) {
3         FOREACH place IN transition.incomingPlaces {
4             Token token = REMOVE(place.tokens, FIRST);
5         }
6     }
7
8     postTransitionFire(Transition transition) {
9         FOREACH place IN transition.outgoingPlaces {
10            Token token = INSERT(place.tokens, INSTANCEOF [Token], FIRST);
11        }
12    }
13 }
```

**Listing 4.5:** Spezifikation von Ereignissen für einen Petri-Netz-Editor

benutzt werden. Um kontextabhängige Animationen definieren zu können, kann abgefragt werden, welches Ereignis Auslöser für eine Änderungsoperation war. Somit können für die Animation notwendige Kontextinformationen aus der Simulation einfach gelesen werden (vgl. Abschnitt 5.11).

Ereignisse können auch selbst wieder Ereignisse einplanen. Listing 4.5 zeigt

die Spezifikation von zwei Ereignissen für eine Petri-Netz-Simulation. Beide Ereignisse werden ausgelöst, wenn eine Transition schalten kann. Das Ereignis `preTransitionFire` sorgt dafür, dass auf allen Stellen der Vorbedingung der Transition (berechnet durch den Pfadausdruck `transition.incomingPlaces` aus Listing 4.1) jeweils die erste Marke (`FIRST`-Schlüsselwort) entfernt wird. Dazu wird die Simulationsänderungsoperation `REMOVE` benutzt, die als Parameter einen Aggregationsknoten und eine symbolische Position erhält. Als Rückgabe liefert sie eine Referenz auf das entfernte Objekt.<sup>3</sup>

Das Ereignis `postTransitionFire` generiert auf allen Stellen der Nachbedingung der Transition ein neues `Token`-Objekt. Es bekommt ebenfalls als Parameter einen Aggregationsknoten, eine symbolische Position und liefert eine Referenz auf das neu erstellte Objekt als Rückgabe. Insgesamt lassen sich in Ereignissen bzw. kontextsensitiven Funktionen die folgenden vier Simulationsmodifikationsoperationen benutzen:

- `REMOVE` zum Löschen von Objekten, z. B.  
`REMOVE(StrukturObjekt.Aggregationsknoten, LAST);`
- `INSERT` zum Einfügen von Objekten, z. B.  
`INSERT(StrukturObjekt.Aggregationsknoten, FIRST);` Wird ein vorher gelöscht Objekt eingefügt, wird eine `MOVE`-Änderungsoperation erkannt. Wird ein neues Objekt eingefügt, wird eine `CREATE`-Änderungsoperation erkannt.
- `COPY` zum Klonen von Objekten (kopieren im Struktureditor-Sinn), z. B.  
`StrukturObj x = COPY(StrukturObjekt.Aggregationsknoten, LAST);`  
`x` kann danach mittels `INSERT` eingefügt werden.
- `CHANGE_VAL` implizite Änderungsoperation, die beim Ändern primitiver Attribute ausgelöst wird, z. B.  
`StrukturObjekt.value = VLInt(5);`

Die Modifikationen reichen aus, um beliebige Strukturbäume zu modifizieren, wie sie in von `DEViL` generierten Editoren existieren.

Listing 4.6 zeigt ein etwas komplexeres Ereignis aus dem `Pac-man-Editor`. Es spezifiziert das Verhalten des `Pac-man`, der einen Geist "fressen" darf. Zunächst wird dabei ein weiteres Ereignis direkt ausgeführt, das den Spielstand setzt (Zeile 2). Das Attribut `numberOfEatenGhosts` wird inkrementiert (eine `CHANGE_VAL` Operation wird ausgelöst). Das Ereignis `computeRotation` wird gefeuert. Zum Schluss werden die Spielfiguren entsprechend bewegt, d. h. die Geist-Figur wird

---

<sup>3</sup>Die Zuweisung auf der linken Seite in Zeile 4 kann auch weggelassen werden, da die Referenz nicht mehr gebraucht wird. Sie dient lediglich der Darstellung der Funktionalität des `REMOVE`-Konstrukts.

```
1 eatGhost(Tile from, Tile to, Pacman p, VLInt d){
2   FIRE incrementScore(#[0]Score, 1000) @ TIME_DIRECT;
3   #[0]Root.numberOFEatenGhosts = #[0]Root.numberOFEatenGhosts + VLInt(1);
4
5   FIRE computeRotation(p,d) @ TIME_DIRECT;
6
7   REMOVE(to.item, FIRST);
8   Item i = REMOVE(from.item, FIRST);
9   INSERT(to.item, i, FIRST);
10 }
```

Listing 4.6: Spezifikation eines Ereignisses für den Pac-man-Editor

gelöscht (Zeile 7) und Pac-man auf die Position des Geistes gesetzt (REMOVE und INSERT werden zu MOVE).

### 4.3.4 Spezielle Funktionalitäten

DSIM bietet noch einige weitere Eigenschaften, wie z. B. Logging-Mechanismen, Kachelzugriffsfunktionen, Interaktion sowie Pattern-Matching.

```
1 ...
2 Tile to = NEIGHBOUR_TILE(default, NEUMANN, ghost.tile, Pacman);
3 IF(NOTNULL(to) AND (ghost.eatable == VLBoolean(0))){
4   FIRE eatPacman(ghost.tile, to) @ TIME_DIRECT;
5 } ELSE{
6   to = NEIGHBOUR_EMPTY_TILE_RANDOM(default, NEUMANN, ghost.tile);
7   IF(NOTNULL(to)){
8     FIRE goGhost(ghost.tile, to) @ TIME_DIRECT;
9   }
10 }
11 ...
```

Listing 4.7: Spezifikation von Kachelzugriffsfunktionen im Pac-man Editor

**Kachelzugriffsfunktionen** Listing 4.7 zeigt zwei Kachelzugriffsfunktionen des Pac-man-Editors. Von DEViL generierte Editoren, die Gebrauch vom visuellen Matrix-Muster machen, also kachelbasierte Darstellungen nutzen, setzen eine ganz bestimmte semantische Struktur voraus um das Matrix-Muster anwenden zu können. Kachelzugriffsfunktionen nutzen diese immer gleiche Struktur aus um einfach auf die Umgebung einer bestimmten Kachel zugreifen zu können. Die Funktion `NEIGHBOUR_TILE` erwartet dabei eine Referenzkachel, sowie eine Nachbarschaftsdefinition (*von NEUMANN* im Beispiel, also Kacheln links, rechts, oben sowie unten von der Referenzkachel). Sie liefert als Ergebnis die Kachel, auf der ein Strukturobjekt eines bestimmten Typs (hier `Pacman`) beheimatet ist. Die Kachelfunktion `NEIGHBOUR_EMPTY_TILE_RANDOM` liefert zufällig eine leere



Kachel in der Umgebung zurück. Die Spezifikation in Listing 4.7 berechnet die Bewegung eines Geistes im Pac-man-Editor. Ist auf einer der Nachbarkacheln ein Pac-man-Objekt vorhanden, so wird der Pac-man gefressen, andernfalls wird der Geist zufällig bewegt.

Kachelzugriffsfunktionen sind ein Resultat aus der Implementierung des Pac-man Editors. In [128] konnte gezeigt werden, dass bereits eine kleine Anzahl verallgemeinerter Zugriffsfunktionen den Spezifikationsaufwand erheblich reduzieren kann. Neben den beiden vorgestellten Funktionen gibt es noch eine Reihe anderer Funktionen, die z. B. den Zugriff über eine Richtung oder die Kachelnummer liefern.

**Log-Ausgaben** Logging-Mechanismen sind bei der Analyse von Simulationen wichtig. DSIM bietet ein Log-Konstrukt an, das ähnlich wie die C `printf` Funktion arbeitet; also eine variable Parameterliste handhaben kann und verschiedene Logging-Stufen bietet. Da Logging sehr rechenintensiv sein kann, lässt es sich in DSIM jederzeit ausschalten.

```

1 ...
2 IF(#[0]Root.logging == VLBoolean(0))
3   LOGLEVEL = NONE;
4 ...
5 IF(fired > 1) {
6   LOG(WARNING, "Statechart ist nicht-deterministisch bei Übergang %s!",
7     transition.name);
8 }
9 ...

```

Listing 4.8: Log-Ausgaben

Listing 4.8 zeigt eine einfache Anwendung des Logging-Mechanismus. In Abhängigkeit eines Attributs in der Simulationsstruktur wird ggf. das Logging abgeschaltet (Zeile 3). In Zeile 6 wird eine Log-Ausgabe mit einem Parameter und Log-Level `WARNING` definiert. Generell werden während der Simulation geschaltete Ereignisse und angewendete Animationsmuster gelogged. Dies lässt sich jedoch jederzeit beliebig anpassen.

**Interaktion** DSIM erlaubt das Einbinden von Benutzerinteraktion innerhalb der Simulationsspezifikation. Prinzipiell gibt es zwei Arten von Interaktion: modale und kontinuierliche Interaktion. Die erste Variante hält die Simulation an und zwingt den Anwender eine Eingabe zu machen. Die kontinuierliche Interaktion ist nicht zwingend für das Fortschreiten einer Simulation notwendig. Sie kann zu jedem beliebigen Zeitpunkt geschehen. In der Simulation kann auf diese Interaktion eingegangen werden, muss es jedoch nicht. Der Spezifikation ist dies nicht

unbedingt anzusehen, wie Listing 4.9 zeigt. Dort werden zwei Benutzerereignisse spezifiziert - der Einfachheit halber zwei Implementierungen in einem Beispiel. Dies geschieht im optionalen CONFIGURATION-Block von DSIM. Das erste Ereignis registriert die *Pfeil-nach-oben* Taste unter dem Namen `keyUp`. Das zweite registriert eine beliebige Taste (`<Key-Press>`) unter dem Namen `input` und konvertiert das Ergebnis in den `VLString`-Typ, wobei der Wert des `VLString` die Taste selbst ist (`%A`. Tcl-Notation). In der Simulation kann eine solche Benutzerinteraktion über das

```
1 CONFIGURATION {
2   // Pac-man
3   REGISTER "<Up>" AS keyUp CONVERT "1" VLInt;
4
5   // Statecharts
6   REGISTER "<Key-Press>" AS input CONVERT "%A" VLString;
7 }
8
9 LOOP {
10  // Statecharts
11  IF (ACTION_EVENT(input) AND
12     ACTION_EVENT_VALUE(input) == outgoingTransition.label) {
13    ...
14  }
15
16  // Pac-man
17  IF (ACTION_EVENT(keyUp)) {
18    FIRE movePacman(1) @ TIME_DIRECT;
19  }
20 }
```

Listing 4.9: Benutzerinteraktion

`ACTION_EVENT(eventName)` Schlüsselwort abgefragt werden. Ist es aufgetreten, kann man den Wert mittels `ACTION_EVENT_VALUE(eventName)` abfragen (Zeile 12). So können sowohl modale als auch kontinuierliche Interaktion realisiert werden.

### Mit Spezifikationen der Form

```
Transition toFire =
  SELECT(list, "Select transition to fire.", NOTNULL);
```

können Dialoge geöffnet werden, in denen dem Benutzer eine Auswahl an möglichen Sprachkonstrukten gegeben wird. Die Liste der Sprachkonstrukte wird dem Anwender dabei in der graphischen Oberfläche präsentiert. Sollten nicht alle Objekte eine graphische Repräsentation haben, so wird eine textuelle Auswahlliste präsentiert. In Abbildung 4.5 ist das Resultat eines solchen Konstrukts zu sehen. Die Transitionen, die feuern können, werden rot hinterlegt dargestellt. DEViL stellt sicher, dass die Auswahl des Benutzers korrekt ist, d. h. es muss eines der möglichen Objekte ausgewählt werden und die Auswahl darf nicht leer sein. In ähnlicher Form können auch Fragedialoge erstellt werden, die primitive Werte als Eingabe erwarten.

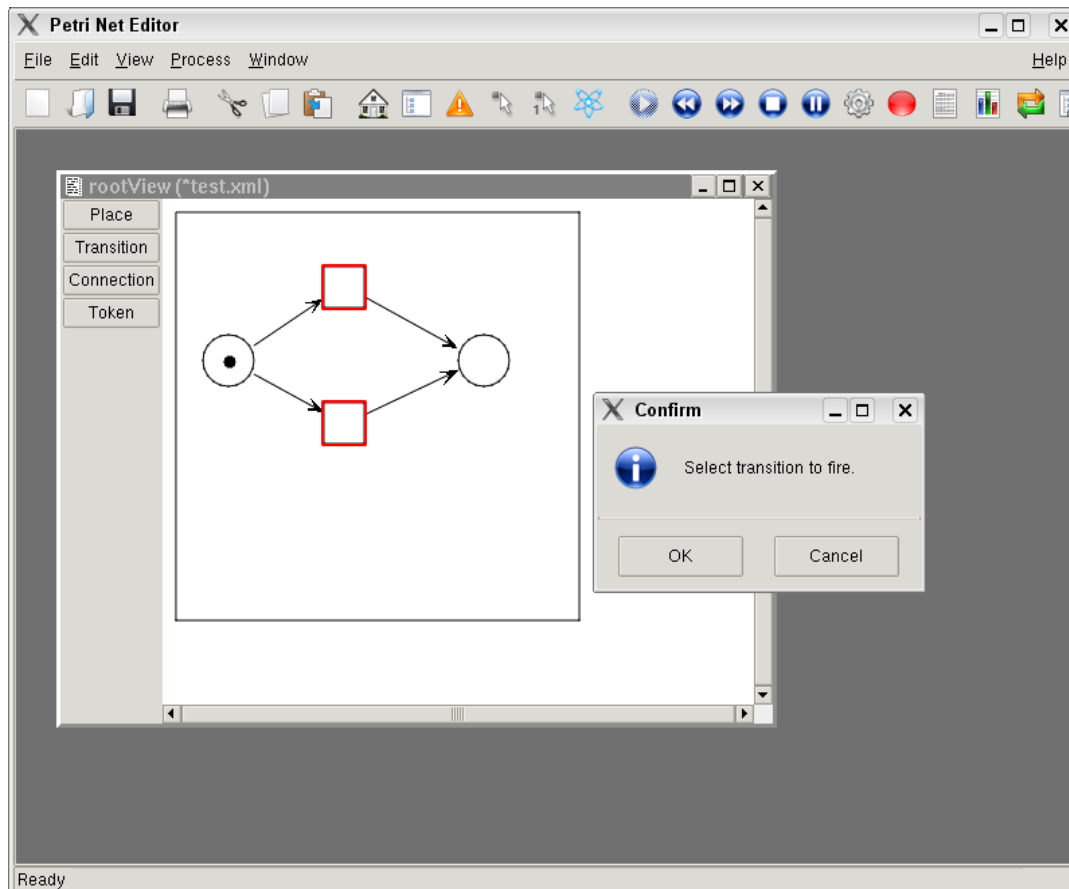


Abbildung 4.5: Interaktionsdialog

**Konfiguration der Simulation** Der oben erwähnte optionale `CONFIGURATION`-Block dient neben der Definition von Benutzerinteraktion dazu, auch benutzerdefinierte Funktionen, z. B. externe C-Header einzubinden (genauso wie in C mit dem `#include`-Makro). Außerdem können hier einige globale simulationsspezifische Einstellungen vorgenommen werden. Die Wichtigste davon ist die Simulationsstrukturwurzel. Mit der Spezifikation

```
SIMULATION_ROOT PetriNet
```

kann die Wurzel der Simulationsstruktur angegeben werden. Existieren in einer Petri-Netz-Spezifikation mehrere Objektinstanzen vom Typ `PetriNet`, so wird der Benutzer durch Hinzufügen einer solchen Zeile vor der Simulation gefragt, auf welcher Petri-Netz-Instanz simuliert werden soll. So können mehrere Petri-Netze komfortabel nebeneinander existieren, wobei immer nur genau ein Netz simuliert wird.

**Mustererkennung** In Sprachen der vierten Kategorie der Klassifikation müssen in der Regel Sprachkonstrukte gefunden werden, die bestimmte Eigenschaften

erfüllen. Meist werden diese Eigenschaften in der Sprache selbst durch (visuelle) Regeln beschrieben. Die Regeln definieren dabei eine räumliche Semantik zwischen Sprachkonstrukten. In einer Verkehrssimulation kann z. B. die Rechts-vor-links-Regel visuell durch ein Kachelmuster ausgedrückt werden. Häufig sind dies nicht nur strukturelle Eigenschaften, sondern konkrete Wertbelegungen von Attributen. DEViLs Pfadausdrücke können dies nicht vollständig leisten. Aus diesem Grund wurde ein XPath Auswerter [2] integriert. So kann die gesamte Simulationsstruktur mit XPath-Ausdrücken ausgewertet werden.

```
1 ...
2 VList[Streetsection] listOfProducer = MATCH("//Streetsection[@prodConsNorth=1] |
3                                     //StreetsectionNE[@prodConsSouth=1] |
4                                     //StreetsectionWE[@prodConsWest=1], OF [
5                                     Streetsection]);
6 FOREACH producer IN listOfProducer {
7     FIRE produceCar(producer);
8 }
```

**Listing 4.10:** Mustererkennung mit XPath

Listing 4.10 zeigt einen Ausschnitt aus einer Verkehrssimulation. Dem `Match`-Konstrukt wird ein XPath Ausdruck übergeben, der alle Straßenabschnitte der Simulationsstruktur aufammelt, die als Produzenten definiert sind (`prodCons*` Attribut auf 1).

### 4.3.5 Textuell vs. visuell

In den eingangs erwähnten verwandten Generatorsystemen wird die Simulationspezifikation häufig visuell erstellt. Dies ist insbesondere bei Systemen auf Basis von Graphtransformation der Fall. Dies ist zunächst ein Vorteil gegenüber DSIM, allerdings muss erwähnt werden, dass DSIM darauf ausgelegt ist, sehr kompakt auch komplexes Verhalten zu spezifizieren. In den genannten Generatorsystemen müssen dagegen bereits bei sehr einfachen Problemen viele Regeln erstellt werden, oft sogar in unterschiedlichen Kalkülen wie bei GenGed (vgl. Abschnitt 2.7.2). Um auch kaskadierende Regelanwendungen auszudrücken muss der Graph in der Regel künstlich erweitert werden. Im schlimmsten Fall werden sogar Simulation und Animation innerhalb der Graphstruktur vermischt.

Unter anderem operiert DSIM auf der Simulationsstruktur bzw. dem semantischen Modell der Sprache. Eine graphische Repräsentation muss noch nicht zwingend vorliegen. Eine Spezifikation auf Basis einer graphischen Notation hätte den Nachteil, dass Änderungen in der graphischen Darstellung auch Änderungen in der Simulationsspezifikation nach sich ziehen würden. Dies widerspricht der

Trennung von Layout und Logik. Trotzdem ist es vorstellbar, dass für DSIM eine visuelle Sprache “on-top” definiert wird. Diese wäre wahrscheinlich aufgrund der Verwendung von Pfadausdrücken wiederum sehr textlastig. Ein weiterer Nachteil ist, dass die Manipulation von strukturell weit entfernten Objekten nur schlecht visuell ausgedrückt werden kann.

## 4.4 Verwandte Arbeiten

Ein Ansatz zur Spezifikation von simulierbaren visuellen Sprachen ist der der Abstract State Machines [44] (ASM) und der dazu gehörigen von Microsoft entwickelten domänenspezifischen Sprache AsmL [66]. In [19] wird AsmL benutzt um das Verhalten von DSLs zu spezifizieren. AsmL-Spezifikationen sehen wie Pseudo-Code auf abstrakten Datenstrukturen aus und beinhalten regel-basierte Konstrukte. Des Weiteren erlauben sie auch die Verwendung von Iteratoren über Datenstrukturen. In [15] werden ASMs für eine Fallstudie zur Spezifikation einer Lichtkontrollanlage benutzt.

In [31] und [117] werden Graphersetzungsregeln zur Simulation visueller Sprachen benutzt. Hier muss ein Vorher-/Nachher-Zustand in Form von Regeln spezifiziert werden, der dann über ein Transformationswerkzeug, z.B. AGG, eine Simulation in Gang setzt.

DSIM ist nicht als eine GPL auf Datenstrukturen so wie AsmL zu verstehen, sondern speziell auf die Eigenschaften visueller Sprachen zugeschnitten. Der Sprachumfang ist nicht so mächtig wie bei AsmL, jedoch ausreichend und es werden viele Konzepte bereitgestellt, um einfach auf Sprachkonstruktinstanzen zuzugreifen (Iteratoren, Quantoren, Pfadausdrücke). Außerdem integriert DSIM Konzepte von Simulationssprachen, die bisher noch nirgends auf dem Gebiet der VL-Simulation zu finden sind. Diese sind:

- Eigenständige Simulationsstruktur.
- Gruppieren von Simulationsobjekten.
- Kontext-sensitive Funktionen.
- Verschiedene Interaktionsmechanismen.

Gegenüber der Spezifikation durch Graphtransformation ist die einfache Lesbarkeit auch bei komplexen Simulationen als Vorteil zu nennen. Der ereignisbasierte Ansatz erlaubt eine sehr flexible Handhabung der Simulationszeit. Dieser hat sich bereits in der Literatur beim DEVS-System [118] bewährt und kommt auch im DiaMeta-System zum Einsatz. Die Erweiterung bzw. Einschränkung der semantischen Struktur zur Simulationsstruktur erlaubt es, auch komplexe Simulationen abzubilden und das System zu modularisieren.



# 5 Animationskonzept

## Inhalt

---

<b>5.1</b>	<b>Das Animationsframework</b>	<b>114</b>
5.1.1	Einordnung der Animationskomponente	114
5.1.2	Der " <i>interesting-events</i> "-Ansatz	116
5.1.3	Animation durch lineare graphische Interpolation	117
5.1.4	Der deklarative Animationsansatz	122
5.1.5	Berechnung der Animation	126
5.1.6	Dynamische Animationsobjekte	127
5.1.7	Statische Animationsobjekte	128
5.1.8	Graphische Anknüpfungspunkte	130
5.1.9	Informationsverlust zwischen Simulation und Animation	131
<b>5.2</b>	<b>Eigenschaften generierter Umgebungen</b>	<b>135</b>
<b>5.3</b>	<b>Diskussion</b>	<b>138</b>
<b>5.4</b>	<b>Verwandte Arbeiten</b>	<b>139</b>

---

In den vorherigen Kapiteln wurde beschrieben, wie es in DEViL möglich ist mit der Hilfe von DSIM visuelle Programme zu simulieren. Die Simulation mit DSIM bedeutet dabei die Manipulation einer Instanz einer visuellen Sprache. Dies geschieht mit dem generierten Simulator in diskreten Schritten, d. h. die graphische Darstellung wird nicht kontinuierlich angepasst. Durch diese diskreten Zustandsveränderungen kann zwar die Simulation eines Systems durchgeführt werden und auch bereits Aussagen über ein Verhalten getroffen werden, es führt jedoch nicht auf natürlichem Wege zu einem schnellen Verständnis des Anwenders, da dieser abrupte Zustandsänderungen nicht gut erfassen kann.

In den folgenden Abschnitten soll nun die vorhandene Simulation um eine geeignete Animationskomponente erweitert werden. Die Animation soll es ermöglichen visuelle Sprachen der Typen eins bis vier der Klassifikation einfach und flexibel zu animieren. Dazu werden graphische Interpolation und eine neue Form der visuellen Muster verwendet. Um auch Sprachen mit hohem Niveauunterschied zwischen Simulation und Animation abdecken zu können, wird das Strukturkopplungsprinzip verwendet.

### 5.1 Das Animationsframework

Zunächst will ich auf das allgemeine Simulationskonzept und wie die Animation darauf aufbaut eingehen. Im Anschluss werde ich zeigen, wie Animation in DEViL spezifiziert wird und welcher Ansatz nicht zum Erfolg führt.

#### 5.1.1 Einordnung der Animationskomponente

In Abbildung 5.1 ist die Struktur eines generierten Editors mit Simulations- und Animationserweiterung zu sehen. Auf der linken Seite ist die bereits eingeführte Transformationskette aus Abbildung 2.24 auf Seite 64 zu sehen. Neu ist die erweiterte Kette auf der rechten Seite.

Der Simulator arbeitet auf der Simulationsstruktur, die eine Erweiterung (oder Einschränkung) der semantischen Struktur der visuellen Sprache ist. Die Simulationsstruktur wurde bereits durch DSIM erstellt. Die standardisierten Änderungsoperationen auf der Simulationsstruktur führen zur *Baumfragmentstruktur*. Diese enthält die, jeweils nach einem Simulationsschritt veränderte Simulationsstruktur und zusätzlich noch Referenzen auf Knoten der Struktur, die im aktuellen Simulationsschritt gelöscht wurden, also nicht mehr Teil der Simulationsstruktur sind. Somit entsteht aus der Simulationsstruktur ein Baum, der noch quasi unsichtbare Referenzen auf gelöschte Objekte enthält. Durch den Animationsalgorithmus wird die Baumfragmentstruktur durchlaufen und entsprechend eines später noch



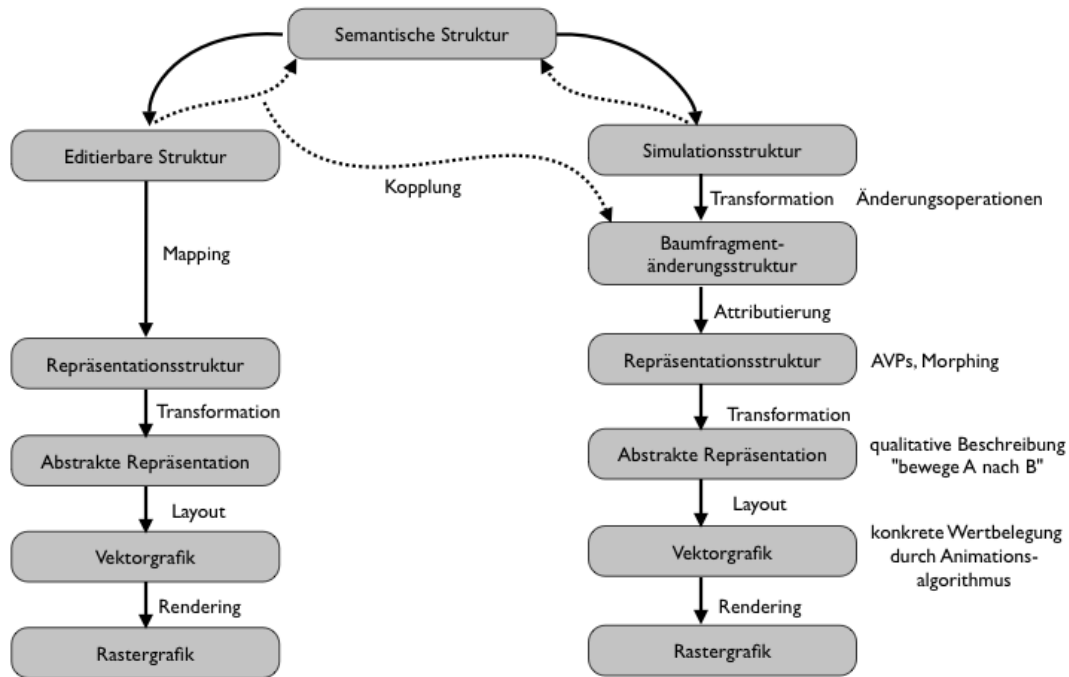


Abbildung 5.1: Einordnung der Animationskomponente

vorgestellten Plans mit Animationsfunktionen attribuiert. Die entstandene Repräsentationsstruktur enthält somit Animationsmuster und Morph-Funktionen. Diese Funktionen werden später noch erläutert. Die abstrakte Repräsentation enthält qualitative Beschreibungen zur Animationsdarstellung, wie "bewege Objekt A in die Einfügeposition B". Sie dient als Eingabe für den Animationsalgorithmus.

Das Layout im Sinne einer Animation besteht nicht nur aus einem statischen Bild, sondern aus einer Reihe von *Keyframes*, zwischen denen interpoliert wird. Das Rendering sorgt letztendlich für die Darstellung auf dem Monitor. Dies ist eine Standardfunktion des zu Grunde liegenden Systems.

Der mit "Kopplung" benannte Pfeil stellt noch eine Besonderheit des Animationsframeworks dar: die automatische Animation von gekoppelten Strukturen in DEViL. Werden Änderungen des Simulators von der Simulationsstruktur an die semantische Struktur der visuellen Sprache übertragen, so triggert dies ggf. Änderungsoperationen in gekoppelten Strukturen. Gekoppelte Strukturen sind ein Spezialfall der editierbaren Strukturen: Sie haben ein gegenüber der semantischen Struktur anderes Niveau. Es können auch mehrere dieser Strukturen existieren. Die Änderungen werden wiederum in die Baumfragmentstruktur eingetragen. Die Animationskomponente ist somit in der Lage, auch gekoppelte Strukturen geeignet und mit denselben Mechanismen zu animieren wie auch die Simulationsstruktur.

### 5.1.2 Der “interesting-events”-Ansatz

Der erste Ansatz, die Animation zu erreichen, folgte dem Paradigma der “interesting-events”, den man bereits seit vielen Jahren erfolgreich in der Algorithmenanimation einsetzt [114]. Dabei wird der Algorithmus (also die Simulationsspezifikation) an bestimmten “interessanten” Stellen durch Aufrufe von Animationsfunktionen annotiert. Im Petri-Netz-Beispiel sind dies die Stellen, an denen Marken gelöscht bzw. erstellt werden.

```

1  EVENTS{
2    preTransitionFire(SET incomingPlaces OF
3      Place, Transition t) {
4      incomingPlaces.marks--;
5      FOREACH place IN incomingPlaces {
6        ACTION "tokenDrawing" (place, t);
7      }
8    }
9
10   postTransitionMove(Transition t, Place p) {
11     ACTION "tokenDrawing" (t, p);
12     FIRE postTransitionDecrement(p);
13   }
14
15   postTransitionDecrement(Place p) {
16     p.marks++;
17   }
18 }
19
20 ANIMATIONVIEW rootView {
21   ANIMATION tokenDrawing (Place p, Transition t){
22     DRAWING token = tokenDrawing @ POSITION(p);
23     MOVE token (POSITION(p),POSITION(t)) 2sec,
24               EASE_IN;
25   }
26   ANIMATION tokenDrawing (Transition t, Place p){
27     DRAWING token = tokenDrawing @ POSITION(t);
28     MOVE token (POSITION(t), POSITION(p)) 2000 msec,
29               EASE_BACK;
30   }
31 }

```

Listing 5.1: Annotation von DSIM durch Animationsereignisse

Abbildung 5.1 verdeutlicht die Annotation der DSIM Sprache mit Animationsfunktionen. In Zeile 21 bzw. 26 ist die Definition zweier Animationsfunktionen zu sehen. Hier wird eine generische Zeichnung instanziiert, die eine Marke darstellt (Zeile 22 bzw. 27) und diese dann bewegt (Zeile 23 u. 28). In den Ereignissen können die Funktionen aufgerufen werden.

Dieser Ansatz hat mehrere schwerwiegende Nachteile:

1. Größe und Position von Sprachkonstrukten im Folgezustand sind unbekannt.
2. Der Animator muss sich darauf verlassen, dass der Spezifizierer der Simulation den Code korrekt annotiert.

3. Es gibt keine getrennte Entwicklung von Simulation und Animation.
4. Zusätzliche Fehlerquellen, da der Simulationscode eine weitere Komplexitätsebene erhält.
5. Die Animationsabbildung ist nicht mehr verlässlich formal richtig.
6. Durch zusätzliche Aufrufe von Animationsfunktionen ergibt sich ein erhöhter Spezifikationsaufwand.

Das Erstellen und Löschen von Objekten wirkt sich nicht nur unmittelbar auf das Objekt selbst aus, sondern kann auch dazu führen, dass Objekte in der Umgebung bzw. an ganz anderen Positionen im Strukturbaum ihre Position und Größe verändern. D. h. also, dass Animationsfunktionen Positionsangaben nur grob abschätzen können, da sie die graphische Repräsentation des folgenden Simulationsschrittes nicht kennen. Außerdem kann so nur eine Animation über bestimmte Animationsobjekte beschrieben werden, also nicht deklarativ über alle veränderten Objekte. Dieser Punkt ergibt sich daraus, dass DEViL syntax-gerichtete Struktureditoren generiert.

Wenn Simulation und Animation nicht mehr getrennt werden führt dies zu einer nicht gewünschten verschränkten Entwicklung. Dies ist vergleichbar mit der Programmierung von Logik und Layout, wie man es heute in vielen anderen Softwareentwicklungsbereichen eigentlich vermeiden möchte. Klassisches Beispiel hierfür ist die Entwicklung von Webanwendungen in denen Layout (HTML) und Logik (PHP, Javascript...) getrennt werden. Die Separierung von Simulation und Animation führt zu einer verbesserten Entwicklung im Team, wobei es auf der einen Seite Spezialisten für die Simulation und auf der anderen Seite Experten für die Animation gibt.

Außerdem entstehen mögliche neue Fehlerquellen. So kann es sein, dass die Simulation zwar korrekt funktioniert, die Animation dieses jedoch nicht richtig widerspiegelt. Animationsfunktionen können falsch sein oder sogar ein völlig verzerrtes Bild der Simulation darstellen. Die Animationsabbildung wäre somit nicht mehr verlässlich. Zuletzt ist der erhöhte Spezifikationsaufwand zu nennen, der durch explizite Funktionsaufrufe entsteht.

Diese Aspekte führen also für die imperative Spezifikation im Sinne der "interesting Events" in eine Sackgasse.

### 5.1.3 Animation durch lineare graphische Interpolation

Die Alternative, die die oben genannten Nachteile umgeht, ist die lineare graphische Interpolation.

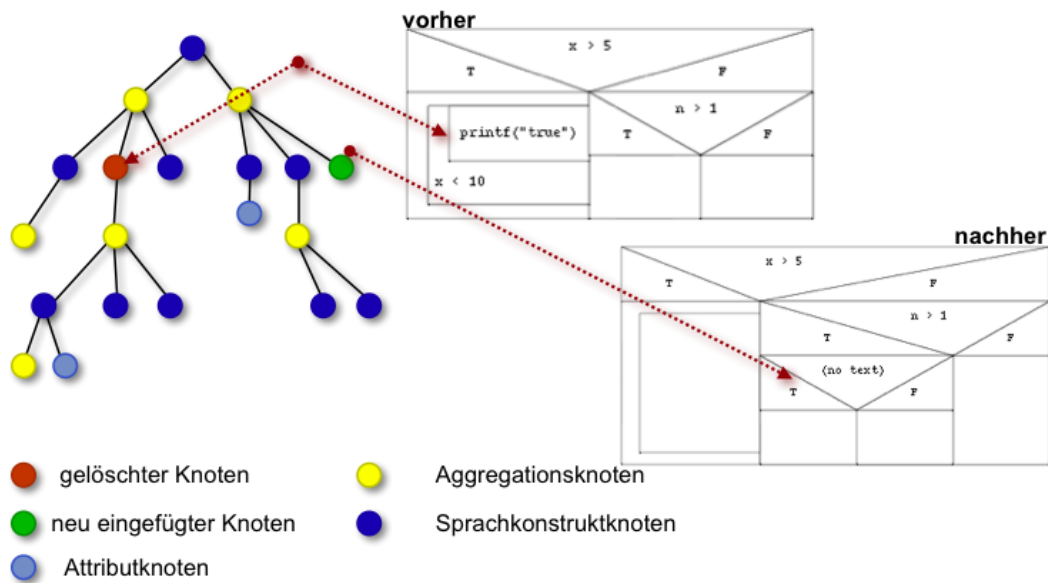


Abbildung 5.2: Graphische Repräsentation

Da der Zustand der graphischen Repräsentation vor und nach einem Simulationsschritt bekannt ist - er kann durch jeweils einen Attributauswerterlauf auf der Simulationsstruktur vor und nach einem Simulationsschritt berechnet werden - können die Größe und die Position aller Strukturobjekte vollständig bestimmt werden. Eine kontinuierliche oder auch "weiche" Animation kann so durch die lineare graphische Interpolation entstehen.

Abbildung 5.2 zeigt einen Strukturbaum für eine Instanz eines Nassi-Shneiderman Diagramms, wobei während der Simulation sowohl ein Sprachkonstruktknoten (samt Unterbaum) gelöscht wird (roter Knoten) und an einer anderen Stelle im Baum ein Knoten entsteht (grüner Knoten). Die Korrespondenz zwischen graphischer Darstellung und Struktur ist durch die gestrichelten Pfeile verdeutlicht. Die graphische Darstellung wird durch den Attributauswerter berechnet, der auf die bekannte Weise (durch die bereits eingeführten visuellen Muster) jeweils vor und nach einem Simulationsschritt durch den Strukturbaum läuft. Die Ergebnisse des zweiten Laufs werden im nächsten Simulationsschritt wiederverwendet. Die Ausgaben (Positionen/Größe/Transparenz/Rotation/Farbe aller Sprachkonstruktknoten) werden zur Weiterverarbeitung zwischengespeichert.

Die Animation ist also die lineare graphische Interpolation zwischen beiden Zuständen. Dies führt zu folgendem deklarativen Spezifikationsansatz in DEViL:

*"Lineare graphische Interpolation ist die Regel, die Ausnahme muss (deklarativ) spezifiziert werden."*

Um diese Aussage zu verstehen, müssen wir erneut einen Blick auf die Spezifikation mittels DSIM werfen. Wir tun dies wieder anhand der Spezifikation für simulierte

Petri-Netze aus Listing 4.5 auf Seite 104.

In dieser Spezifikation gibt es zwei Aktionen, die für die Animation von Bedeutung sind:

1. der REMOVE Aufruf, der eine Marke aus der Vorbedingung löscht.
2. der INSERT Aufruf, der eine neue(!)<sup>1</sup> Marke in der Nachbedingung erstellt.

Beide Aktionen werden von der Animationskomponente erkannt und es werden Standardanimationsaktionen berechnet. Hier seien nochmals die vier unterschiedlichen Änderungsoperationen in DSIM zusammengefasst:

- REMOVE zum Löschen von Objekten, z. B.  
`REMOVE (StrukturObjekt.Aggregationsknoten, LAST);`
- INSERT zum Einfügen von Objekten, z. B.  
`INSERT (StrukturObjekt.Aggregationsknoten, FIRST);` Wird ein vorher gelöscht Objekt eingefügt, wird eine MOVE-Änderungsoperation erkannt. Wird ein neues Objekt instanziiert, wird eine CREATE-Änderungsoperation erkannt.
- COPY zum Kopieren von Objekten, z. B.  
`StrukturObj x = COPY (StrukturObjekt.Aggregationsknoten, LAST);`  
x kann danach mittels INSERT eingefügt werden.
- CHANGE\_VAL implizite Änderungsoperation, die beim Ändern primitiver Werte ausgelöst wird, z. B.  
`StrukturObjekt.value = VLInt(5);`

Diese vier Änderungsoperationen triggern automatisch Animationsfunktionen, ohne dass der Animator etwas dazu tun muss.

- CREATE (aus INSERT abgeleitet) lässt ein Objekt von der Größe 0 bis zu seiner endgültigen Größe wachsen.
- REMOVE lässt ein Objekt schrumpfen, bis es nicht mehr zu sehen ist.
- MOVE (aus REMOVE und INSERT abgeleitet) bewegt ein Objekt von der Position in Simulationsschritt  $S_n$  bis zur Position in Simulationsschritt  $S_{n+1}$ .
- COPY kopiert ein Objekt. Dieses wird bei INSERT von der Position des kopierten Objekts bis zur neuen Einfügeposition bewegt. Dabei verändert das kopierte Objekt seinen Transparenzzustand von durchsichtig bis sichtbar.
- CHANGE\_VAL ist an Attributknoten gebunden, was einerseits Referenzen sein können oder primitive Datentypen wie Zeichenketten. Eine Änderung löst hier eine Triggerung von Animationsfunktionen im übergeordneten Sprachkonstrukt aus.

---

<sup>1</sup>Die Marke wird lt. Petri-Netz-Definition neu generiert und nicht bewegt.

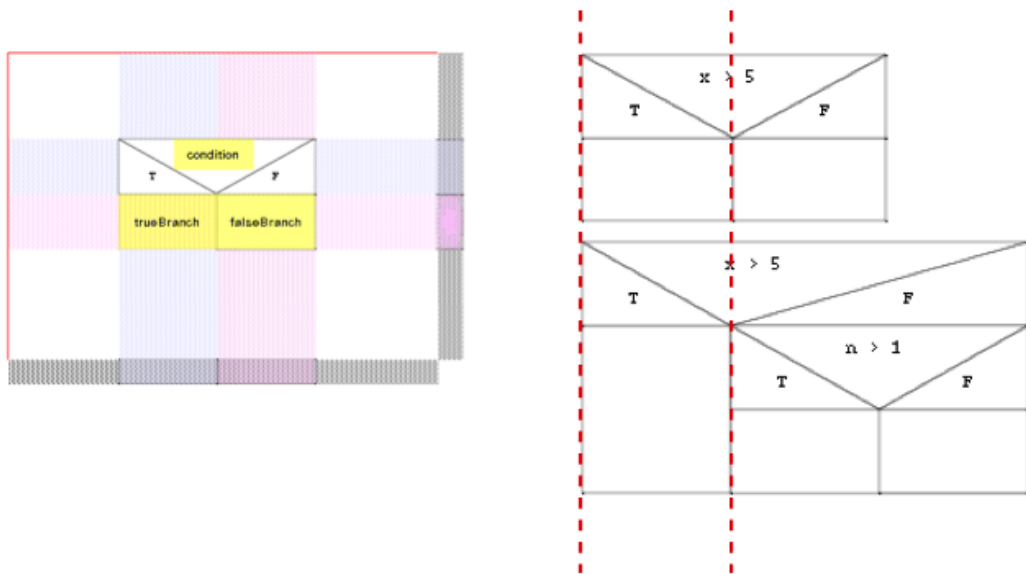


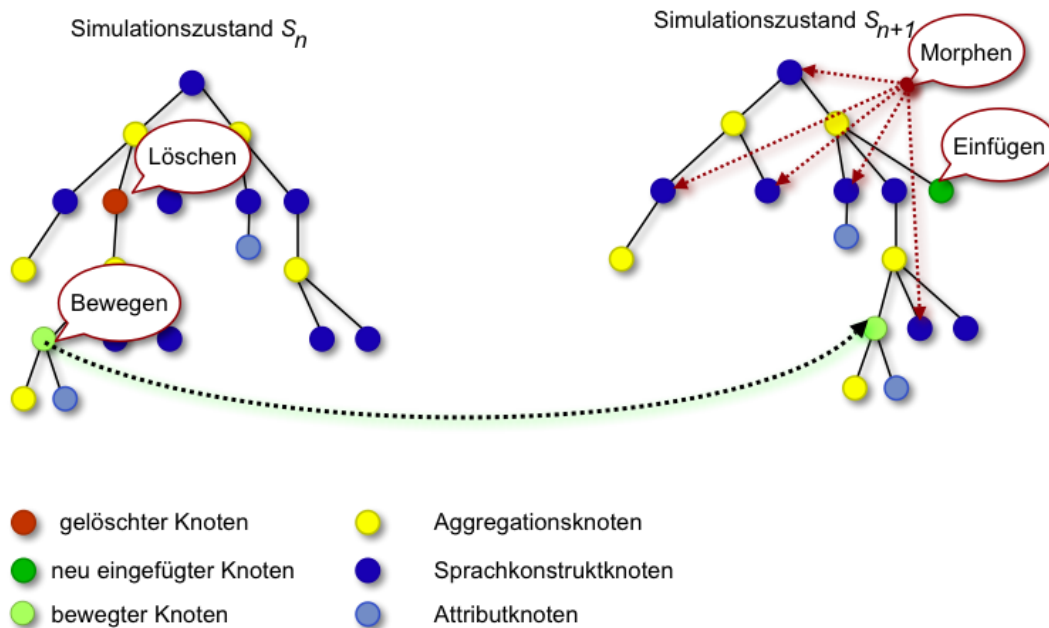
Abbildung 5.3: Nicht lineare Ausdehnung einer generischen Zeichnung

Die Simulationsänderungsoperationen stellen die Anknüpfung für die Animation dar. Durch die schmale und einfache Schnittstelle kann die Animation formal aus der Simulation abgeleitet werden. Die Schnittstelle besteht nur aus der Simulationsänderungsoperation und einer Objektreferenz. Die Animation ist also ein konsequentes Abbild der Simulation.

Wie bereits erwähnt löst eine Änderungsoperation nicht nur eine Änderung der Darstellung des aktuellen Objekts aus, sondern kann auch noch die Umgebung beeinflussen. Ein Beispiel ist eine Warteschlange. Wird ein Element entfernt, so müssen alle anderen Objekte in der Warteschlange aufrücken. Im Waschstraßenbeispiel würde es sich dabei um Objekte vom Typ "Auto" handeln. Diese müssten entsprechend der Eigenschaften von Autos langsam anfahren und bis kurz vor ihrer endgültigen Position wieder langsam abbremsen ("Easing").

Diese Auswirkungen auf benachbarte Objekte und auch alle anderen sind in DEViL, bedingt durch die generischen Zeichnungen und deren Ausdehnungsverhalten, nicht durch ein einfaches lineares Schrumpfen oder Vergrößern zu erreichen.

Abbildung 5.3 zeigt auf der linken Seite eine generische Zeichnung für einen bedingten Ausdruck in Nassi-Shneiderman-Diagrammen, wie er auch in DEViL zur Spezifikation benutzt wird. Er beinhaltet einen Container `trueBranch` und einen Container `falseBranch` - jeweils zum Einfügen von Anweisungssequenzen. Wird in den `falseBranch`-Container ein neues Element eingefügt, so dehnt sich die generische Zeichnung nicht gleichmäßig aus (rechte Seite). Der `trueBranch` bleibt



**Abbildung 5.4:** Aktionen, die bei der graphischen Interpolation ausgelöst werden müssen.

gleich groß, während der `falseBranch` gedehnt wird - verdeutlicht durch die rot gestrichelten Linien.

Eine lineare graphische Interpolation mit konstanter Ausdehnung reicht hier also nicht aus um eine flüssige Animation zu erreichen. An dieser Stelle führt das Animationsframework das so genannte "Morphen" von Sprachkonstrukt-knoten ein, das alle graphischen Primitive einer generischen Zeichnung (d. h. alle Linien, Rechtecke, Bilder, Polygone, Kreise etc.) flüssig vom Urzustand in den Endzustand überführt. Auch Farbänderungen werden erkannt und harmonisch entsprechend einer Funktion im HSB-Farbraum angepasst. Ebenso werden Rotationen und Transparenz gemorpht.

Abbildung 5.4 zeigt die auf einem Strukturbaum ausgelösten Animationsfunktionen. Neben dem Löschen und Erstellen von graphischen Objekten muss für alle anderen nicht modifizierten Objekte das Morphen ausgeführt werden um Platz zu schaffen bzw. freizugeben. Wichtig ist, dass der gesamte Baum von der Wurzel durchlaufen werden muss, da eine Positionsänderung sich theoretisch auf jedes Element im Baum auswirken kann. Auch diese Aufgabe übernimmt die Animationskomponente automatisch. Das Triggern der Standardanimationsfunktionen für die Modifikationsoperationen von DSIM wird automatisch aus einer Simulation generiert - ebenso wie das Morphen von Objekten. Der Sprachentwickler erhält aus einer Simulationsspezifikation automatisch eine dazu passende Animation ohne eine Zeile zusätzlichen Code schreiben zu müssen.

### 5.1.4 Der deklarative Animationsansatz

Die Standardanimationsfunktionen reichen häufig aus um anspruchsvolle Animationen zu erreichen, wie die Evaluation zeigt. Allerdings kann es notwendig sein, die Standardfunktionen anzupassen.

Aus dem Petri-Netz-Beispiel wird so eine Animation generiert, die Marken, falls sie gelöscht werden, zusammenschrumpfen lässt und neu erstellte Marken langsam erscheinen lässt. Dies ist jedoch noch nicht das gewünschte Ergebnis: die gelöschten Marken sollen eigentlich nicht sofort verschwinden, sondern zunächst bis zur Transition, die geschaltet hat, fliegen. Neu erstellte Token sollen nicht langsam erscheinen, sondern von der Transition bis zur endgültigen Position fliegen. Beide Aktionen sollen nacheinander geschehen.

Um dies zu spezifizieren, werden die so genannten *“Animated Visual Patterns”* (kurz AVPs) benutzt, die ebenso deklarativ wie ihre Pendanten, die visuellen Muster, graphische Eigenschaften kapseln, diesmal jedoch Animationsfunktionen. AVPs werden ebenso wie die visuellen Muster in der Spezifikation für die graphische Repräsentation spezifiziert und dort an Sprachkonstrukt-knoten gebunden. Sie überschreiben das Standardverhalten der Animationskomponente.

AVPs reagieren auf Änderungsoperationen im Simulator, sie *“lauschen”* also immer auf die vier Typen der Änderungsoperationen. Sie überschreiben das Standardanimationsverhalten an dem Knoten, an dem sie angewendet werden. Des Weiteren implementieren AVPs immer ein zeitliches Verhalten, d. h. sie beschreiben, wann sie während der Animation ausgeführt werden und wie lange ihre Ausführung dauert. Dies ist die *Animationszeit*.

Im Beispiel der Petri-Netze muss also um eine geeignete Animation zu erreichen, am Sprachkonstrukt-knoten Token der graphischen Spezifikation, die AVPs AVPOnRemoveMove sowie AVPOnCreateMove gebunden werden (siehe auch Listing 5.2).

```

1  SYMBOL rootView_Token INHERITS VPForm, VPSetElement,
2                                AVPOnRemoveMove, AVPOnCreateMove
3  COMPUTE
4    SYNT.drawing= ADDR OF(TokenDrawing);
5
6    SYNT.onRemoveMoveEndPosition = POSITION(1);
7    SYNT.onRemoveMoveAnimationTime = 1;
8
9    SYNT.onCreateMoveStartPosition = POSITION(1);
10   SYNT.onCreateMoveAnimationTime = 2;
11  END;
```

Listing 5.2: Anwendung der AVPs



AVPonRemoveMove wird aufgerufen, wenn ein Token entfernt wird. Ist dies der Fall, soll das Token bis zur Position der Transition fliegen, die geschaltet hat. Erreicht wird dies durch das Überschreiben des Kontrollattributs `SYNT.onRemoveMoveEndPosition`. Auf der rechten Seite steht ein Makro, das später erläutert wird.

Falls ein Token erstellt wird, soll es von einer Startposition bis zur endgültigen Position bewegt werden. Dies wird durch Überschreiben des Kontrollattributs `SYNT.onCreateMoveStartPosition` des AVPs `AVPonCreateMove` erreicht.

Damit ein Token zunächst von der Stelle der Vorbedingung bis zur Transition und von da aus zur Stelle der Nachbedingung wandert, müssen noch die Kontrollattribute, die das zeitliche Verhalten modifizieren, überschrieben werden. Dies geschieht über die Attribute `SYNT.onRemoveMove = 1` bzw. `SYNT.onCreateMove = 2`. So wird die zeitliche Hintereinanderschaltung von Animationsereignissen erreicht, die in der Simulation eigentlich zum selben Zeitpunkt auftreten. Somit ist die Animationszeit von der Simulationszeit unabhängig und kann als ein eigenes abstraktes Zeitkonzept betrachtet werden.

AVPs werden an Sprachkonstrukt-knoten angewendet und es können an einem Knoten Muster beliebig kombiniert werden, wenn sich die einzelnen Muster auf unterschiedliche Simulationsmodifikationen beziehen. So können an einem Knoten beispielsweise die Muster `OnCreateBlink`, `OnCreateMove`, `OnRemoveBlur` und `OnRemoveMove` definiert werden, was zu einer Animation *Blinken->Bewegen* bei einer `CREATE` Aktion bzw. *Transparenzänderung->Bewegen* bei einer `REMOVE` Aktion führt.

Die Animationsmuster sind über ihre Änderungsoperation und ihre Animationsart typisiert. Sollen mehrere gleichartig typisierte Muster an einem Knoten gebunden werden, um z. B. je nach Kontext unterschiedliche Animationen auf derselben Simulationsänderungsoperation auszuführen, so führt dies zu Namenskonflikten in den LIDO-Symbolberechnungen.

Zum Beispiel dürfen die Muster `OnCreateBlink`, `OnCreateMove` und `OnRemoveMove` nicht an demselben Knoten angewendet werden, da die Symbolberechnung `OnCreateBlink` doppelt an demgleichen Knoten gebunden werden müsste. Um trotzdem eine Animation der Form *"Blinken->Bewegen->Blinken"* beim Löschen eines Objekts zu erreichen, kann entweder die Grammatikabbildung der Repräsentationsstruktur angepasst und neue Knoten in die Repräsentationsstruktur eingefügt werden oder das Animationsmuster Modul generisch instanziiert werden. Die Zeile

```
patterns.gnrc +instance=_BLINK_ANIMATION :inst
```

aktiviert dabei Elis Mechanismus zur generischen Instanzierung von Modu-

SKK	CREATE	REMOVE	MOVE	CHANGE_VAL	COPY
Klassen	<b>Grow</b> Move Blink Blur Skew Rotate Idle Invi- sible Flash	<b>Shrink</b> Move Blink Blur Skew Rotate Explode Idle Invi- sible Flash	<b>Move</b> Blink Rotate Idle Skew Invi- sible Flash	/	Grow <b>Move</b> Blink Skew <b>Blur</b> Rotate Idle Invi- sible Flash Skew
Aggregation	<b>Morph</b> Idle	<b>Morph</b> Idle	<b>Morph</b> Idle	<b>Morph</b> Idle	<b>Morph</b> Idle
Attribut	/	/	/	TextMorph	/

Abbildung 5.5: Tabelle der AVPs

len. Die AVPs und ihre Kontrollattribute werden dann alle mit einem Namenssuffix versehen und sind wie in diesem Fall z. B. unter dem Namen AVPOnCreateBlink\_BLINK\_ANIMATION verfügbar. Sie können folglich mit den ohne Suffix instanziierten Mustern parallel verwendet werden. So können auch mehrere gleichartige Animationsmuster beliebig kombiniert werden.

Insgesamt stehen eine ganze Reihe von AVPs für alle Simulationsmodifikationen bzw. Knotentypen zur Verfügung - siehe auch Tabelle 5.5.

In fetter Schrift hervorgehoben sind die Standardanimationen, die für jede Simulationsmodifikation automatisch getriggert werden. Alle AVPs sind durch eine Reihe an Kontrollattributen individuell anpassbar. Abbildung 5.6 zeigt die graphischen Darstellungen der Animationsmuster. Das animierte Objekt ist als dunkelgraues Quadrat dargestellt. Die rot gestrichelten Pfeile deuten Bewegung an, durchgezogene rote Pfeile deuten zeitliche Folgezustände in der Animation an.

**Bewegung** Dieses Muster bewegt ein graphisches Objekt von seinem ursprünglichen Ort zu einer beliebigen anderen Position, üblicherweise zum Kontext der Einfügestelle. Das Muster kann durch die Dauer der Bewegung und durch zusätzliche Easing-Funktionen parametrisiert werden (siehe Abb. 5.6c). Durch das Easing kann die Bewegung beim Start und auch beim Ende verlangsamt werden. Durch eine weitere Easing-Funktion bewegt sich das Objekt ähnlich einer Cartoon-Animation über seine Endposition hinaus und bewegt sich erst danach zur endgültigen Position. Eine weitere Variante ist die **Bewegung entlang einer Linie** oder **entlang bestimmter Wegmarken**. Dieses Animationsmuster kann bei der Simulation von Schaltungen

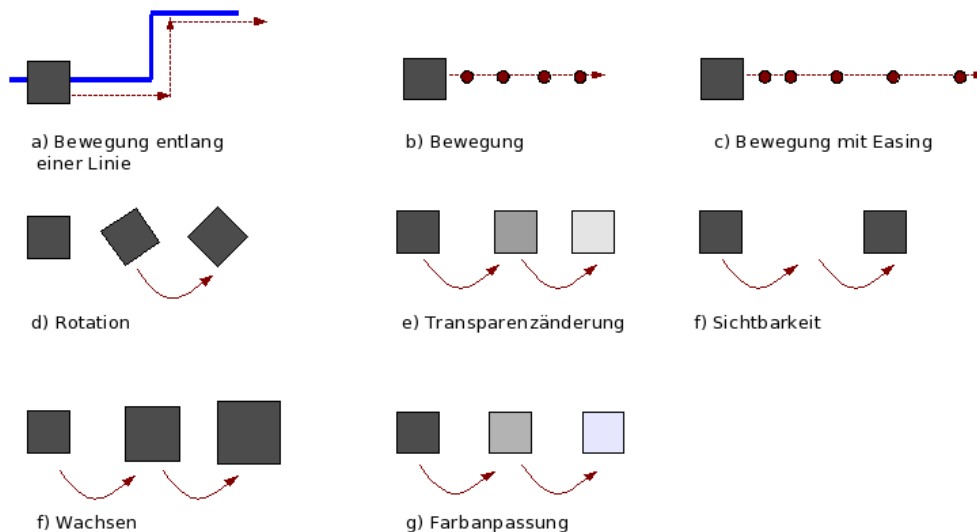


Abbildung 5.6: Animationsmuster

oder Verkehrssimulationen eingesetzt werden. Es kann zusätzlich noch durch einen X/Y-Offset parametrisiert werden. Die Bewegungsrichtung kann invertiert werden.

**Rotation** Hier wird ein graphisches Objekt um seine eigene Achse bewegt. Kontrollattribute sind die Dauer der Rotation, die Geschwindigkeit, der Rotationswinkel sowie die -richtung. Eine besondere Variante ist die **Rotation entlang von Wegmarken**. Sie kommt bei der Verkehrssimulation zum Einsatz. Hier wird z. B. ein Auto, wenn es eine Rechtskurve erreicht um 90 Grad im Uhrzeigersinn rotiert. Die Rotation startet, wenn das Auto über ein anderes Objekt eines bestimmten Typs bewegt wird. Zusätzlich wird der Typ des Objekts gespeichert, den das Auto auf der vorherigen Kachel "überfahren" hat. So können auch Rotationen auf Kurven sowohl in Fahrtrichtung als auch in Gegenrichtung korrekt dargestellt werden.

**Transparenz** Das graphische Objekt ändert seinen Transparenzwert. Kontrollattribute sind der Start-, der Endtransparenzwert sowie die Geschwindigkeit. Bei der COPY-Änderungsoperation ist das Transparenz-Muster Teil der Standardanimation.

**Sichtbarkeit** Hier ändert sich die Sichtbarkeit des graphischen Objekts. Kontrollattribute sind die Frequenz sowie die Dauer.

**Wachsen/Schrumpfen** Das graphische Objekt wird bis zu einem bestimmten Anteil vergrößert bzw. verkleinert. Falls ein Objekt gemorpht wird, so werden ebenfalls diese Animationsmuster angewendet.

**Verdrehung** Das graphische Objekt wird entlang X- oder Y-Achse verdreht. Kontrollattribute sind Start- und Endwerte der Verdrehung jeweils in X- bzw. Y-Richtung.

**Farbanpassung** Dieses Animationsmuster wird implizit auf alle graphischen Primitive mit einer Hintergrundfarbe angewendet. Ändert sich diese, so wird eine flüssige Farbanpassung vorgenommen. Diese erfolgt im HSB-Farbraum, sodass die Anpassung besonders harmonisch wirkt. Kontrollattribute sind hier nicht vorhanden.

**Triggerung** Durch dieses Animationsmuster kann ein Strukturobjekt Animationen an einem anderen Strukturobjekt auslösen. Kontrollattribute sind die Änderungsoperation, bei der eine Triggerung ausgelöst werden soll und das zu triggernde Objekt.

**Priorität** Wenn sich mehrere Animationsobjekte bewegen, kann mit diesem Muster die graphische Priorität gesetzt werden, d. h. es wird definiert, welches Objekt welches andere bei einer Animation überlagert.

**Morphen** Dieses Muster wird auf Objekte angewendet, die ihre Größenausdehnung ändern. Dabei wird jedes graphische Primitiv innerhalb des Objekts angepasst und nicht nur linear vergrößert bzw. verkleinert.

Alle AVPs können nahezu beliebig miteinander kombiniert werden um auch anspruchsvolle Animationen zu erreichen.<sup>2</sup>

### 5.1.5 Berechnung der Animation

Der Algorithmus für die Berechnung der abstrakten Repräsentation aus Abbildung 5.1 muss dafür sorgen, dass alle modifizierten und indirekt geänderten Strukturobjekte korrekt animiert werden. Spezialfälle wie das Bewegen von Teilbäumen, die unterhalb eines gelöschten Teilbaums hängen, müssen erkannt und entsprechende Animationen getriggert werden. Erwähnenswert ist, dass der Algorithmus mit nur einem Durchlauf auf der Baumfragmentstruktur auskommt. Der Durchlauf findet dabei auf dem mit den Simulationsänderungsfunktionen attributierten Baum statt.

---

<sup>2</sup>Mehrere Muster, die dieselben Parameter modifizieren, z. B. die Ausdehnung beim Wachsen oder Schrumpfen, und zur selben Zeit auf demselben graphischen Objekt arbeiten, führen zu fehlerhaften Animationen. Dies wird aus Geschwindigkeitsgründen jedoch nicht überprüft.

Abbildung 5.7 zeigt, dass zur Berechnung der Morph-Funktionen, zunächst die Positionen und Größen der Objekte betrachtet werden, die während der Simulation nicht modifiziert wurden (blaue Pfeile). Der Baum wird dabei solange in der Tiefe mit Morphanimationsfunktionen attribuiert, bis auf einen Knoten gestoßen wird, der bereits durch eine Änderungsoperation markiert ist.

Anschließend werden alle Sprachkonstrukt-knoten betrachtet, die verändert wurden (rote, grüne, hellgrüne Knoten). Dazu werden die durch Änderungsfunktionen markierten Knoten (diese hängen nicht mehr in der Simulationsstruktur; es sind Fragmente) in der Tiefe attribuiert, bis entweder ein Blattknoten erreicht wird oder auf einen markierten Knoten gestoßen wird. Es werden für all diese Knoten die Standardanimationsfunktionen getriggert. Existieren vom Benutzer spezifizierte AVPs für den jeweiligen Knoten, dann werden diese getriggert. Der Wurzelknoten ist dabei die Wurzel der jeweiligen Animationssicht, nicht der des gesamten Modells.

Zu diesem Baumdurchlauf kommt noch ein Baumdurchlauf von LIGA, jeweils zu Simulationsschritt  $S_n$  bzw.  $S_{n+1}$  um die Positionsangaben der graphischen Objekte zu speichern, auf denen die Berechnung der Animationsfunktionen basiert (der zweite Baumdurchlauf kann jeweils wiederverwendet werden).

Einen Fall lässt der Algorithmus bewusst aus: Wird ein Teilbaum bewegt, so werden an den gesamten Baum AVPs vom Typ `MOVE` attribuiert. Wurden innerhalb dieses Teilbaums jedoch auch die Größenverhältnisse der enthaltenen Knoten geändert, z. B. aufgrund einer Löschoption, so müssten auf einige Knoten jedoch AVPs vom Typ `MORPH` angewendet werden. Das Morphen hat jedoch einige andere Parameter als die Bewegung. Insbesondere fehlen Easing-Funktionen, deshalb werden `MOVE`-AVPs konsequent auf den Teilbaum angewendet. Falls dies nicht gewünscht ist, kann der Animationsspezifizierer hier immer noch von Hand das Verhalten überschreiben, indem er das AVP `AVPOnMoveMorph` anwendet.

### 5.1.6 Dynamische Animationsobjekte

Die bisher gezeigten Animationsmuster arbeiten auf Strukturobjekten, d. h. es existieren Simulationsobjekte, die neben ihrer graphischen Darstellung ein Pendant in der Simulationsstruktur haben. Es handelt sich also um eine *strukturelle Animation*, bzw. wenn Attribute angepasst werden, um eine *attribut-orientierte Animation*. In vielen Animationen werden jedoch graphische Objekte animiert, die keinen Repräsentanten in der Struktur besitzen. Sie werden häufig benutzt um auf interessante Stellen in der Animation hinzuweisen (vgl. Kapitel 7.4.1 auf Seite 161). Diese Objekte nenne ich im folgenden "*dynamische Animationsobjekte*". Diese Objekte können in DEViL ebenfalls spezifiziert werden. Dazu ist es ebenfalls möglich Animationsmuster zu benutzen.

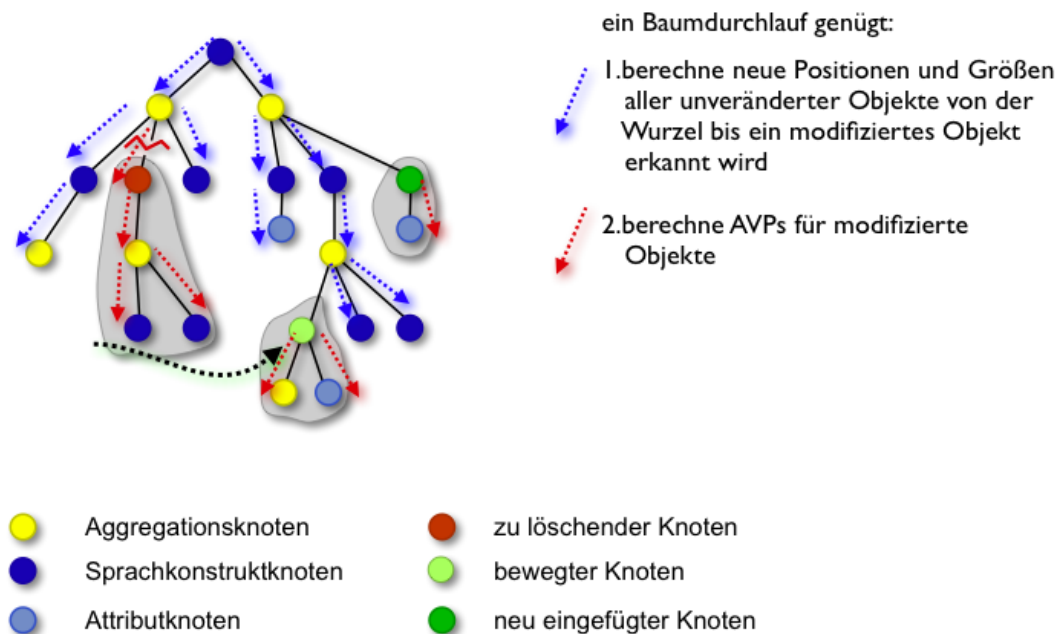


Abbildung 5.7: Berechnung der abstrakten Repräsentation

An einem Sprachkonstrukt-knoten kann ein über die Simulationsänderungsoperationen typisiertes "dynamisches Animationsmuster" angewendet werden. Immer wenn an diesem Knoten die entsprechende Simulationsänderungsoperation ausgelöst wird, wird ein solches dynamisches Animationsobjekt erstellt, das dann mit den bereits eingeführten Animationsmustern und deren Konzepten (zeitliches/räumliches Verhalten) animiert werden kann. Dynamische Objekte können genauso wie generische Zeichnungen definiert und referenziert werden, enthalten jedoch keine Container, da sie keine Unterstrukturen speichern können.

Listing 5.3 zeigt die Spezifikation solcher dynamischer Animationsobjekte. An der Symbolattributierung des Strukturknotens `Load` wird ein dynamisches Animationsobjekt instanziiert (Zeilen 4 - 11). Es wird immer dann erstellt, wenn am selben Knoten eine `COPY`-Simulationsänderungsaktion hervorgerufen wird. Ihm wird die generische Zeichnung `PointDrawing` zugewiesen. Durch das Animationsmuster `AVPMoveAlongLineDynamicObjekt` kann das dynamische Objekt entlang einer Linie, die Drähte in der Datenpfad-Simulation repräsentiert, animiert werden.

### 5.1.7 Statische Animationsobjekte

Es gibt noch eine weitere Art der Animationsobjekte, die genau wie die dynamischen Varianten keinen Repräsentanten in der Simulationsstruktur haben. Sie treten im Zusammenhang mit *Effect Lines* [55] oder beim Logo-Interpreter auf. Es sind graphische Primitive, die durch die Modifikation von Strukturobjekten entstehen können. Ef-

```

1 SYMBOL pageView_Load INHERITS VPTableRow,
2     AVPCreateDynamicObject, AVPMoveAlongLineDynamicObject
3 COMPUTE
4     // instanziiere Zeichnung für dynamisches Objekt
5     SYNT.createDynamicObjectDrawing = ADDRUF(PointDrawing);
6     // zeichne dyn. Objekt an Position des Programmzählers
7     SYNT.createDynamicObjectPosition = POSITION(VLObjectPointer(
8         SELECT(vlList("getElement", THIS.objId, "SMProgramCounter"), eval()), "pcOut");
9     // instanziiere dyn. Objekt nur, wenn am Knoten LOAD eine
10    // COPY Simulationsaktion ausgelöst wird
11    SYNT.createDynamicObjectModificationAction = COPY;
12
13    // Bewege dyn. Objekt zur Position des RAM am Signaleingang ramIn
14    SYNT.moveAlongLineDynamicObjectEndPosition = POSITION(VLObjectPointer(SELECT(
15        vlList("getElement", THIS.objId, "SMRAM"), eval()), "ramInSignal");
16    // Start-Position des dyn. Objekts am Programmzähler am Ausgang pcOut
17    SYNT.moveAlongLineDynamicObjectStartPosition = POSITION(VLObjectPointer(SELECT(
18        vlList("getElement", THIS.objId, "SMProgramCounter"), eval()), "pcOut");
19    // Bewege dyn. Objekt entlang der benannten Drähte
20    SYNT.moveAlongLineDynamicObjectObjects = vlList(SELECT(
21        vlList("getWire", "pcToRam1", THIS.objId), eval()),
22        SELECT(vlList("getWire", "pcToRam2", THIS.objId), eval()));
23 END;

```

Listing 5.3: Spezifikation eines dynamischen Animationsobjekts

fect Lines visualisieren Geschwindigkeit, indem hinter einem sich bewegenden Objekt "Geschwindigkeitslinien" gezeichnet werden. Üblicherweise sind dies Linien, horizontal zur Bewegung. Im Logo-Interpreter werden diese Linien gebraucht um die Spur des Cursors zu verdeutlichen. Diese Art der Animationsobjekte nenne ich statisch, da sie nicht durch Animationsmuster modifiziert werden und statisch auf der Zeichenfläche positioniert werden. Sie können beliebig viele Simulationsschritte überdauern (wie beim Logo-Beispiel) oder auch nur einige wenige Simulationsschritte (wie bei den Effect Lines). In ihrer Form und Farbe können sie aus beliebigen geometrischen Primitiven bestehen. Auch Texteinblendungen um Erklärungen darzustellen sind möglich.

```

1 SYMBOL turtleView_Turtle INHERITS VPSetElement, VPForm,
2     AVPOnMorphCreateStaticObject, AVPCreateStaticLineObject
3 COMPUTE
4     SYNT.drawing = ADDRUF(TurtleDrawing);
5     SYNT.rotate = THIS.pers_direction;
6
7     // in Abhängigkeit des boolean Attributs "pen"
8     // (Stift oben oder Stift unten) wird gezeichnet
9     SYNT.onMorphCreateStaticObjectDraw = THIS.pers_pen;
10    // Positionskoordinaten des Linienprimitivs
11    SYNT.position = VLPointArray(THIS.oPosition, FUTURE_START_POS);
12 END;

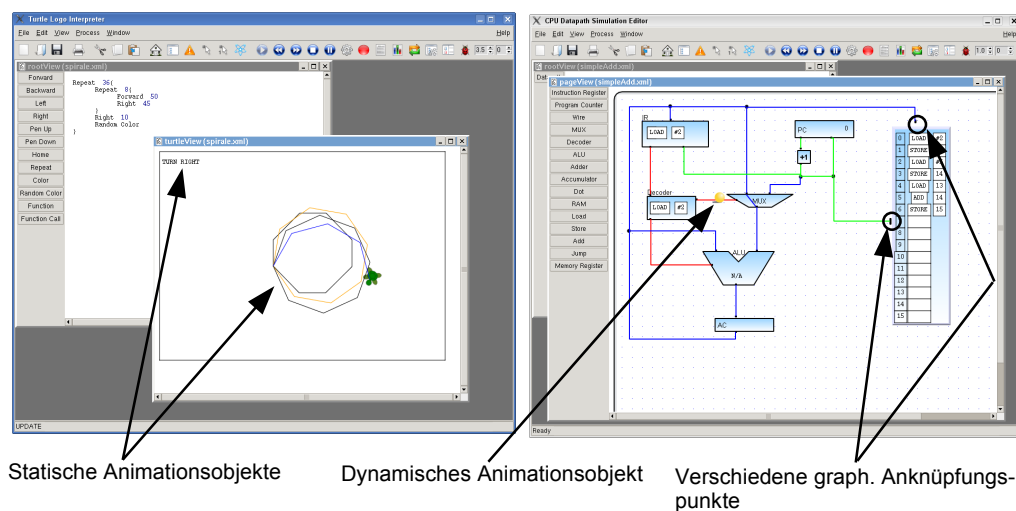
```

Listing 5.4: Spezifikation eines statischen Animationsobjekts

Im DEViL Animationsframework werden sie ebenfalls als eine spezielle Variante

der Animationsmuster betrachtet, d. h. sie können genauso wie AVPs erstellt und parametrisiert werden.

Listing 5.4 zeigt die Deklaration eines statischen Animationsobjekts am Beispiel des Logo Interpreters. Das statische AVP erstellt eine Linie, sobald das zugehörige Strukturobjekt gemorpht wird. Das statische AVP besteht also immer aus zwei Berechnungsrollen: eine für die Simulationsänderungsoperation z. B. `AVPOnMorphCreateStaticObject`, auf die reagiert werden soll und eine für das konkrete Aussehen z. B. `AVPCreateStaticLineObject`. Die gezeigten Kontrollattribute konfigurieren das statische AVP. Mit dem Makro `FUTURE_START_POS` kann auf die Position des Animationsobjekts nach einem Simulationsschritt zugegriffen werden.



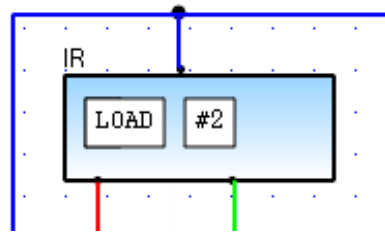
**Abbildung 5.8:** Beispiele statischer und dynamischer Animationsobjekte.

Abbildung 5.8 zeigt nochmals beide Varianten von Animationsobjekten: die statischen Animationsobjekte, die die Spur des Cursors in der Logo-Simulation darstellen und die Signalimpulse im Datenpfad-Editor.

### 5.1.8 Graphische Anknüpfungspunkte

Wenn graphische Objekte bewegt werden, stehen häufig die Start- bzw. Endpositionen fest. Es kommt jedoch auch vor, dass sich Objekte entlang von Wegmarken oder über bestimmte andere graphische Objekte hinwegbewegen sollen. Im Beispiel der Datenpfad-Animation tritt dieses Problem an den Ausgängen von CPU-Elementen auf (vgl. Abbildung 5.9). Das Instruktionsregister hat in diesem Beispiel zwei Ausgänge. Man kann zwar die Bewegung des Instruktionsobjekts entlang des Drahtes angeben, es ist jedoch unklar, welcher Ausgang benutzt werden soll. Um dieses





**Abbildung 5.9:** Ausschnitt aus einem generierten Editor. Positionierungsproblem: welcher Ausgang des Instruktionsregisters soll genutzt werden?

Problem zu umgehen kann man so genannte *graphische Anknüpfungspunkte* definieren. Dazu werden in der generischen Zeichnung des Instruktions-Registers so genannte *“Crosslines”* eingefügt. Sie lassen sich relativ zum Sprachkonstrukt positionieren, sind benannt und können in der Spezifikation der Animationsmuster als Positionsangaben benutzt werden. Zur Laufzeit der Animation wird dann die relative Position im Sprachkonstrukt berechnet und als Wegmarke verwendet. Sie bilden quasi Anknüpfungspunkte für die Animation. In der generischen Zeichnung aus Abbildung 5.10 werden zwei Crosslines definiert, jeweils ein benanntes Crossline für jeden Ausgang.

An der Anwendungsstelle eines Animationsmusters kann der Crossline-Name referenziert werden:

```
SYNT.createDynamicObjectPosition =
POSITION_OBJECT_CROSSLINE(Load, "out1");
```

Das Makro `POSITION_OBJECT_CROSSLINE` berechnet dann automatisch zur Laufzeit die korrekte Position der `out1` Crossline-Definition.

Somit ist es auch möglich innerhalb von generischen Zeichnungen Pfade zu beschreiben. Dies kommt in Editoren vor, in denen basierend auf einem Hintergrundbild Elemente bewegt werden. Dieses Verfahren wurde innerhalb der Monitor Simulation benutzt (vgl. Abschnitt 7.4.1 auf Seite 163).

### 5.1.9 Informationsverlust zwischen Simulation und Animation

Durch die bewusst schmal gewählte Schnittstelle zwischen Simulation und Animation - den Änderungsoperationen - entsteht ein Informationsverlust. Die Ursache oder das Umfeld von Änderungsoperationen können nicht mehr in der Animation abgefragt werden. Ein ähnliches Problem existiert in herkömmlichen Programmiersprachen. Wenn eine Methode aufgerufen wird, kann nicht ohne weiteres herausgefunden werden, was die Ursache war. In einigen Skriptsprachen wie Tcl [80] kann man mittels spezieller Konstrukte auf den dynamischen Vorgänger zugreifen und Informationen aus dessen Umgebung abrufen.

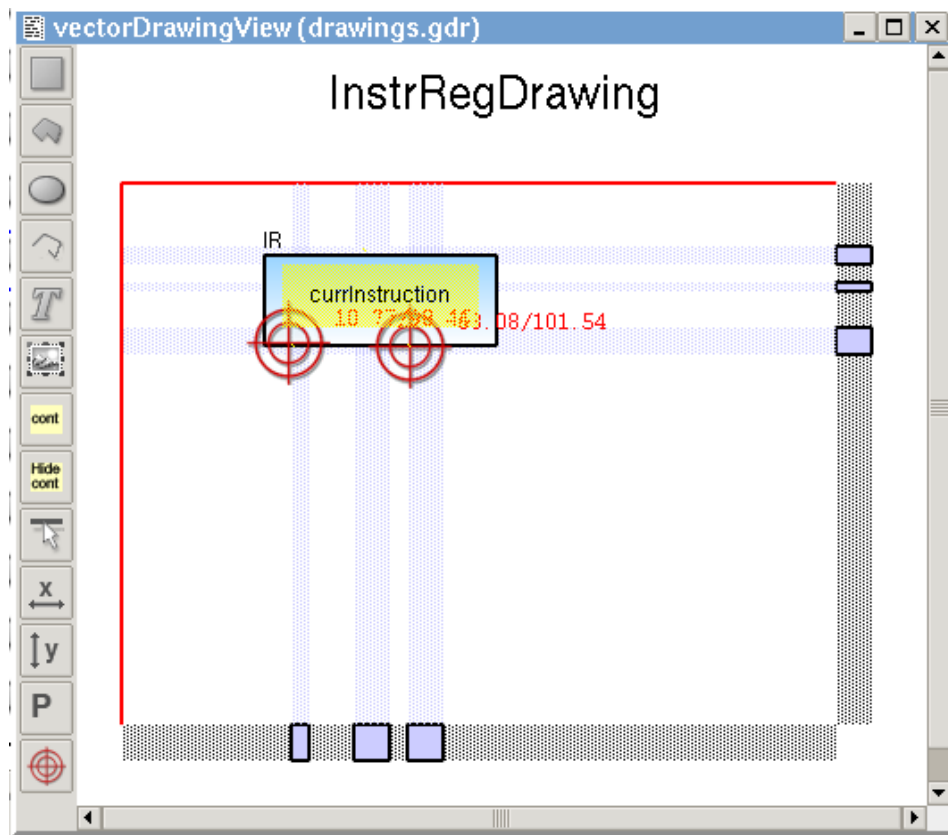


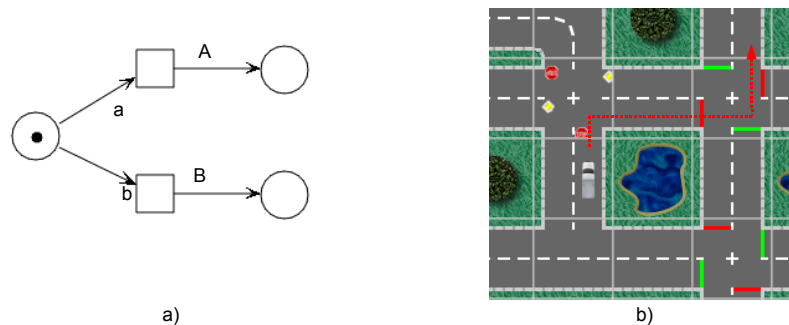
Abbildung 5.10: Positionierungsproblem: Definition von Crosslines zur dynamischen Positionierung

```

1  EVENTS {
2    preTransitionFire(Transition transition) {
3      FOREACH place IN transition.incomingPlaces {
4        Token token = REMOVE(place.tokens, FIRST);
5      }
6    }
7
8    postTransitionFire(Transition transition) {
9      FOREACH place IN transition.outgoingPlaces {
10       Token token = INSERT(place.tokens, INSTANCEOF [Token], FIRST);
11     }
12   }
13 }

```

Listing 5.5: Informationsverlust zwischen Simulation und Animation in einer Petri-Netz Spezifikation



**Abbildung 5.11:** Informationsverlust in der Simulation

Die DSIM-Spezifikation in Listing 5.5 zeigt das Problem in der Simulationskomponente der Petri-Netz Simulation: In der Vorbedingung wird eine REMOVE-Änderungsoperation ausgelöst, in der Nachbedingung eine INSERT-Operation. Diese Informationen werden an die Animationskomponente übergeben. Dort kann dann nicht mehr ermittelt werden, welche Transition gefeuert hat und somit der Verursacher der Aktion gewesen ist. Dies ist jedoch notwendig um eine Animation zu schaffen, die die Marken zur entsprechenden Transition bewegt. Abbildung 5.11a verdeutlicht das Problem visuell. Die Marke wird in der Vorbedingung gelöscht und zwei neue Marken werden in der Nachbedingung generiert. Für die Animation stellt sich die Frage, über welche Transition die Marke bewegt werden soll.

Die in DEViL gewählte Lösung ist die so genannte *Verursacherliste*. Alle Sprachkonstruktinstanzen, die als aktueller Parameter an einem Ereignis beteiligt sind, das eine Änderungsoperation auf einem Sprachkonstrukt auslöst, werden protokolliert und in die Verursacherliste geschrieben. Im obigen Beispiel bedeutet dies, dass das Transition-Objekt `transition` als erster Parameter in die Verursacherliste für die Änderungsoperation REMOVE bzw. INSERT des Tokens geschrieben wird.

In der Animationsspezifikation kann auf diese Liste über ein Makro zugegriffen werden; das Resultat ist eine Positionsangabe. Listing 5.6 zeigt nochmals den Zugriff auf die Verursacherliste. Mit `SYNT.onRemoveMoveEndPosition = POSITION_INITIATOR(1);` wird am Strukturknoten `Token` auf das erste Element in der Verursacherliste zugegriffen, d. h. es wird der erste aktuelle Parameter des Ereignisses `preTransitionFire` aus Listing 5.5 gelesen. Das Ergebnis ist ein Transition Objektknoten, dessen Position an das Muster `AVPOnRemoveMove` übergeben wird

Neben der Verursacherliste kennt ein modifiziertes Objekt außerdem das Ereignis, das die Modifikation ausgelöst hat. Dies ist sinnvoll, weil Simulationsänderungsoperationen desselben Typs natürlich auch in unterschiedlichen Ereignissen auftreten

```

1 SYMBOL rootView_Token INHERITS VPForm, VPSetElement,
2                               AVPOnRemoveMove
3 COMPUTE
4   SYNT.drawing= ADDRUF(TokenDrawing);
5
6   SYNT.onRemoveMoveEndPosition = POSITION(1);
7   SYNT.onRemoveMoveAnimationTime = 1;
8 END;
```

**Listing 5.6:** Zugriff auf die Verursacherliste mit dem POSITION-Makro, das das erste Element der Verursacherliste auswählt.

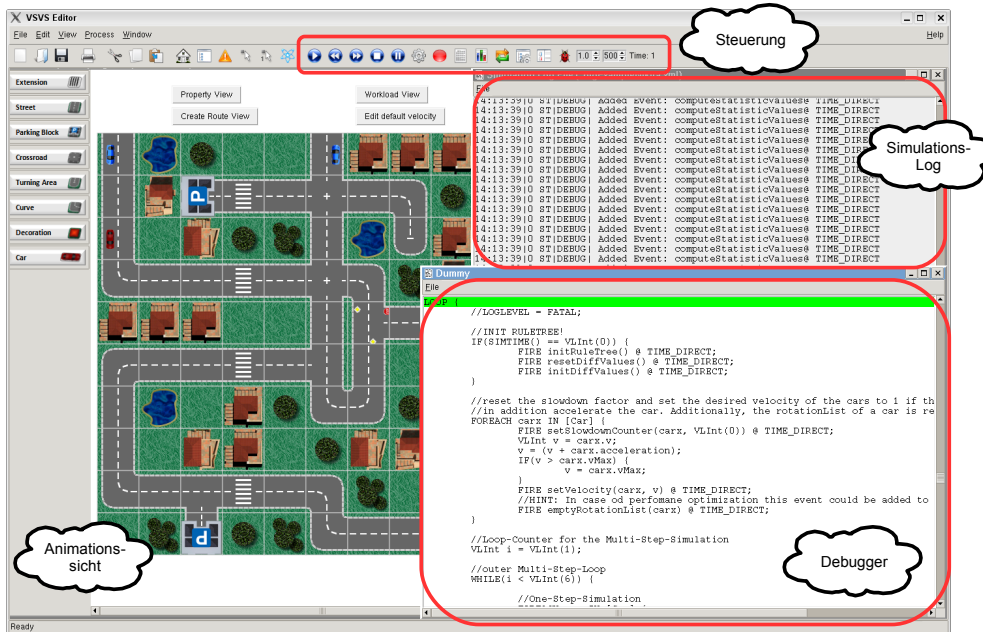
```

1 SYMBOL pageView_Value INHERITS VPSingletonForm, VPSimpleListElement,
2                               AVPCreateDynamicObject, AVPMoveAlongLineDynamicObject
3 COMPUTE
4   SYNT.createDynamicObjectModificationAction = CREATE;
5   SYNT.createDynamicObjectAnimate = IF(EQ(VLString(INITIATOR_EVENT(CREATE))),
6                                       String("createValueInAccumulator()"), 0, 1);
7   ...
8 END;
```

**Listing 5.7:** Zugriff auf das auslösende Ereignis innerhalb einer Animationspezifikation

können. Listing 5.7 zeigt die Anwendung des INITIATOR\_EVENT-Makros. Das dynamische Animationsobjekt wird nur animiert, falls das auslösende Ereignis für die CREATE-Aktion am Strukturknoten Value createValueInAccumulator() war. Wurde eine Value Knoteninstanz innerhalb eines anderen Ereignisses instanziiert, so wird keine Animation ausgelöst.

Eine weitere Variante des Informationsverlustes zwischen Simulation und Animation lässt sich im Struktureditor für die Verkehrssimulation beobachten. Dort wird ein realistisches Fahrverhalten mit Hilfe des *Nagel-Schreckenberg*-Algorithmus [74] simuliert. Der Simulationsschritt für ein Autos wird dabei wiederum in Mikroschritte unterteilt. Fährt das Auto mit Geschwindigkeit fünf, so kann es maximal fünf Kacheln pro Simulationsschritt überfahren. In jedem dieser fünf Mikroschritte muss jedoch geprüft werden, ob das Auto auf die entsprechende Kachel bewegt werden darf, ob die Bewegung also den Verkehrsregeln entspricht und ob sich keine anderen Autos im Weg befinden. Nach einem Simulationsschritt steht das Auto also theoretisch fünf Kacheln von der Ausgangsposition entfernt. Falls das Auto in der Zwischenzeit Kurven befahren hat, wie in Abbildung 5.11b (rot gestrichelter Pfeil) zu sehen ist, kennt die Animationskomponente nur Start- und Endposition. Somit würde eine geradlinige Bewegung, die beide Positionen direkt verbindet, generiert. Dies entspricht natürlich nicht einer realitätsnahen Animation. Deshalb speichert die Simulationskomponente zusätzlich eine Einfügehistorie für jedes Objekt und jeden



**Abbildung 5.12:** Generierter Struktureditor mit Simulations- und Animationsunterstützung

Simulationsschritt, die in diesem Fall aus den benutzten Kacheln zwischen Start und Ziel besteht. Über ein entsprechendes Makro

```
SYNT.onMoveMoveAlongLineObjects =
POSITION_INSERTION_HISTORY (THIS.objId) ;
```

am zugehörigen Auto-Knoten kann auf eine Liste dieser Einfügepositionen zugegriffen werden. Entsprechend wird dann diese Liste in den AVPs (hier das AVPOnMoveAlongLine) benutzt.

## 5.2 Eigenschaften generierter Umgebungen

In Abbildung 5.12 ist ein generierter Struktureditor mit Simulations- und Animationsunterstützung zu sehen. Alle von DEVIL generierten Editoren mit Simulationsunterstützung enthalten einige zusätzliche Funktionen um Simulation und Animation zu steuern. Des Weiteren enthalten die Editoren zusätzliche Sichten, die Meta-Informationen zur Simulation anzeigen. Dazu gehören eine Sicht für Log-Ausgaben, eine Simulationsstruktur-Baumansicht, die ähnlich wie die bekannte Strukturbauansicht das komplette Modell mit erweiterten Attributen und Werten anzeigt. Eine Debugger-Sicht, die DSIM Code enthält, sowie eine Zeitschienenansicht, die Auskunft über gefeuerte Ereignisse und angewendete Animationsmuster gibt. Die Simu-

lation und Animation des generierten Editors kann durch die Werkzeugleiste gesteuert werden. Im folgenden werden die einzelnen Werkzeugleistenknöpfe zur Steuerung erläutert (von links nach rechts).

**Steuerung der Simulation** Über die ersten fünf Knöpfe in der Werkzeugleiste wird die Simulation gesteuert. Es ist ein Starten oder Stoppen sowie ein schrittweises Ausführen der Simulation möglich. Durch den Pause-Knopf kann eine evtl. gerade stattfindende Animation pausiert werden.

**Simulationsmodus** Über den Modus-Knopf wird die Animation, falls vorhanden, abgeschaltet. Es werden dann nur die diskreten Schritte der Simulation angezeigt. Dieser Modus erlaubt die Ausführung von langlaufenden Simulationen.

**Aufnahme** Über diesen Knopf wird die Animation in einen Mpeg-Film exportiert. Zusätzlich liegen SVG Bilder der einzelnen Animationschritte vor.

**Log-Ausgabe** Hier werden alle Ausgaben, seien es interne durch den Simulator oder durch den Sprachentwerfer explizit vorgenommen Ausgaben gelogged. Die Ausgaben können als Textdatei exportiert werden.

**Statistik** Ruft das Statistikfenster für Auslastungsgraphen auf (wird im späteren Kapitel "Analyse" genauer erläutert).

**CPH** Dieser Knopf steuert den Change-Propagation-Handler (CPH). Der CPH schreibt Änderungen in der Simulationsstruktur in das semantische Modell der Sprache zurück, üblicherweise nach jedem Simulationsschritt. Dieses Intervall kann hier geändert werden. Dies unterstützt auch wieder langlaufende Simulationen.

**Simulationsstruktur** Der Knopf öffnet eine Baumansicht auf die Simulationsstruktur. Diese kann genauso wie das semantische Modell editiert werden. Es werden aktuelle Wertebelegungen von Zufallszahlen und die Mengen von Pfadausdrücken angezeigt.

**Ereignis-Log** Der Knopf öffnet eine Ansicht, in der alle Ereignisse und deren Modifikationen an Strukturobjekten protokolliert werden. Die Darstellung basiert auf dem externem Werkzeug *ZVTM / ZGRViewer* [87] und kann im Detail im Analyse-Kapitel dieser Arbeit betrachtet werden.

**Debugger** Der Knopf aktiviert den Debugger für den Simulationscode. Die Debugger-Ansicht wird geöffnet und der DSIM-Code kann schrittweise ausgeführt werden.

**Animationsfaktor** Dieser Knopf manipuliert die abstrakte Animationszeit. Ein Wert größer als eins führt zu einer Zunahme der Geschwindigkeit der Animation. Ein Wert kleiner als eins führt zu einer Verlangsamung der Animation.

**Simulations-Idle-Zeit** Der Knopf ändert die Wartezeit, wenn in einem Simulationsschritt kein Ereignis ausgelöst wird. Es wird dann eine Stoppuhr eingeblendet. Dies ist besonders bei Warteschlangensimulationen interessant.

**Simulationszeit** Diese Anzeige gibt die aktuelle Simulationszeit wieder - nicht Animationszeit!

Eine Besonderheit in generierten Editoren ist, dass die Sichten auch während einer Animation Benutzerinteraktion zulassen, d. h. die Editoren erlauben eine *“in-place”*-Modelländerung wie in [88]. Die generierten Editoren legen dabei ein optimistisches Verhalten zu Grunde; sie erlauben dem Anwender die gesamte Modellinstanz zu ändern. So kann der Anwender eine Simulation natürlich unmöglich machen, indem er z. B. Strukturelemente löscht, die im späteren Verlauf noch benötigt werden. Ich halte aber den Ansatz *“der Anwender weiß was er tut”* im Grunde für richtig, da er es erlaubt, noch sehr flexibel in die Simulation einzugreifen. Sollte dies doch zu Problemen führen, lässt sich die Benutzerinteraktion während der Simulation sehr einfach abschalten.

DiaMeta verfolgt hier den Ansatz, die Menge der möglichen Benutzerinteraktionen einzuschränken, indem die aktuellen Folgezustände berechnet werden. Dies führt aber insbesondere bei zufallsbasierten Simulationen nicht allzu weit, denn es kann natürlich nicht berechnet werden, ob ein Sprachkonstrukt in den folgenden Schritten noch benötigt wird oder ob ein Attributwert verändert werden darf.

Die von DEViL generierten Editoren kann man auch ohne graphische Bedienoberfläche starten. So ist es möglich, die Editoren per Batchskript mehrfach hintereinander aufzurufen. Ausgaben können dann in eine Datei umgeleitet werden. So können auch zufallsbasierte Simulationen leicht mehrfach gestartet werden.

Außerdem enthalten von DEViL generierte Editoren eine Zooming-Funktionalität: Während der Animation kann die Zeichenfläche beliebig vergrößert oder verkleinert werden.

### 5.3 Diskussion

Durch die AVPs entstehen dem Sprachentwickler diverse Vorteile gegenüber einem imperativen Ansatz. Die lineare graphische Interpolation wird automatisch generiert, d. h. es entsteht einem Simulationsentwickler kein zusätzlicher Implementierungsaufwand. Die Animation kann so jederzeit zusätzlich erstellt werden. Wie sich in der Evaluation zeigen wird, ist die Standardanimation in vielen Fällen bereits ausreichend für eine ansprechende Animation. Somit ist der vorgestellte Ansatz insbesondere für das *Rapid-Prototyping* geeignet, in dem sehr schnell Editoren mit eingeschränkter Funktionalität und mit unterschiedlichem Aussehen erstellt werden müssen.

Ist die automatisch generierte Animation nicht ausreichend, so können die Animationsmuster benutzt werden um eine passende Animation herzuleiten. Der deklarative Ansatz spezifiziert dabei, *was* passieren soll, und nicht, *wie* eine Animation ablaufen soll. Hinzu kommt, dass die bewährten Spezifikationsmechanismen der visuellen Muster adaptiert und wiederverwendet werden. Ein eventueller Einarbeitsaufwand ist somit sehr gering. Animationsmuster lassen sich nahezu beliebig miteinander kombinieren ("composable-Konzept" aus Kapitel 2.5) und definieren ein eigenes Zeitkonzept. Anspruchsvolle Animationssequenzen lassen sich somit sehr einfach erstellen. Ebenso wie die visuellen Muster lassen sich die Animationsmuster durch Kontrollattribute leicht anpassen. Elemente der Cartoon-Animation wie z. B. *Easing* oder komplexe Bewegungsmuster wurden dazu übernommen.

Die Anwendung der Animationsmuster auf Objektebene und der Zugriff auf die Simulationsstruktur sowie Konzepte wie die Verursacherliste ermöglichen eine kontextabhängige Animation.

Die schmale Schnittstelle zwischen Simulation und Animation durch die eingeführten Änderungsoperationen erlaubt es, eine formal korrekte Animation herzuleiten. Es können keine Animationsereignisse übersehen werden. Dieser Aspekt wird häufig in der Literatur als wichtig angesehen, da es sich hier um eine Art Abbildung einer Simulationsstruktur auf eine Animationsstruktur handelt. Ist diese Abbildung beweisbar richtig, können Fehlerquellen auf die Modellierung der Simulation zurückgeführt werden. Zudem lässt sich somit eine Animation zum Debuggen ihrer Simulation benutzen. Des Weiteren führt diese Separierung zu einer räumlichen Trennung beider Spezifikationsaspekte. Dies erlaubt eine einfachere Entwicklung im Team.



## 5.4 Verwandte Arbeiten

In GenGed [31] wird eine Abbildung mittels Graphtransformation spezifiziert, um eine Animation zu generieren. Diese Vorgehensweise ist ebenfalls deklarativ und sehr formal. Komplexe Animationen, die von einer normalen Bewegung abweichen und eine "weiche" Animation können nicht spezifiziert werden.

In [122] wird das Alma-System zur Programm- und Algorithmenanimation eingesetzt. Dabei wird bei jeder Strukturänderung, was der Anwendung einer Regel entspricht, die Visualisierung angestoßen. Diese Änderungen entsprechen in etwa den Simulationsmodifikationen in DEViL.

Das Amulet Framework (C++ Bibliothek) [72] erlaubt es, auf graphische Objekte einer GUI Animationen anzuwenden. Diese werden in so genannte *Slots* eingeordnet. Dabei wird durch das Framework der Status des Objekts abgefragt. Ändert sich dieser, so werden Animationsaktionen ausgelöst, die deklarativ spezifiziert werden können. Eine Änderung einer Integer-Variablen von 0 auf 100 kann zu einer Animation führen, die langsam bis auf 100 hochzählt. Amulet unterscheidet viele Zustände von Objekten, so zum Beispiel auch, ob ein Objekt sichtbar ist oder nicht. In einen Slot können beliebig viele Animationen gepackt werden. Dieser Ansatz ähnelt dem Konzept der AVPs, die ebenfalls in einer ähnlichen Form spezifiziert werden und genauso kombinierbar sind.

Pavane [98] ist ein Programmvisualisierungssystem, das für Zustandsänderungen ebenfalls graphische Interpolation benutzt.

Bei den Generatorsystemen setzt DiaMeta auf Simulationsseite ebenfalls Graphtransformation ein. Um eine Animation zu erreichen, wird der Graph um Animationskanten erweitert, die Zustandsübergänge auf die Zeit abbilden können. Somit lässt sich jeder Animationszustand auf einen Zustand des Graphtransformationssystems abbilden. DiaMeta kann aus jedem Zustand eine gültige Menge an Folgezuständen berechnen. Dadurch können Benutzerinteraktionen, die ein nicht korrektes Modell erzeugen würden, leicht herausgefiltert werden. Von DEViL generierte Editoren sind hier optimistisch, d. h. auch nicht-korrekte Simulationsstrukturen können während der Simulation oder Animation durch Benutzerinteraktion entstehen. Dies führt jedoch nicht zu Fehlern in der Simulation. Die generierten Editoren sind hier robust genug um solche Fälle zu kompensieren. Das optimistische Änderungsverhalten von DEViL ist somit nicht unbedingt ein Nachteil.

Die Abbildung bzw. die Distanz zwischen Simulation und Animation in DiaMeta kann durch die Erweiterung der konkreten Syntax angepasst werden. In DEViL kann die Abbildung ggf. durch statische bzw. dynamische Animationsobjekte sowie durch Strukturkopplung angepasst werden. Ein Pendant zu Animationsmustern besteht in DiaMeta nicht. Spezielle Animationen, wie nicht-lineare Bewegungen von Strukturobjekten (z. B. das Easing), müssen in DiaMeta (noch) ausprogrammiert werden.

Ein anderer Ansatz ist die *reactive animation* von Harel [30]. Simulation wird hier durch das Ausführen von UML-Diagrammen mit Hilfe von Rhapsody realisiert. Eine Animation wird durch die Verknüpfung mit einem reinen Animationstool wie Macromedia Flash oder Director [56] realisiert. Die Wahl von TCP/IP als allgemeines Übertragungsprotokoll zwischen Simulation und Animation kann dabei ähnlich wie in DEViL gesehen werden. Auch hier ist die Schnittstelle eng und eine Animation kann theoretisch auch entkoppelt von der Simulation betrachtet werden. Die Abbildung von Simulation und Animation ist jedoch bei Harel nicht so formal und auch hier muss die Animation ausprogrammiert werden.

# 6 Analyse

## Inhalt

---

6.1 Analyse der Simulationsspezifikation . . . . .	142
6.2 Analyse des visuellen Programms . . . . .	145
6.3 Aspektorientierte Analyse . . . . .	146

---

Die Simulation und Animation visueller Sprachen wurde eingeführt um zu mehr Programmverständnis zu führen. Die mentale Kluft zwischen Programmdarstellung und Programmausführung sollte verringert werden. Anspruchsvolle Animationen gestatten es dem Anwender, komplexes und evtl. paralleles Programmgeschehen anschaulich darzustellen. Eine große Auswahl an frei kombinierbaren Animationsmustern, statische und dynamische Animationsobjekte, eine beliebige Manipulation der Simulationszeit sowie eine sehr freie Abbildung von Simulation und Animation tragen dazu bei. Die Animation wird somit das ultimative Mittel der Analyse. Trotzdem sind einige Aspekte der Analyse bisher nicht betrachtet worden. Zum einen möchten Sprachanwender quantitative und qualitative Aussagen über Teile des Programms treffen. Dies können z. B. in einer Verkehrssimulation Aussagen über Auslastungen von Straßenabschnitten sein oder noch allgemeinere Aussagen wie "ein Kreisverkehr ist hinsichtlich des Verkehrsflusses besser als eine Kreuzung".

Zum anderen möchte man auch den Entwickler einer Simulation bzw. Animation unterstützen. Die Animation ist zum Zweck der Spezifikation einer Simulation in von DEViL generierten Editoren sicherlich ein gutes Analysemittel, da die Animation durch die enge Schnittstelle eine formale Abbildung der Simulation ist. Die automatisch abgeleitete Animation spiegelt also das Simulationsgeschehen exakt wieder. In den folgenden Abschnitten sollen einige, in das DEViL-System integrierte Hilfsmittel vorgestellt werden, die helfen Simulation bzw. Animation korrekt zu spezifizieren und auch analytische Aussagen zur Sprachinstanz zu treffen.

### 6.1 Analyse der Simulationsspezifikation

Auf Seiten der Spezifikation von Simulation und Animation sind Ereignisgraphen, der Debugger und Animationsmetainformationen als Hilfsmittel integriert. Abbildung 6.1 zeigt einen Ausschnitt eines Ereignisgraphen. Hier sind alle gefeuerten Ereignisse (blau hinterlegte Knoten) auf einer Zeitskala protokolliert. Zusätzlich werden veränderte Strukturobjekte aufgenommen (grau hinterlegte Knoten) und durch Pfeile verdeutlicht, welches Ereignis eine Änderungsoperation auf welchem Strukturobjekt vorgenommen hat. Rote Pfeile deuten auf eine Löschoption, grüne Pfeile auf eine Erstellungsoperation und violette auf eine Kopie hin. Zusätzlich wird dargestellt, welche Animationsfunktionen aus den Simulationsänderungsoperationen für welches Objekt ausgelöst wurden (türkis hinterlegte Knoten). Dieses Analysemittel ist besonders für den Sprachentwerfer interessant. Die zeitliche Korrespondenz zwischen Änderungsoperation und auslösendem Ereignis ist beim Sprachentwurf wichtig. Häufig kommt es vor, dass ein ganz anderes Ereignis Verursacher einer Operation war.

Ein weiteres wichtiges Mittel für den Entwurf einer korrekten Simulation ist der integrierte DSIM-Debugger. Wie in anderen Debuggern auch, können Halte-

## 6.1. ANALYSE DER SIMULATIONSSPEZIFIKATION

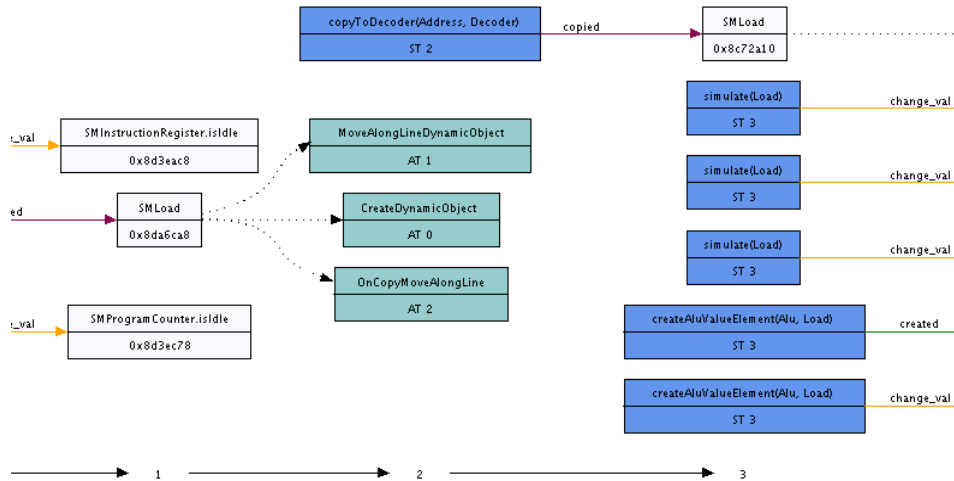


Abbildung 6.1: Ausschnitt aus einem Ereignisgraphen

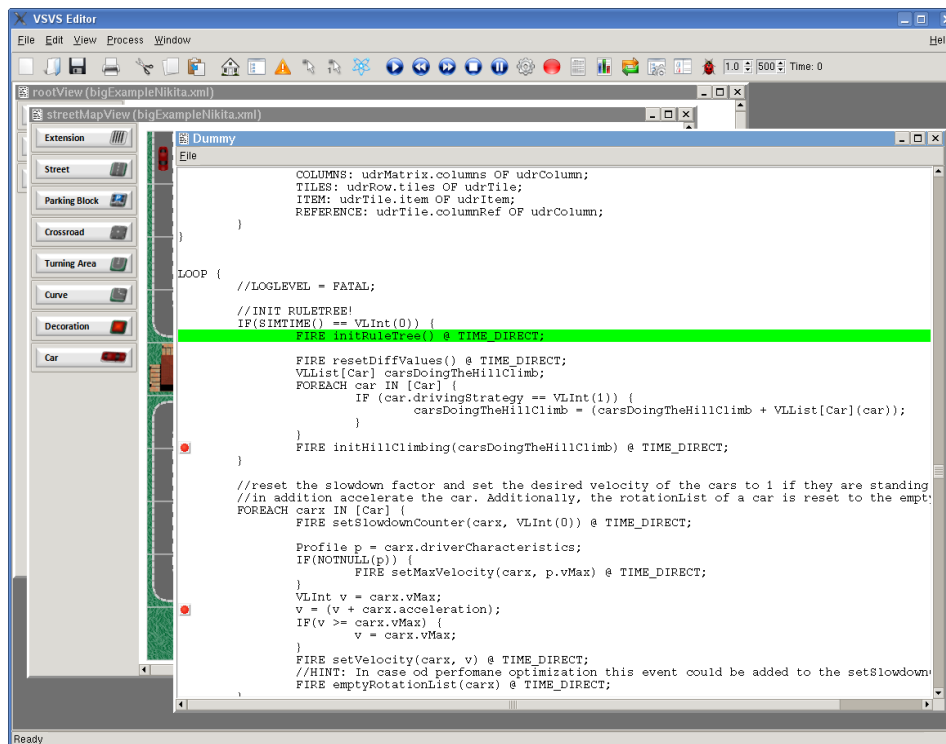
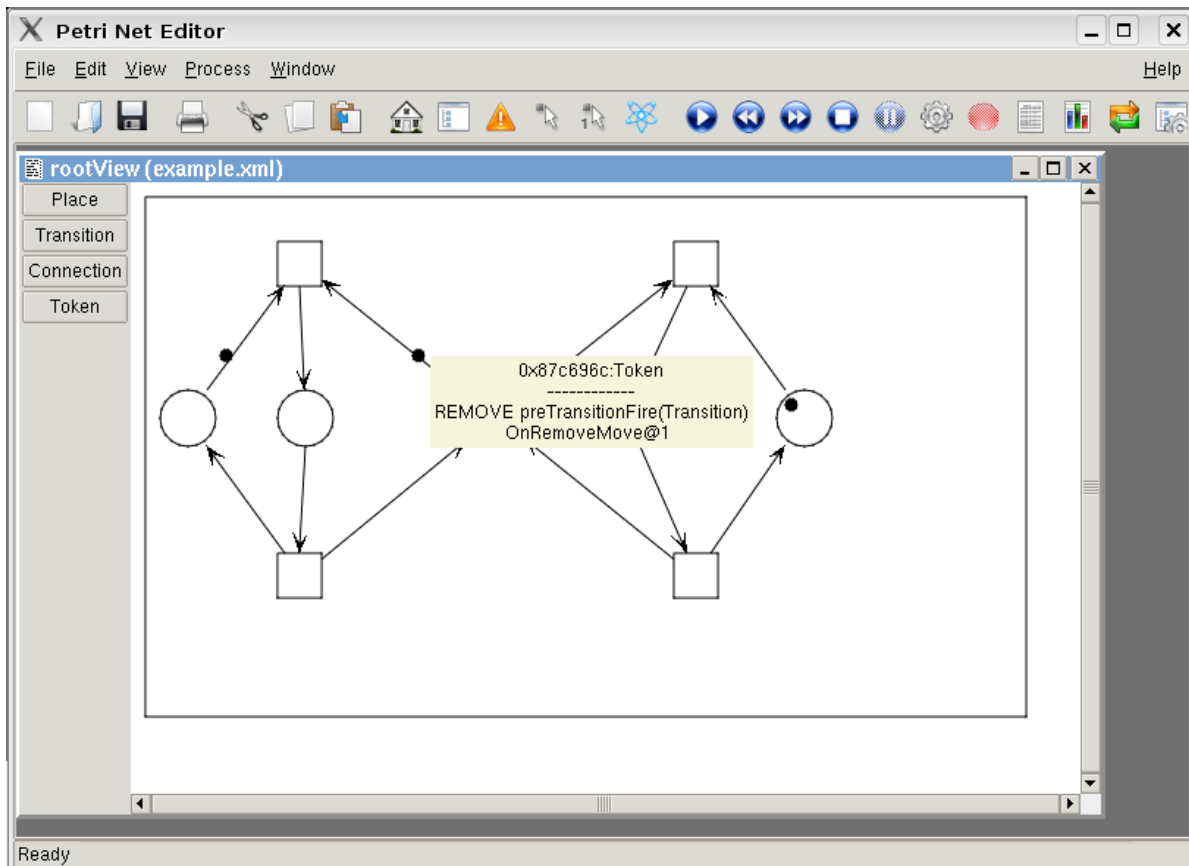


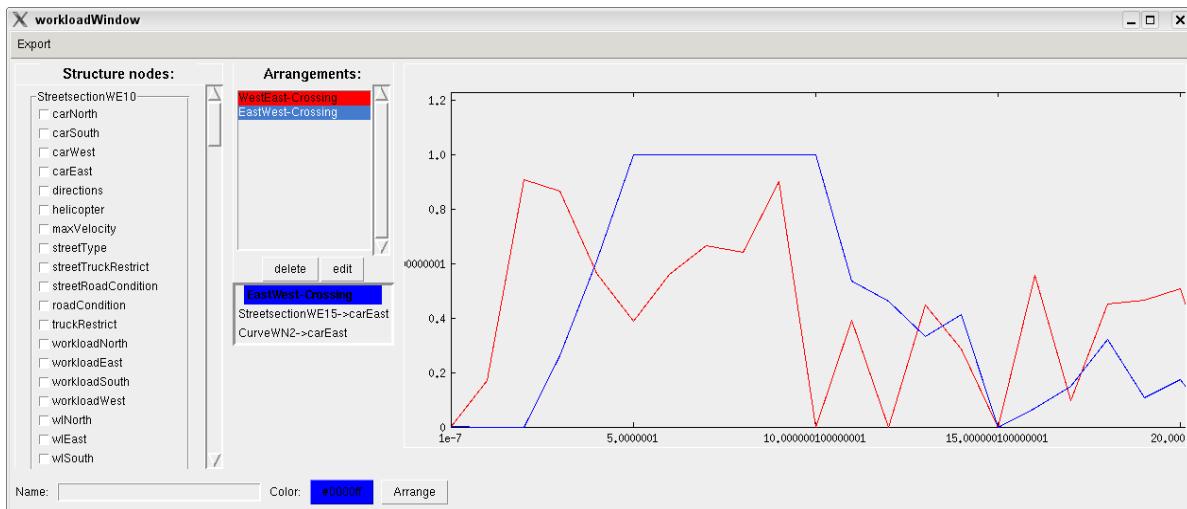
Abbildung 6.2: Der integrierte DSIM Debugger mit zwei aktiven Haltepunkten



**Abbildung 6.3:** Animationssicht mit Metainformationen zu ausgelösten (Animations-)Ereignissen eines Objekts

punkte gesetzt und so der Kontrollfluss beobachtet werden. Ein interessanter Aspekt ist, dass der Debugger auch als eine alternative Sicht, quasi als Algorithmenanimationssicht, betrachtet werden kann. Dies ist möglich, da DSIM sehr einfach gehalten ist und der Simulationserwerber sich voll auf die algorithmische Implementierung des Verhaltens konzentrieren kann. Implementierungsdetails bleiben verborgen. Da DSIM Ähnlichkeiten zu Pseudo-Code hat, kann der Debugger auch als Programmvisualisierungswerkzeug betrachtet werden.

Auch direkt in die Animationssicht können Zusatzinformationen eingeblendet werden. In Abbildung 6.3 ist eine Animationssicht zu sehen in der Informationen zu gefeuerten Ereignissen und ausgeführten Animationsmustern eingeblendet werden. Die Informationen werden als Popup-Fenster zum Strukturobjekt angezeigt, über dem sich der Mauszeiger befindet. Im Bild ist dies eine Marke eines Petri-Netzes an dessen Strukturknoten die Ereignisse `preTransitionFire` sowie `postTransitionFire` und das dazugehörigen AVP `OnRemoveMove` ausgelöst wurde.



**Abbildung 6.4:** Auslastungsgraph: Links im Bild eine Liste mit Sprachkonstruktioninstanzen und deren Attributen, in der Bildmitte sind die *Arrangements* zu sehen und rechts der generierte Auslastungsgraph.

Des Weiteren kann die Simulationsstruktur genauso wie die semantische Struktur in einer Baumansicht dargestellt und manipuliert werden. Die Baumansicht zeigt dabei alle erweiterten Attribute sowie die letzten Belegungen der Zufallsvariablen und außerdem eine aktuelle Belegung für die Pfadausdrücke an.

## 6.2 Analyse des visuellen Programms

Für Warteschlangen- oder Verkehrssimulationen ist es interessant zu wissen, wie stark bestimmte Teile einer Sprachinstanz frequentiert sind. Für von DEViL generierte Editoren bedeutet dies, dass untersucht werden muss, wie viele Elemente sich innerhalb eines Aggregationsknotens über einen bestimmten Zeitraum befinden. In einer Waschstraßen Simulation gibt es mindestens zwei Aggregationsknoten, die von Interesse sind: die Warteschlange und die Bedienschlange. DEViLs Analysekomponente erlaubt es, Aggregationsknoten zu beobachten, d. h. während der Simulation werden Auslastungen der beobachteten Knoten ständig protokolliert und in einem Auslastungsgraphen visualisiert.

Die Verkehrssimulation basiert auf einem gekachelten Straßenfeld, d. h. eine Kreuzung besteht aus mindestens neun Kacheln. Will man die Auslastung einer Kreuzung messen, so kann man jedoch nicht nur einfach einen Aggregationsknoten der jeweiligen Richtung beobachten, sondern man muss mehrere Knoten, die quasi die Zufahrt zur Kreuzung bilden, zusammenfassen. DEViL erlaubt dieses Zusammenfassen von Aggregationsknoten als so genannte *Arrangements*. Der Anwender wählt mehrere Aggregationsknoten in der Sprache aus und fasst sie

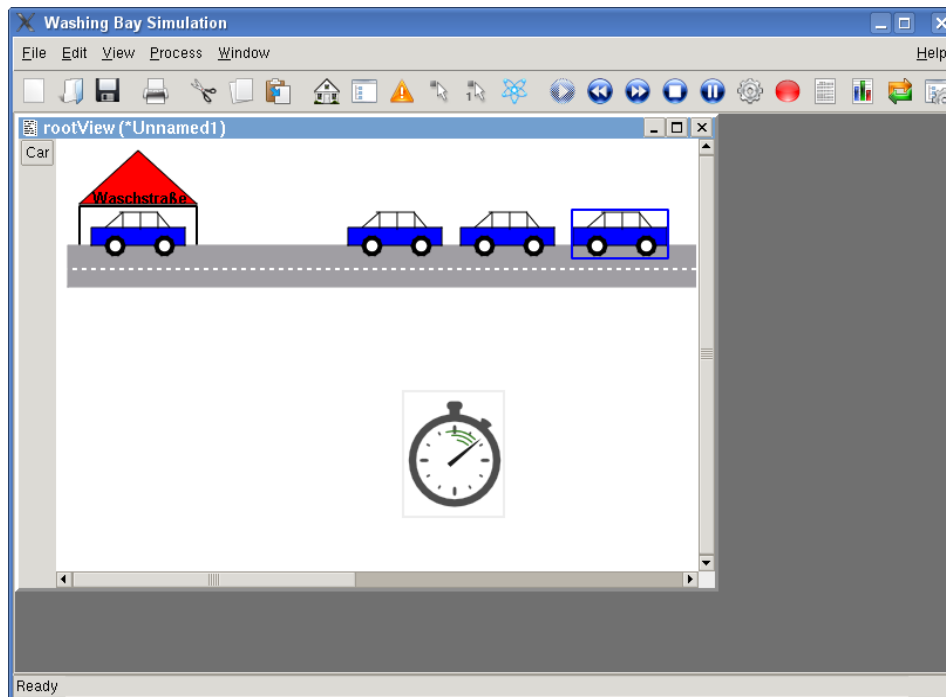


Abbildung 6.5: Darstellung von Leerlaufzeiten in der Simulation

als ein Arrangement zusammen. Intern werden diese Knoten nun als eine Warteschlange betrachtet und die jeweiligen aggregierten Knoten addiert. So lassen sich komfortabel auch für kachelbasierte Sprachen Auslastungsgraphen generieren. In Abbildung 6.4 ist ein solcher Auslastungsgraph zu sehen. Links im Bild ist eine Reihe von Sprachkonstruktioninstanzen mit deren Attributen zu sehen. Der Anwender kann beliebig viele dieser Knoten zu Arrangements zusammenfassen. Im Bild sind die zwei Arrangements *WestEast-Crossing* sowie *EastWest-Crossing* zu sehen. Im Bild ist das blaue Arrangement ausgewählt, das aus den Sub-Attribut Knoten *StreetSectionWE14->carEast* sowie *CurveWN2->carEast* besteht. Beide Knoten werden im Auslastungsgraphen zusammengefasst.

Abbildung 6.5 zeigt, wie Leerlaufzeiten der Simulation verdeutlicht werden. Wird in einem Simulationsschritt kein Ereignis abgearbeitet, so wird im Editor eine Stoppuhr angezeigt. Warteschlangensimulationen spiegeln so, trotz des *“next-event to time advance”*-Ansatzes des Simulators, Simulationszeiten realistisch wieder. So können auch ggf. ungewollte Simulationsleerlaufzeiten erkannt werden.

### 6.3 Aspektorientierte Analyse

Eine Variante der Analysemöglichkeiten ist die Untersuchung der Simulationsstruktur auf bestimmte Eigenschaften. Diese Eigenschaften sind in der Regel über das



```

aspect ElementCounter {
  advice "SMCar" : slice class {
    int counter;
  public:
    int count() const { return counter; }
    void incrementCounter() {counter++;}
  };

  advice execution("% Simulator::simulateNextStep(...)") &&
    that(simulator) : after(Simulator & simulator) {

    std::vector<SMCar*>::iterator iter;
    for (iter = simulator.getCars()->begin(); iter != simulator.getCars()->end(); iter++) {
      SMCar * car = *iter;
      char * role = dynamic_cast<AggregateRelation*>(car->getParent()->getRole());

      if(strcmp(role, "waitingQueue") == 0)
        car->incrementCounter();

      if(car->count() > 10)
        vlShowInformationDialog("Some cars are now waiting
                               a long period of time!", "Information");
    }
  }

  advice construction("SMCar")
    && that(car) : before( SMCar& car ) {
    car.counter = 0;
  }
};

```

Listing 6.1: Ein Aspekt für die Warteschlangensimulation

gesamte Modell und den Simulationskern verteilt, deshalb wird in Spezifikationen für DEViL das Konzept der aspektorientierten Programmierung (AOP) eingesetzt [57]. Die AOP versucht logische Programmelemente, so genannte Aspekte, getrennt voneinander zu entwickeln und anschließend in das Programm einzupflegen. Dieser Prozess wird auch "Weben" genannt. DEViL benutzt den Aspekteweber *AspectC++* [110].

In Listing 6.1 ist ein Aspekt zu sehen. Er erweitert die Waschstraßen-Simulation und implementiert eine Warnung, falls ein Auto mehr als 10 Simulationsschritte warten muss. Wie zu erkennen ist, spannt sich dieser Aspekt über zwei getrennte und im Code weit entfernte Eigenschaften. Zum einen muss in jeder Autoinstanz die Wartezeit mitgezählt werden und zum anderen muss auf die Inkrementierung der Simulationszeit geachtet werden. Dies wird in der AOP auch als "*Cross-Cutting-Concern*" bezeichnet.

Zur Realisation wird der Simulationsklasse *Car* eine Variable hinzugefügt, ausgedrückt durch das Schlüsselwort *slice*. Immer wenn die Methode *simulateNextStep* ausgeführt wird - ausgedrückt durch den *execution-Advice* - wird die Zählvariable jeder Autoinstanz erhöht.

Die aspektorientierte Erweiterung in von DEViL generierten Editoren ermöglicht eine Überprüfung von Modelleigenschaften ohne bestehende Spezifikationen mit zusätzlichem Code aufzublähen oder unleserlich werden zu lassen. Das obige Beispiel könnte auch durch eine Spracherweiterung von DSIM erreicht werden. Dazu hätte die Simulationsstruktur erweitert und im `Loop`-Teil der Simulation die entsprechende Inkrementierung des erweiterten Zählattributs hinzugefügt werden müssen. Somit wären zwei eigentlich getrennte Aspekte mit all ihren Konsequenzen miteinander verwoben worden, nämlich Simulation und Strukturuntersuchung.

Die AOP Erweiterung eignet sich übrigens nicht nur für die Simulation, auch in anderen Teilen der Analyse von Struktureditoren lässt sie sich einbauen. Konsistenzfunktionen über dem semantischen Modell lassen sich so auch ausdrücken. Ein Nachteil ist die stark an C++ orientierte Syntax (es kann beliebiger C++ Code eingefügt werden) sowie die Offenlegung interner Schnittstellen. Dies könnte jedoch durch eine spezialisierte (evtl. visuelle) domänenspezifische Sprache vermieden werden.

# 7 Evaluation

## Inhalt

---

<b>7.1</b>	<b>Grundlagen der Usability</b>	<b>151</b>
<b>7.2</b>	<b>Usability-Maße</b>	<b>152</b>
<b>7.3</b>	<b>Allgemeine Methoden zur Usability Untersuchung</b>	<b>153</b>
<b>7.4</b>	<b>Usability des Generators</b>	<b>154</b>
7.4.1	Implementierung von Beispielsprachen	155
7.4.2	Feld-Beobachtung	166
7.4.3	Expertenbewertung	170
7.4.4	Kontrolliertes Experiment	173
<b>7.5</b>	<b>Geschwindigkeit</b>	<b>177</b>

---

Das hier entworfene Konzept zur Simulation und Animation visueller Sprachen in DEViL hat das Ziel einfach anwendbar zu sein und ein großes Sprachspektrum abzudecken. Einfache Anwendbarkeit bedeutet, dass Benutzer, die zum ersten Mal mit dem DEViL System arbeiten, Konzepte der Simulation und Animation schnell verstehen und anwenden können. Benutzer, die bereits mit DEViL visuelle Sprachen implementiert haben, sollen in der Simulations- und Animationserweiterung bekannte Konzepte wiedererkennen. So soll die Hürde ein Programm zu simulieren möglichst klein gehalten werden. Dies ist gerade im Rapid-Prototyping ein wichtiger Aspekt. Hier müssen schnell Sprachimplementierungen mit eingeschränkter Funktionalität implementiert werden um Designentscheidungen explorieren zu können.

Eine Usability-Untersuchung soll die einfache Anwendbarkeit des entworfenen Simulations- und Animationskonzepts belegen. Usability ist dabei definiert als die Effektivität, Effizienz und Zufriedenheit mit der Benutzer ein bestimmtes Ziel erreichen. Das Ziel ist in diesem Fall der Entwurf einer Simulation und Animation für eine visuelle Sprachen. Schmidt hat in [101] bereits die Usability des Generatorsystems DEViL und der generierten Produkte gezeigt. Er hat dabei durch Fallstudien, Fragebögen und kontrollierte Experimente beide Ebenen untersucht und kommt zu dem Schluss, dass das Spezifikationskonzept gut einsetzbar ist - sowohl für kleine visuelle Sprachen als auch für komplexere Systeme. Auch Teamarbeit wird ausreichend unterstützt. Lediglich Anfängern macht das System noch Probleme, was jedoch prinzipbedingt ist, denn ein einfacherer Spezifikationsmechanismus würde das Sprachspektrum einschränken. Auf Produktebene zeigt sich, dass die generierten Editoren sehr benutzerfreundlich sind. Dies wird durch etablierte Interaktionsmechanismen, *direct manipulation*, *Drag-and-Drop* sowie der Kapselung von Expertenwissen erreicht.

Cebeci hat in [17] das DEViL Spezifikationskonzept mit dem Eclipse Framework [13] verglichen. Dabei kommt er zu dem Schluss, dass die DEViL-Konzepte visuelle Muster, Pfadausdrücke, 2,5 dimensionale-Sichten und Konsistenzprüfungen dem GMF Konzept überlegen sind. Vorteil von GMF sind die breite Akzeptanz der generierten Editoren sowie die umfangreichen Erweiterungsmöglichkeiten durch das Plugin-Konzept von Eclipse und die große Entwicklergemeinschaft.

Die folgende Usability Untersuchung wird sich auf die Spezifikationsmechanismen für Simulation und Animation konzentrieren. Ob eine Simulation für eine gegebene Sprache sinnvoll ist bzw. ob eine visuelle Sprache benutzerfreundlich ist, soll nicht Teil der Untersuchung sein. Usability Untersuchungen hinsichtlich der Simulation werden in [82] sowie in [35] betrachtet. Untersuchungen von visuellen Sprachen können mittels des *Cognitive Dimensions Framework* von Petre et al. in [42] durchgeführt werden.

Im Folgenden soll zunächst auf die Grundlagen der Usability eingegangen werden. Zentrale Begriffe und Methoden der Usability Untersuchung werden vorgestellt.

Dabei orientiere ich mich an dem Buch "An Introduction to Usability" von Patrick W. Jordan [51]. Im Anschluss werde ich die Usability Untersuchung des Simulations- und Animationskonzeptes vorstellen.

### 7.1 Grundlagen der Usability

Usability ist laut ISO-Norm 9241-11 wie folgt definiert:

Die Usability eines Produktes ist das Ausmaß an Effizienz, Effektivität und Zufriedenheit die von einem bestimmten Benutzer erreicht werden kann, der ein bestimmtes Ziel mit dem Produkt verfolgt.

Effektivität bedeutet dabei, bis zu welchem Ausmaß ein Ziel oder eine Aufgabe erfüllt werden kann. Dabei kann ein Ziel nicht nur ganz oder gar nicht erreicht werden, sondern auch nur zu Teilen. Wenn beispielsweise auf einer Fertigungsstraße pro Tag 100 Komponenten erstellt werden sollen und am Ende des Tages 80 Komponenten tatsächlich erstellt wurden, so kann man sagen, dass die Fertigungsstraße eine Effektivität von 80% erreicht. Bezogen auf die hier erstellte Erweiterung des DEViL-Systems bedeutet Effektivität, ob ein Struktureditor für eine visuelle Sprache geeignet simulier- und animierbar generiert werden kann bzw. ob es Einschränkungen gibt.

Effizienz misst, mit welchem Aufwand ein Ziel erreicht werden kann. Als Maß kann die Zeit genommen werden, die benötigt wird, um die Aufgabe zu erledigen. Aber auch die Fehler, die der Benutzer macht, können gezählt werden. Wenn eine Aufgabe nur zu erledigen ist, wenn sich der Benutzer sehr stark konzentriert, so ist die Effizienz niedriger als bei einem System, bei der der Benutzer die Aufgabe sehr leicht erledigen kann. Dies wirkt sich auch direkt auf die Zufriedenheit aus. Zufriedenheit ist ein eher subjektives Empfinden und ist schwer zu messen. Auch kann es sein, dass ein erfahrener Benutzer eine Aufgabe als angenehmer empfindet, als ein noch unerfahrener Benutzer. Somit hängt die Zufriedenheit, aber auch Effizienz und Effektivität immer auch vom Benutzerkreis und dem entsprechenden Ziel ab.

Benutzer werden hinsichtlich mehrerer Faktoren, die Einfluss auf die Usability haben, klassifiziert. Dazu gehören u.a.

- Erfahrung mit dem zu untersuchendem Produkt.
- Erfahrung im Problembereich.
- Kultureller Hintergrund.
- Alter und Geschlecht.
- Behinderungen.

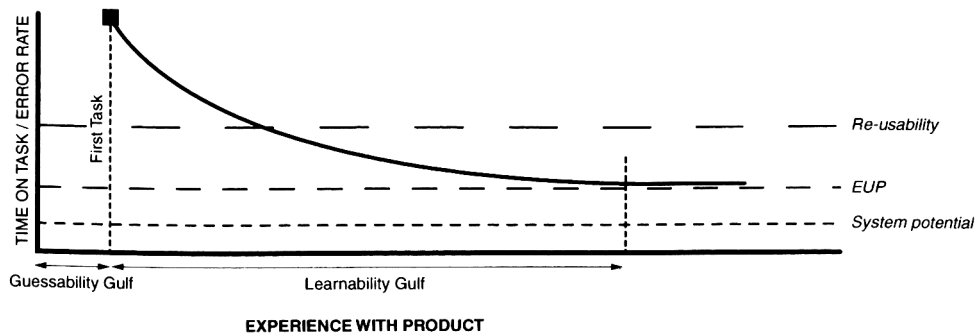


Abbildung 7.1: Einfluß der Erfahrung auf die Usability (aus [51])

Der Aspekt der Erfahrung des Benutzers nimmt eine Sonderstellung ein. Ist ein Benutzer bereits erfahren im Umgang mit einem Produkt, so sinkt seine Fehlerrate und er beurteilt das Produkt anders als ein unerfahrener Benutzer. Ein erfahrener Benutzer kann aus den Aufgaben, die er bereits mit dem Produkt oder einem ähnlichen Produkt erledigt hat, verallgemeinern. Diese Kenntnisse kann er bei einer neuen Aufgabe anwenden. Wenn das Produkt konsistent gestaltet ist, kann er sein Ziel schneller und mit weniger Fehlern erreichen.

Abbildung 7.1 zeigt den Zusammenhang zwischen verschiedenen Aspekten der Usability und der Zeit, die ein Anwender mit einem Produkt verbringt. Je mehr Erfahrung der Benutzer mit dem Produkt hat, umso weniger Fehler macht er und die Lernkurve nähert sich der so genannten *experienced user performance* (EUP) an. Dabei handelt es sich um die Zeit, die ein erfahrener Benutzer braucht um eine Aufgabe zu lösen. Das Systempotential (*system potential*) ist die theoretisch minimale Zeit, die benötigt wird um eine Aufgabe mit einem bestimmten Produkt zu lösen. Das Systempotential wird häufig auch von einem erfahrenen Benutzer nicht erreicht, da er die optimale Problemlösungsmethode nicht gelernt hat und meist auch gar nicht danach sucht, da er mit seiner bisherigen Methodik zufrieden ist. *Re-usability* beschreibt den Aufwand den ein erfahrener Benutzer benötigt um das Produkt nach einer längeren Pause wieder zu benutzen. Der Selbsterklärungsgrad (*guessability*) beschreibt die Kosten für einen Benutzer, der ein Produkt zum ersten Mal benutzt.

## 7.2 Usability-Maße

Zur Messung von Usability wurden verschiedene Methoden entwickelt [51]. Die Methoden, die auch ich in der folgenden Untersuchung verwenden möchte, werde ich nun kurz darstellen:

**Task Completion** misst inwieweit eine Aufgabe mit dem Produkt erfüllt wurde. Ich werde diese Methode an Hand der implementierten Beispielanwendungen benutzen und zeigen inwieweit der jeweilig implementierte Editor das Ziel erreicht hat.

**Quality of Output** misst, wie gut das Produkt gewissen Qualitätskriterien entspricht. Dies ist im Zusammenhang mit generierten Struktureditoren ein recht subjektives Kriterium. Trotzdem kann es Aufschluss über etwaige Schwachstellen bieten. Ich werde diese Methode an zwei Stellen anwenden. Das Spiel *“Mensch-ärger-Dich-nicht”* wurde bereits mit anderen Generatorsystemen implementiert und die Ergebnisse dokumentiert. Auch mit DEViL wurde ein entsprechender Editor entworfen. Somit können bis zu einem gewissen Maß die Ergebnisse verglichen werden. Außerdem existiert in der Literatur eine Petri-Netz-Implementierung mit Hilfe von Abstract State Machines. Diese werde ich ebenfalls mit dem von DEViL generierten Editor vergleichen.

Des Weiteren werde ich innerhalb von **kontrollierten Experimenten** Interviews mit Probanden machen und hier Fragen zur Qualität des generierten Produkts stellen.

**Deviation from critical path** misst wie weit der Anwender bei der Bearbeitung einer Aufgabe von der effizientesten Vorgehensweise abweicht. Dazu gehören auch das Konsultieren von Handbüchern, Online-Hilfen oder Beispielen. Ich werde diese Methode im Rahmen von kontrollierten Experimenten einsetzen. In diesen Experimenten wird ebenfalls die Zeit gemessen - dies ist die Methode **time-on-task**.

Die Methode **quantitative attitude analysis** misst, wie zufrieden der Benutzer mit der Bewältigung seiner Aufgabe ist bzw. wie gut er das Ergebnis findet. Diese Variante werde ich im Zusammenhang mit Interviews benutzen, die nach einem kontrollierten Experiment gemacht werden.

## 7.3 Allgemeine Methoden zur Usability Untersuchung

In Jordans Buch werden alleine 16 Methoden zur Usability Untersuchung vorgestellt. Dies zeigt, dass eine vollständige Usability Evaluation nahezu beliebig umfangreich werden kann. Ich möchte deshalb fünf Methoden herausgreifen um die Usability des entworfenen Konzepts darzustellen. Diese sollen nun im Folgenden kurz erläutert werden.

**Fragebogen:** Die Benutzer füllen einen Fragebogen aus auf dem entweder Antwortmöglichkeiten vorgegeben sind oder der Benutzer seine Meinung frei formulieren kann. Ich werde eine Mischung aus beiden Varianten verwenden. Die Benutzer werden hauptsächlich Fragen beantworten, bei der die Antworten durch Ankreuzen eines festen Spektrums vorgegeben sind. Abschließend wird es eine qualitative Frage geben, wie *“Sind Sie zufrieden mit dem Ergebnis?”* oder *“Haben Sie ergänzende Vorschläge?”* Der Vorteil von Fragebögen ist, dass sehr schnell eine große Anzahl an Probanden befragt werden kann, sowie die Anonymität, die zu markanteren Ergebnissen führen kann, als z. B. bei einem Interview. Von Nachteil sind Fragen,

die vom Probanden nicht richtig verstanden wurden und so zu missverständlichen Antworten führen.

In einem **Interview** wird ein einzelner Proband zum Produkt befragt. Die Fragen können dabei sehr stringent formuliert sein, sodass der Interviewer genau zu bestimmten Punkten Antworten bekommt. Eine andere Variante sind unstrukturierte Interviews, bei denen der Proband die Diskussion in eine Richtung führen kann, die für ihn besonders wichtig ist. Gegenüber Fragebögen haben Interviews den Vorteil, dass der Proband die Möglichkeit hat, sich Fragen erklären zu lassen, die er nicht verstanden hat. Von Nachteil ist, dass die Probanden sich in einem direkten Gespräch nicht so sehr zu Extremaussagen hinreißen lassen.

In einem **kontrolliertem Experiment** wird dem Probanden quasi unter Laborbedingungen eine Aufgabe gestellt, die mit dem Produkt bearbeitet werden muss. Diese Experimente offenbaren häufig kleine Bedienungsfehler und auch Abweichungen vom effizientesten Lösungsweg, die sonst nicht aufgefallen wären. Ungünstig ist die künstliche Situation, die manche Benutzer als unangenehm empfinden.

Bei der **Feld Beobachtung** werden Benutzer bei der täglichen Arbeit mit dem Produkt beobachtet. Dies kommt der realen Situation am nächsten. Ich hatte im Rahmen einer Projektgruppe, die mit zehn Studenten ein Jahr lang an einer visuellen Sprache zur Verkehrssimulation gearbeitet hat, die Möglichkeit eine Feld-Beobachtung zu machen. Die Ergebnisse werden später vorgestellt.

Bei der **Experten-Abschätzung** bewertet ein Experte das Produkt. Er vergleicht es mit anderen Produkten oder arbeitet eine Feature-Checkliste ab. Von Vorteil ist diese Methode, da keine Probanden benötigt werden. Außerdem kann der Experte bei Problemen direkt Lösungen vorschlagen. Von Nachteil ist, dass der Experte natürlich keine empirische Aussage über die tatsächliche Usability geben kann.

## 7.4 Usability des Generators

Wie bereits zu Anfang des Kapitels erwähnt, möchte ich nun auf die Usability des Generators, d. h. die Spezifikation von Simulation und Animation eingehen. Die Usability der generierten Produkte hat Schmidt bereits untersucht. Der Sinn von Simulation und dynamischer Zeichengebung wurde in der Literatur bereits bestätigt.

Wenn die Usability des hier gezeigten Ansatzes untersucht werden soll, stellen sich prinzipiell folgende grundlegende Fragen:

1. Ist die Methodik ausreichend um ein möglichst großes Spektrum an Sprachen zu simulieren und animieren?



2. Wie einfach lassen sich Struktureditoren für VLs mit einer Simulation ausstatten?
3. Wie nah kommt die automatisch generierte Animation an das gewünschte Ergebnis heran?
4. Wie einfach lässt sich die Animation durch Animationsmuster anpassen?
5. Wie gut ist das entworfene Konzept hinsichtlich der Arbeit im Team?

Der erste Punkt ist besonders wichtig, denn eine Einschränkung des Sprachspektrums würde bedeuten, dass nicht unbedingt alle Sprachen nachträglich mit Simulations- und Animationsunterstützung ausgestattet werden könnten. Insbesondere müsste ein Sprachentwickler bereits zu Beginn des Entwurfs über eventuelle Einschränkungen nachdenken. Dies würde zu einer nicht optimalen Sprache führen. Ich werde diesen Punkt im Folgenden anhand der implementierten Beispielsprachen untersuchen.

Der zweite Punkt spricht die Simulationsspezifikation an. Er ist nicht ganz leicht zu untersuchen, da Verhaltensspezifikationen prinzipiell beliebig komplex sein können, was in der Natur der jeweiligen Sprache liegt. Ich werde versuchen, anhand eines kontrollierten Experiments zu zeigen, dass einfache Simulationen in relativ kurzer Zeit entworfen werden können. Hier spielen insbesondere auch die Debugging-Fähigkeiten des Systems eine Rolle. Durch Fragebögen werde ich versuchen herauszuarbeiten, ob Komponenten wie der DSIM-Debugger, Log-Ausgaben oder die automatisch generierte Animation helfen, eine korrekte Simulation zu entwerfen.

Die Punkte drei und vier zielen auf einen Kernaspekt dieser Arbeit ab: die graphische Interpolation bzw. die Animationsmuster. In den Beispielimplementierungen und anhand von Fragebögen werde ich klären, ob und wie gut das Ziel eine anspruchsvolle Animation für eine Sprache zu implementieren, erreicht wurde. Dies ist ein sehr subjektiver Aspekt, trotzdem kann gerade in Interviews geklärt werden, wo noch Schwächen existieren bzw. inwieweit ein Benutzer mit einer Animation zufrieden ist.

Die Teamarbeit konnte ich anhand der einjährigen studentischen Projektgruppe genauer untersuchen.

### 7.4.1 Implementierung von Beispielsprachen

Der wichtigste Aspekt um die Usability des Systems darzustellen ist, das breite Spektrum an Beispielimplementierungen zu zeigen. Die Sprachen wurden zum Teil von mir, aber auch in Studien- oder Projektarbeiten entwickelt. Insgesamt wurden bisher 16 Sprachen mit Simulations- und teilweise Animationsunterstützung ausgestattet. Dazu wurde mehr als 5000 Zeilen DSIM Code spezifiziert. Im Folgenden sollen die

implementierten Sprachen vorgestellt und Besonderheiten betrachtet werden. Am Ende jeder Darstellung ist ein Internet-Link zu finden, der auf ein Video verweist, das die Animation der jeweiligen Sprache zeigt.

**Petri-Netze (Abb. 7.2a-c)** Petri-Netze sind eine bekannte Modellierungstechnik um parallele Abläufe zu beschreiben. Im Rahmen dieser Arbeit wurden mehrere Varianten implementiert. Die erste Variante ist eine klassische Petri-Netz Simulation mit Kantengewichtung von eins. Die Marken fliegen weich von der Vorbedingung zur Nachbedingung einer Transition. Können mehrere Transitionen mit der gleichen Vorbedingung schalten, wird eine Transition zufällig gewählt. Die einfache Petri-Netz-Implementierung benötigt nur 29 Zeilen an Simulationscode. Zwei Pfadausdrücke erweitern die Simulationsstruktur und berechnen Vor- und Nachbedingung einer Transition. Das Besondere an Petri-Netzen ist der Verlust an Information zwischen Simulation und Animation. Die Transition, die geschaltet hat, kann nicht einfach in der Animation berechnet werden. Aus diesem Grund wurde die, in Abschnitt 5.11 auf Seite 133 dargestellte, *Verursacherliste* implementiert.

Die zweite Petri-Netz-Implementierung ist interaktiv. Die Simulation fragt den Benutzer, welche Transition schalten soll, falls mehrere zur Auswahl stehen. Dazu werden die entsprechenden Transitionen visuell in der Darstellung markiert. Dieses Beispiel zeigt also modale Benutzerinteraktion als Teil der Simulationssprache. Die Spezifikation ist mit 94 Zeilen ebenfalls nicht sehr lang. Die Animation beider Petri-Netz Implementierungen zeigt - wie erwartet - die kontinuierliche Darstellung der Zustandsübergänge an. Feuernde Transitionen blinken zusätzlich. Hier wird die Triggerung von Animationsmustern eingesetzt.

Die dritte Petri-Netz-Variante nutzt gekoppelte Strukturen. Dabei existieren zwei verschiedene Sprachimplementierungen. Die erste Implementierung verwendet ein spezielles Petri-Netz um eine Ampelanlage zu steuern, die zweite Implementierung verwendet eine Petri-Netz-Instanz um das bekannte *Dining-Philosophers*-Problem darzustellen. Alle Petri-Netz-Varianten basieren auf demselben Simulationscode. Die *Ampel*- bzw. *Dining-Philosophers*-Sicht wird durch Strukturkopplung erstellt. Hierzu wird der bereits vor dieser Arbeit entworfene Mechanismus benutzt. Der Simulator erkennt Änderungen in der gekoppelten Struktur und kann sie, genauso wie die Basisstruktur, animieren. So werden im *Dining-Philosophers*-Beispiel die Gabeln entsprechend bewegt. Das Schalten der Ampel erfolgt weiterhin diskret, da ein Morphen der Farben nicht der Realität entsprechen würde.

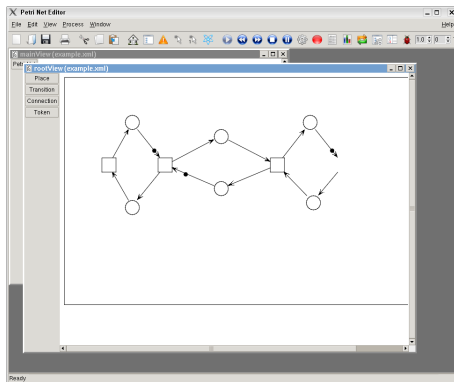
[<http://devil.cs.upb.de/videos/simplePetri.html>]

[<http://devil.cs.upb.de/videos/simplePetri-userChoice.html>]

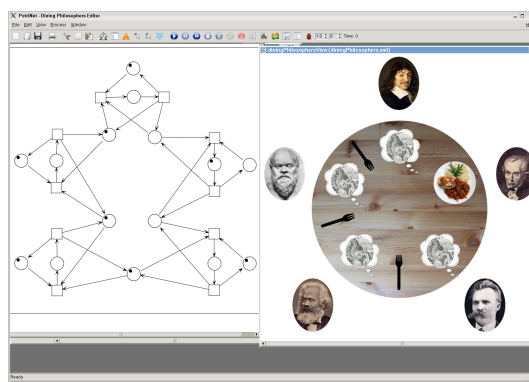
[<http://devil.cs.upb.de/videos/simplePetri-signalLights.html>]

[<http://devil.cs.upb.de/videos/simplePetri-diningPhilosophers.html>]

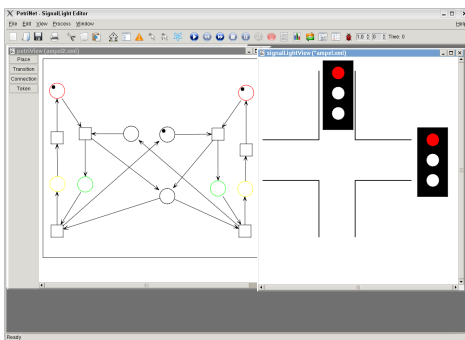
## 7.4. USABILITY DES GENERATORS



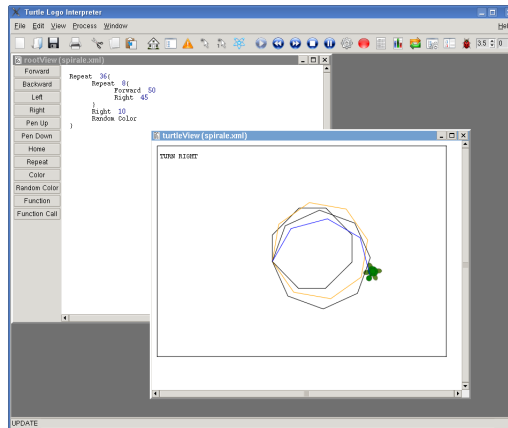
a) Petri-Netze



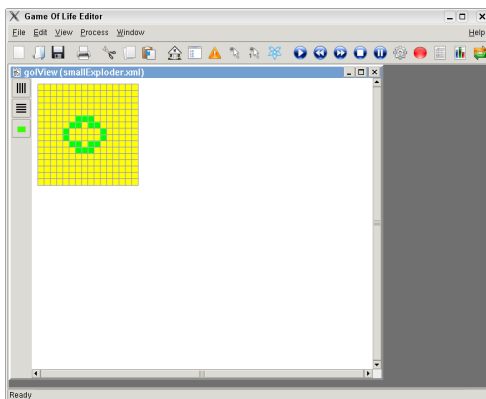
b) Petri-Netz steuert Dining Philosophers



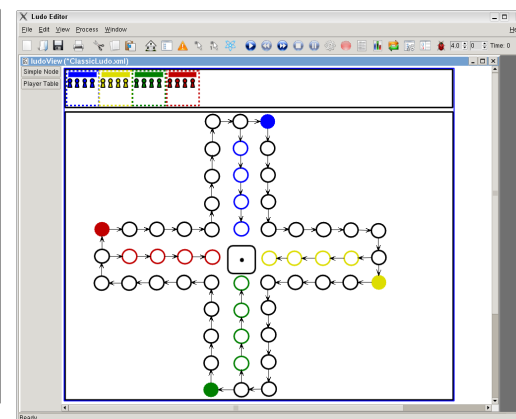
c) Petri-Netz steuert Ampel



d) Logo



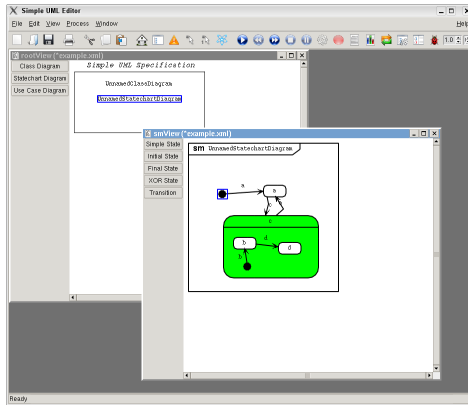
e) Game-of-Life



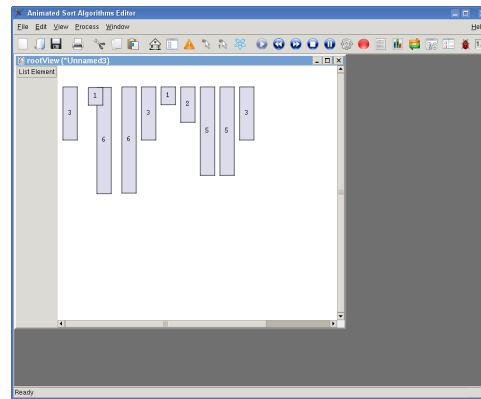
f) Mensch-ärger-Dich-nicht

Abbildung 7.2: Beispielimplementierungen

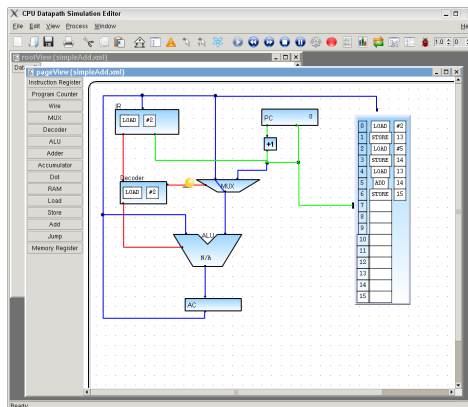
# KAPITEL 7. EVALUATION



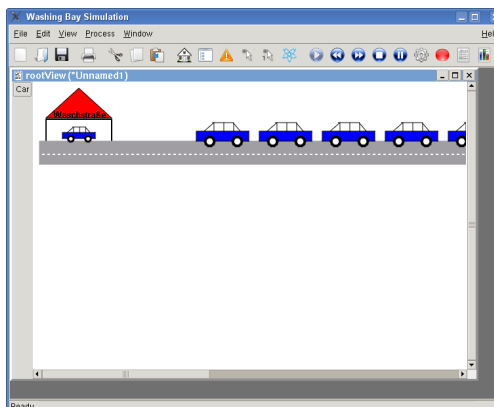
a) Statecharts



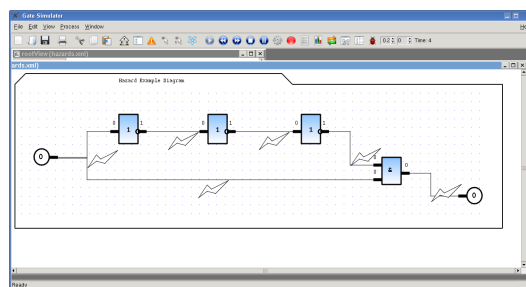
b) Bubblesort/Quicksort



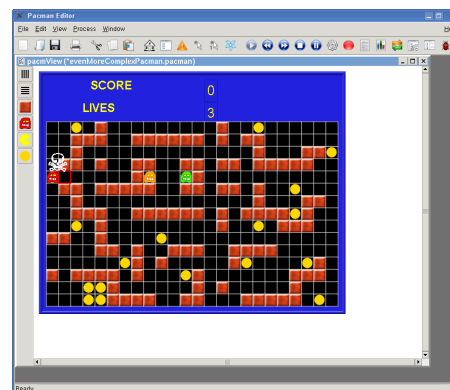
c) CPU Datenpfad



d) Waschstraße



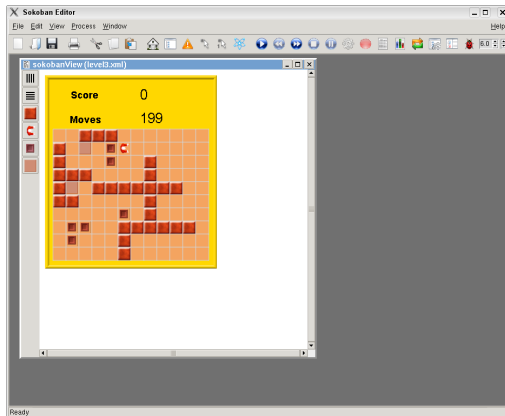
e) Elektronische Schaltkreise



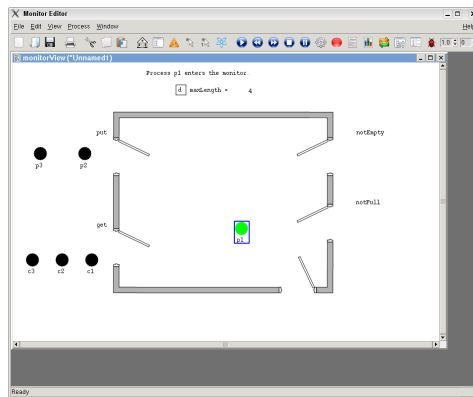
f) Pac-man

Abbildung 7.3: Beispielimplementierungen

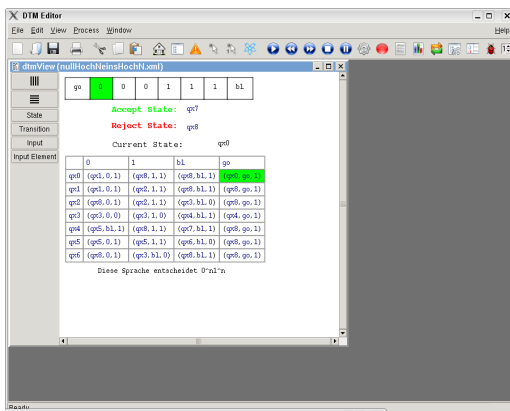
## 7.4. USABILITY DES GENERATORS



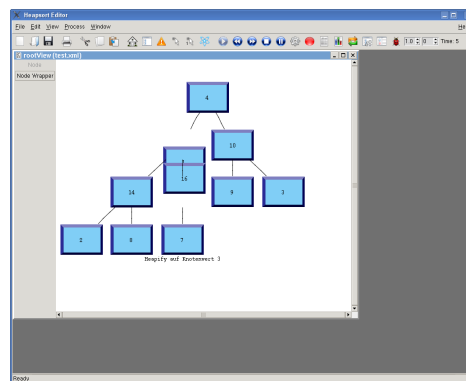
a) Sokoban



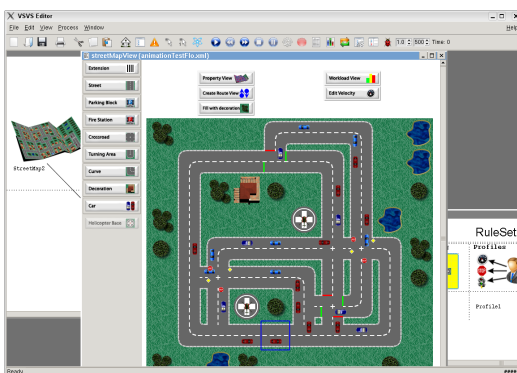
b) Monitor



c) deterministische Turingmaschine



d) Heapsort



e) Verkehrssimulation

Abbildung 7.4: Beispielimplementierungen

**Logo (Abb. 7.2d)** Logo ist eine imperative Lernsprache, die ohne Zustandsvariablen auskommt. Mit Befehlen wie `Vorwärts 100`, `Rechts 90` usw. kann ein Cursor, meist dargestellt durch eine Schildkröte, bewegt bzw. rotiert werden. Der implementierte Editor erlaubt die Zusammensetzung der Befehle durch Textbausteine. Insgesamt wurden 12 Befehle implementiert, darunter auch Funktionsdefinitionen und Funktionsaufrufe. Das Beispiel demonstriert auf Seiten der Simulation insbesondere die Verwendung von kontext-sensitiven Funktionen (vgl. Abschnitt 4.3.1). So lassen sich sehr leicht neue Befehle integrieren. Besonderheit ist hier, dass Teile der visuellen Sprache kombiniert werden können, die eine spezielle Simulationssemantik haben. Diese werden dann auf ein dediziertes Sprachkonstrukt - den Cursor - angewendet. Die Animation zeigt die Rotation und die Bewegung der Schildkröte. Hier wurden die Animationsmuster *Bewegung* und *Rotation* benutzt. Zusätzlich hinterlässt der Cursor eine Bewegungsspur. Diese wurde mit Hilfe statischer Animationsobjekte realisiert. Hilfetexte stellen den aktuell ausgeführten Befehl als Text in einer Ecke des Simulationsfensters dar. Die Animation konnte mit nur drei Zeilen Code hergeleitet werden. Sie entspricht vollständig der Animation in anderen Systemen, die einen Logo-Interpreter realisieren.  
[<http://devil.cs.upb.de/videos/turtle.html>]

**Game-of-Life (Abb. 7.2e)** Das Game-of-Life ist ein kachelbasierter Struktureditor auf dem Kacheln als lebendig oder tot (mit entsprechender Visualisierung) gekennzeichnet werden können. In jedem Simulationsschritt wird überprüft ob Nachbarkacheln als tot oder lebendig markiert werden können. Die Entscheidung darüber wird getroffen, indem die Anzahl der lebendigen Kacheln in der Umgebung gezählt wird. Ist eine tote Kachel von beispielsweise drei lebendigen Kacheln umgeben, so wird sie im nächsten Schritt als lebendig markiert. Anhand weniger solcher Regeln lassen sich mit dem Game-of-Life-Editor eindrucksvolle Simulationen erstellen. Der implementierte Editor benötigt nur 39 Zeilen Simulationscode und demonstriert die Fähigkeit von DSIM mit so genannten Nachbarschaftsfunktionen die Anzahl an Kacheln eines bestimmten Typs in der Umgebung einer bestimmten Kachel zu ermitteln. Der Editor hat keine Animation. Die diskreten Simulationsschritte reichen aus um die Zustandsübergänge eindrucksvoll zu visualisieren.  
[<http://devil.cs.upb.de/videos/gol.html>]

**Ludo (Abb. 7.2f)** Ludo ist die Implementierung des bekannten "*Mensch-ärgere-Dich-nicht*"-Spiels. Der generierte Editor erlaubt es, das Spiel gegen drei Computergegner zu spielen. Dabei lag der Fokus auf der Interaktion mit dem Spieler. Der Computer würfelt und der Benutzer kann entscheiden, welche Spielfigur ziehen soll. Dabei werden nur Spielfiguren angezeigt, deren Zug nicht gegen die Regeln des Spiels verstoßen. Mit 338 Zeilen Simulationscode gehört Ludo zu den größeren Beispielimplementierungen. Dies liegt vor allem daran, dass bei einem Wurf für den aktuellen Spieler geprüft werden muss, ob die Figur auch ziehen darf. Die Animation der

Spielfiguren besteht im Wesentlichen aus linearen Bewegungen. Die Augenzahl des Würfels wird durch ein Überblenden animiert. Beides kann automatisch und ohne zusätzlichen Code abgeleitet werden.

[<http://devil.cs.upb.de/videos/ludo.html>]

**Statecharts (Abb. 7.3a)** Statecharts sind Teil der UML und modellieren hierarchische Zustandsübergänge. Der implementierte Beispielditor demonstriert auf Seiten der Simulation die attributorientierte Simulation, d. h. es wird immer nur ein Boole'sches Attribut des jeweiligen Zustands auf aktiv bzw. inaktiv gesetzt. Zustandsübergänge werden durch Tastatureingaben ausgelöst. Insgesamt werden für die Simulation nur 78 Zeilen Code benötigt, obwohl die Simulationsspezifikation aufgrund der *XOR-Superstates* recht komplex ist. Die Animation morphet generische Zeichnungen, die den Zuständen je nach Boole'schem Attribut zugeordnet sind. So lässt sich ein flüssiger Farbverlauf von weiß nach grün für einen aktivierten Zustand darstellen. Auch hier wird kein zusätzlicher Animationscode benötigt.

[<http://devil.cs.upb.de/videos/statecharts.html>]

**Bubblesort/Quicksort (Abb. 7.3b)** Bubblesort und Quicksort sind beides Algorithmen-simulationen, die den Fokus auf die Tauschoperationen der Elemente legen. Das semantische Modell besteht nur aus einer Klasse, die ein Listenelement verdeutlicht. Entsprechend klein ist auch die editierbare Struktur. Die Simulation soll die algorithmischen Fähigkeiten und insbesondere die Zugriffsoperationen auf benachbarte Elemente in DSIM zeigen. So kann der Simulationscode mit nur 13 Zeilen für Bubblesort und 93 Zeilen für Quicksort sehr klein gehalten werden. Eine Besonderheit ist, dass der DSIM-Code im integrierten Debugger angezeigt werden kann und man somit noch eine Algorithmen-simulation erhält. Die Animation zeigt die Tauschoperationen flüssig an - es wird kein zusätzlicher Code benötigt!

[<http://devil.cs.upb.de/videos/bubblesort.html>]

[<http://devil.cs.upb.de/videos/quicksort.html>]

**CPU Datapath (Abb. 7.3c)** Die Datenpfad-Simulation stellt die Pipeline einer CPU dar und es können kleine Programme geschrieben werden, die Teil der Sprache sind und im RAM der VL-Instanz abgelegt werden. Die Simulation demonstriert - wie beim Logo-Editor - die Anwendung von kontext-sensitiven Funktionen. Je nachdem wo sich eine Instruktion innerhalb des Strukturbaums befindet, werden unterschiedliche Simulationsaktionen ausgelöst. Die Instruktionen werden quasi durch die Simulationsstruktur transportiert. Dies ist also auch ein klassisches Beispiel für eine transportorientierte- oder Fertigungssimulation. Die Animation bietet die Besonderheit, dass die Instruktionen entlang von Signalkabeln bewegt werden. Die Bewegung ist also nicht gleichmäßig gerichtet, sondern enthält auch noch Kurven. Dies lässt sich sehr leicht durch das Animationsmuster *AVPOnMoveMoveAlongLine* ausdrücken. Neben den Sprachkonstrukten, die bewegt werden, gibt es in der Anima-

tion noch Datenimpulse, die durch einen fliegenden Ball, der sich ebenfalls entlang von Kabeln bewegt, dargestellt wird. Hier kommen dynamische Animationsobjekte zum Einsatz (vgl. Abschnitt 5.1.6 auf Seite 127). Die recht hohe Anzahl an Zeilen für den Animationscode (160 Zeilen) ist damit zu erklären, dass die Animation kontextabhängig ist. Je nachdem wo sich eine Instruktion im Baum befindet, muss eine andere Animation getriggert werden. Befindet sich beispielsweise eine Instruktion im Decoder, so muss ein Datenimpuls an den Multiplexer gesendet werden; dies ist im Instruktionsregister jedoch nicht der Fall. Die Spezifikation für die Animation ist recht einfach und deklarativ gehalten. Zur Bestimmung von Positionen für Datenimpulse innerhalb einer generischen Zeichnung, wird das Konzept der graphischen Anknüpfungspunkte aus Abschnitt 5.1.8 verwendet.

[<http://devil.cs.upb.de/videos/datapath.html>]

**Washing Bay (Abb. 7.3d)** Die Waschstraßensimulation demonstriert eine zufallsbasierte Simulation. Auto-Objekte werden zufällig in einer Warteschlange erstellt und werden in die Waschstraße bewegt, sobald diese frei ist. Aus der Waschstraße werden sie entfernt, wenn die ebenfalls zufällige Bedienzeit abgelaufen ist. Die Animation stellt die Bewegungen der Autos dar. Durch das Morphen von Autos, die nicht modifiziert wurden, jedoch ihre Position durch Aufrücken in der Warteschlange verändern, wird eine realistische Animation erzeugt. Diese kann noch erhöht werden, wenn im Bewegungsmuster das *Easing* aktiviert wird (ein Kontrollattribut). Dann fahren die Autos langsam an und bremsen ebenfalls wieder langsam (slow-in-slow-out).

[<http://devil.cs.upb.de/videos/washingbay.html>]

**Electronic Circuits (Abb. 7.3e)** Mit dem Editor für elektronische Schaltkreise lassen sich kleine Schaltungen mit Gattern realisieren. Die gesamte Simulation ist attribut-orientiert spezifiziert. Somit werden auf Seiten der Animation dynamische Animationsobjekte benötigt um die Datenimpulse zu repräsentieren. Dies ist auch der Grund, dass die Anzahl der Zeilen für die Animation bei 109 liegt und die Simulationsspezifikation somit um einige Zeilen kleiner ist. Die Zeilen für den Animationscode hätten sicher reduziert werden können, wenn eine eher struktur-orientierte Simulationsvariante gewählt worden wäre. Allerdings war dieser Editor ein studentisches Projekt und zeigt somit, dass auch alternative Herangehensweisen vom entworfenen Konzept unterstützt werden.

[<http://devil.cs.upb.de/videos/electronic-circuits.html>]

**Pac-man (Abb. 7.3f)** Der Pac-man-Editor war Teil einer Studienarbeit [128] und wurde in [49] veröffentlicht. Er realisiert das bekannte Pac-man-Spiel, wobei der Benutzer durch die Pfeiltasten den Pac-man steuern und so genannte Powerpillen einsammeln muss. Dabei wird er von drei Geistern gejagt, die alle unterschiedlich intelligent im Sinne der Wegfindung sind. Ein Geist bewegt sich zufällig über das Spiel-



feld, ein anderer sucht die geringste euklidische Distanz zum Pac-man und der dritte Geist implementiert die so genannte Hill-Climbing Strategie, wobei jedem Feld ein Diffusionswert zugeordnet wird. Er beschreibt den schnellsten Weg zum Pac-man. Die Simulation enthält Interaktionsmechanismen und benutzt erweiterte Attribute der Simulationsstruktur um die Diffusionswerte zu speichern. Kachelzugriffsfunktionen werden für die Bewegung und für den Zugriff auf die Umgebung der Spielfiguren verwendet. Zusätzlicher C-Code ist nötig um eine Tiefensuche zu realisieren und die Diffusionswerte zu berechnen. Insgesamt werden nur 258 LOC für eine uneingeschränkt spielbare Pac-man Implementierung benötigt. Die Animation stellt die Bewegung und die Rotation des Pac-man flüssig dar. Hierzu werden lediglich einige Kontrollattribute überschrieben, die den zeitlichen Ablauf steuern. Eine gefressene Powerpille blinkt auf, bevor sie verschwindet. Wenn der Pac-man gefangen wird, wird ein Totenkopf eingeblendet, der sich über dem Pac-man schwebend bewegt. Hierzu werden nochmals sieben Zeilen Animationscode benötigt.

[<http://devil.cs.upb.de/videos/pacman.html>]

**Sokoban (Abb. 7.4a)** Sokoban ist ebenfalls wie Pac-man ein kachelbasiertes Spiel. Eine Spielfigur - der Sokoban - muss, vom Benutzer gesteuert, auf einem Spielfeld Kisten in bestimmte Positionen ziehen. Es spezifiziert im Vergleich zu Pac-man keine Neuerungen, ist jedoch einfacher im Hinblick auf die Strategie und hat deshalb auch weniger LOC in der Verhaltensspezifikation. Sokoban wird deshalb in einem kontrollierten Experiment im weiteren Verlauf dieser Evaluation noch genutzt.

[<http://devil.cs.upb.de/videos/sokoban.html>]

**Monitor (Abb. 7.4b)** Die Monitor-Implementierung soll die Kommunikation von Prozessen demonstrieren. Der Editor soll zu Lehrzwecken eingesetzt werden. Besonderheit ist, dass die Animation auf einem statischen Hintergrundbild stattfindet. Die Sprachkonstrukte bewegen sich innerhalb des Hintergrundbildes an bestimmte Positionen. Diese Positionen werden im Editor für generische Zeichnungen mit Hilfe des bereits genannten Konzepts der graphischen Anknüpfungspunkte spezifiziert.

[<http://devil.cs.upb.de/videos/monitor.html>]

**Deterministische Turingmaschine (Abb. 7.4c)** Das Beispiel implementiert einen Editor bei dem eine deterministische Turingmaschine mit Eingabe und Zustandsübergangsfunktionen modelliert werden kann. In jedem Simulationsschritt wird die Eingabe vom Band gelesen und anhand der definierten Zustandsübergänge ein neues Symbol geschrieben, der Lesekopf entsprechend weiterbewegt und ein neuer Zustand ausgewählt. Das Beispiel demonstriert im Wesentlichen das Setzen von Referenz-Attributen innerhalb der Simulation. Die gesamte Zustandsübergangsfunktion ist mit Hilfe von Referenzen auf eine vordefinierte Zustands- bzw. Eingabemenge realisiert. Die Animation zeigt die Übergänge des Lesekopfes während der Simulation. Die Hintergrundfarbe wird dabei entsprechend angepasst. Abbildung 7.4c

	Semantische Struktur	Editierbare Struktur	Simulation	Kopplung	Animation	syntactic sugar
Petri-Netz	38	40	29		4	0
Dining-Philosophers	38	189	29	95	4	0
Signal-Lights	38	158	29	57	4	0
Logo	66	40	211		3	3
Game of Life	25	40	39		0	0
Ludo	54	142	338		0	0
Statecharts	39	72	78		2	0
Bubblesort	8	40	13		0	0
Quicksort	8	40	93		0	0
CPU Datapath	140	469	263		160	0
Washing bay	9	25	35		0	0
Electronic circuits	65	331	99		109	0
Pac-man	53	120	258 (173)		21	7
Sokoban	40	85	157 (5)		10	0
Monitor	38	127	162		1	0
DTM	45	84	96		0	9
Verkehrssimulation	529	4928	3372		13	15
Heapsort	15	49	225		0	0

**Tabelle 7.1:** Spezifikationsaufwand in LOC für Editoren mit Simulations- und Animationsunterstützung. Zahlen in Klammern sind LOC für C-Funktionen.

zeigt ein Beispiel, das die Sprache  $L = \{0^n 1^n \mid n \geq 1\}$  entscheidet.  
[\[http://devil.cs.upb.de/videos/dtm.html\]](http://devil.cs.upb.de/videos/dtm.html)

**Heapsort (Abb. 7.4d)** Der Editor kann den Heapsort-Algorithmus auf Eingabedaten anwenden. Die Daten werden dabei als Knoten eines Binärbaumes modelliert. Der Algorithmus stellt die Tauschoperationen der Baumelemente flüssig dar. Dieses

Beispiel zeigt die Fähigkeiten der Animationskomponente, auch auf einem Baum-Layout zu arbeiten, dass von *Dot* [39] automatisch berechnet wurde.

[<http://devil.cs.upb.de/videos/heapsort.html>]

**Verkehrssimulation (Abb. 7.4e)** Die Sprache zur Verkehrssimulation wurde von einer aus zehn Studenten bestehenden Projektgruppe innerhalb eines Jahres entwickelt. Es ist das größte Beispiel und besteht aus mehr als 3000 Zeilen Simulationscode. Mit der Sprache können kachelbasierte Verkehrsszenarien entworfen werden. Auf diesen fahren dann verschiedene Typen von Autos und LKW entsprechend implementierter Strategien. Die Projektgruppe hat sich dazu entschieden als Basis das Nagel-Schreckenberg-Modell [74] zu implementieren. Mit diesem Modell war es zum ersten Mal möglich, den *Stau aus dem Nichts* nachzuvollziehen. Es existieren verschiedene Sichten um Eigenschaften der Strecken zu modellieren. So können Geschwindigkeitsbegrenzungen, Fahrerprofile, Straßenbedingungen oder Eigenschaften von Kreuzungen in speziellen Sichten eingestellt werden. Die Autos können auch entsprechend vordefinierter Routen fahren. So ist es möglich statistische Daten zu sammeln. Diese werden dann in einer speziellen Sicht graphisch aufbereitet. Zusätzlich können innerhalb der Sprachinstanz Verkehrsregeln anhand von Vorher-/Nachher-Regeln definiert werden. Diese sind Teil der visuellen Sprache und werden auf das Verkehrsszenario angewendet. Innerhalb der Simulationsspezifikation wird XPath genutzt um Regeln zu matchen. Die Verkehrsteilnehmer in der Sprachinstanz fahren unterschiedlich schnell; so kann ein Auto mit der Geschwindigkeit fünf z. B. innerhalb eines Simulationsschrittes fünf Kacheln befahren. Dies kann jedoch durch andere Verkehrsteilnehmer verhindert werden. Somit wird ein Simulationsschritt nochmals in so genannte Mikroschritte unterteilt. Es wird also für ein Auto mit Geschwindigkeit fünf überprüft, ob das Auto in die Kacheln eins bis fünf eingefügt werden kann. Dies hat direkte Konsequenzen für die Animation: Bewegt sich das Auto in den fünf Schritten um eine Kurve, so würde die automatisch generierte Animation eine direkte Bewegung animieren, was natürlich nicht der Realität entspricht. Somit mussten Animationsmuster entworfen werden, die entlang der Einfügehistorie innerhalb eines Simulationsschrittes animieren. Ein ähnliches Problem tritt bei der Rotation entlang von Kurvenabschnitten auf. Je nachdem von wo ein Auto auf eine Kurve fährt muss es entsprechend rotiert werden. Ein Animationsmuster, das den Typ der vorherigen Kachel sowie den Typ der aktuellen Kachel erkennt, löst dieses Problem.

[<http://devil.cs.upb.de/videos/vsvs.html>]

In Tabelle 7.4.1 sind nochmals die Beispielimplementierungen und die Zahl der effektiven Codezeilen (LOC) angegeben. Bemerkenswert ist das Verhältnis von DSIM-Code zu Animationscode. So liegt das Verhältnis zwischen Animations- und Verhaltensspezifikation für alle implementierten Sprachen zusammen genommen bei 1/16. Die automatisch generierte Standardanimation ist also in den meisten Fällen ausreichend.

Die Sprache zur Spezifikation von Simulationsverhalten ist ...	leicht anzuwenden	□□□□	schwer anzuwenden
Den Umgang mit DSIM empfinde ich als ...	leicht anzuwenden	□□□□	schwer anzuwenden

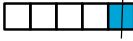



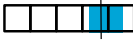

Abbildung 7.5: Ausschnitt aus einem Fragebogen: 5-stufige Likert-Skala

## 7.4.2 Feld-Beobachtung

Wie bereits erwähnt, wurde die Sprache zur Verkehrssimulation im Rahmen einer studentischen Projektgruppe erarbeitet. Die Gruppe arbeitete ein Jahr lang an der Sprache. Während dieser Zeit wurde ein kurzes Seminar abgehalten sowie der Problembereich der Verkehrssimulation erarbeitet. Neben der Implementierung haben die Projektgruppenteilnehmer ein Pflichtendokument, Beispiele, allgemeine Dokumentationen sowie ein Abschlussdokument entworfen. Teil der Arbeit war ein wöchentliches Treffen der zehn Teilnehmer. In diesem Treffen wurde über Konzepte der Sprache sowie die Verteilung der Arbeit besprochen. Die Teilnehmer einigten sich darauf, Gruppen für die semantische Struktur, die visuelle Darstellung, das Testen sowie bestimmte Teilaspekte der Simulation zu bilden. Bemerkenswert ist, dass nur einer der Teilnehmer Erfahrung mit dem DEViL System im Rahmen seiner Studienarbeit, in der der Pac-man-Editor entworfen wurde, hatte. Trotzdem ist die Sprache nicht nur von der Simulation, sondern auch von der semantischen als auch der editierbaren Struktur eine der anspruchsvollsten Beispielimplementierungen.

Die Erfahrungen aus der Projektgruppe liefern Antworten zu dem Aspekt, ob der entworfene Ansatz teamfähig ist, ob er für größere Simulationen bzw. Animationen skaliert und ob die integrierten Debugging-Fähigkeiten helfen, eine korrekte Simulation zu entwerfen. Bereits Schmidt konnte zeigen, dass das DEViL System gut geeignet ist um im Team zu arbeiten, da inkrementelle Entwicklung sowie nachträgliche Änderung von Struktur und Darstellung recht einfach zu realisieren sind.

Am Ende der Projektgruppe beantworteten die Teilnehmer Fragebögen. Die Bögen enthielten Fragen zu den Themenbereichen des Kenntnisstandes mit dem DEViL System, allgemeine Fragen zur Simulation, zur Simulationssprache, zur Animation und zu integrierten Debugging-Möglichkeiten. Die allgemeinen Fragen zum DEViL-System, die u.a. nach den Fähigkeiten in den Bereichen Spezifikation semantischer Strukturen mit DSSL, Sichtspezifikation und visuelle Muster enthielten, habe ich

Frage		Durchschnitt	Standard-abweichung
Wieviele Tage in der Woche haben Sie mit DSIM gearbeitet?		3,2	1,0
Ich verstehe das Konzept der Modellspezifikation (DSSL) zu ... (1) null Prozent ... (5) hundert Prozent		4,5	0,5
Ich verstehe das Konzept der generischen Zeichnungen zu ... (1) null Prozent ... (5) hundert Prozent		4,0	1,1
Ich verstehe das Konzept der visuellen Muster zu ... (1) null Prozent ... (5) hundert Prozent		3,9	1,5
Ich verstehe das Konzept der Sicht-Spezifikation zu ... (1) null Prozent ... (5) hundert Prozent		4,7	0,7
Ich verstehe DEViL insgesamt zu ... (1) null Prozent ... (5) hundert Prozent		3,9	0,6
Bezüglich der Spezifikation visueller Darstellungen mit DEViL bin ich ... (1) Anfänger ... (5) Experte		3,5	1,2
Schätzen Sie, wie oft Sie innerhalb des letzten Jahres ein visuelles Muster angewendet haben.		24	32

**Abbildung 7.6:** Fragebogen Teil 1: Aktueller Kenntnisstand

deshalb gestellt um einen Vergleich zu den neuen Konzepten herzuleiten.

Die Fragen an sich konnten mit einer 5-stufigen Likert Skala mit jeweils zwei unterschiedlichen Extremaussagen beantwortet werden. Aus den Ergebnissen habe ich den Mittelwert gebildet und die Standardabweichung berechnet. Die Standardabweichung gibt an, inwieweit sich die Teilnehmer bei der Aussage einig sind. Bei einer Standardabweichung von 0,5 Punkten schwanken die Aussagen um höchstens ein Feld. Über einer Standardabweichung von 1,0 Punkten gibt es teilweise extreme Ausreißer in den Aussagen.

Abbildung 7.6 zeigt, dass die Gruppe sich einen durchweg guten Kenntnisstand attestiert. Zu sehen ist, dass die Gruppe über Wochen hinweg durchschnittlich an 3,2 Tagen pro Woche mit DEViL gearbeitet hat. Dabei macht die Modellspezifikation offenbar die wenigsten Probleme. Es ist zu erkennen, dass sich die Gruppe nicht ganz einig darüber ist, inwieweit die Sichtspezifikationsmechanismen verstanden wurde. Trotzdem sind die Werte hier recht hoch angesiedelt. Ein Grund dafür ist, dass sich nur Teile der Gruppe mit den Sichten auseinandergesetzt haben, während andere Teile für Modell- oder Simulationsspezifikation zuständig waren. Ein interessanter Aspekt ist, dass die Projektgruppe während der gesamten Laufzeit nur eine Person abgestellt hat, um die Animation zu erstellen. Trotzdem ist eine anspruchsvolle Animation gelungen, die das Fahrverhalten von Autos realistisch abbildet. Dies zeigt, dass die automatisch generierte Animation selbst für große Sprachen gut geeignet ist.

Die Abbildungen 7.7 und 7.8 zeigen, dass sich die Teilnehmer der Projektgruppe alle

## KAPITEL 7. EVALUATION

Frage		Durchschnitt	Standard- abweichung
Das Prinzip der ereignisbasierten Simulation und der Ereigniswarteschlange verstehe ich zu ... (1) null Prozent ... (5) hundert Prozent		4,4	0,8
Das zeitliche Verhalten des Simulators verstehe ich zu ... (1) null Prozent ... (5) hundert Prozent		4,0	1,1
Der Unterschied zwischen dem TIME_NOW und dem TIME_DIRECT Konstrukt ist mir ... (1) völlig unklar ... (5) völlig klar		2,8	1,9
Ich verstehe das Konzept der Simulationsstrukturspezifikation zu ... (1) null Prozent ... (5) hundert Prozent		4,3	0,6
Die Trennung zwischen semantischer Struktur und Simulationsstruktur ist mir... (1) völlig unklar ... (5) völlig klar		3,5	1,5
Die Erweiterung der Simulationsstruktur verstehe ich zu ... (1) null Prozent ... (5) hundert Prozent		4,2	1,0
Die Anwendung von Pfadausdrücken zur Navigation in der Simulationsstruktur verstehe ich zu ... (1) null Prozent ... (5) hundert Prozent		3,8	1,4

**Abbildung 7.7:** Fragebogen Teil 2a: Fragen zur Simulation und zur Simulationspezifikation in DEViL

Frage		Durchschnitt	Standard- abweichung
Die Anwendung von XPath Ausdrücken zum Matchen von Strukturen verstehe ich zu ... (1) null Prozent ... (5) hundert Prozent		3,3	1,5
Wie Änderungen an der Simulationsstruktur vorgenommen werden und welche Operationen dazu existieren ist mir ... (1) völlig unklar ... (5) völlig klar		3,5	1,3
Statische Fehlermeldungen in DSIM sind ... (1) schwer verständlich ... (5) leicht verständlich		3,2	1,3
DSIM schränkt mich bei der Verhaltensspezifikation ... (1) stark ein ... (5) überhaupt nicht ein		3,7	1,4
DSIM schränkt mich beim Entwurf einer maßgeschneiderten Simulationsstruktur ... (1) stark ein ... (5) überhaupt nicht ein		3,0	1,2
Den Umgang mit DSIM empfinde ich als ... (1) unangenehm ... (5) angenehm		3,8	1,6

**Abbildung 7.8:** Fragebogen Teil 2b: Fragen zur Simulation und zur Simulationspezifikation in DEViL





Frage		Durchschnitt	Standardabweichung
Das Prinzip der automatisch generierten Animation ist mir ... (1) völlig unklar ... (5) völlig klar		3,9	1,5
Die Anwendung von Animationsmustern ist mir ... (1) völlig unklar ... (5) völlig klar		3,6	1,3
Das Prinzip der dynamischen Animationsobjekte ist mir ... (1) völlig unklar ... (5) völlig klar		2,4	1,3
Animationsmuster schränken mich beim Entwurf einer Animation ... (1) stark ein ... (5) überhaupt nicht ein		3,7	0,8

Abbildung 7.9: Fragebogen Teil 3: Animation

ein recht gutes Verständnis hinsichtlich der Grundkonzepte der ereignisbasierten Simulation und wie der Simulator konkret funktioniert attestieren. Obwohl alle Teilnehmer das Konzept der Simulationsstrukturdefinition verstehen und auch wissen, wie die Struktur erweitert werden kann, scheint das dahinter stehende Konzept eher unverständlich zu sein. Ich vermute, dass gerade in der Einführungsphase Details des Entwurfs nicht besprochen bzw. nicht verstanden wurden.

Bemerkenswert ist, dass die Operationen zur Änderung der Simulationsstruktur als unbekannt bewertet werden und das obwohl nur fünf dieser Operationen existieren. Ebenso erstaunlich ist die eher schwache Bewertung von DSIM. Jedoch ist bei diesen Fragen die Standardabweichung sehr hoch; die Gruppe ist sich also bei der Bewertung uneins. Ich vermute, dass diese Ergebnisse darauf zurückzuführen sind, dass Teile der Gruppe fast gar nicht oder nur sehr wenig mit DSIM gearbeitet haben. Zudem war die zu lösende Aufgabe der Verkehrssimulation algorithmisch sehr komplex. Ich schätze, dass sich in den Ergebnissen eher der algorithmische Anspruch der Aufgabe als die Komplexität oder Anwendbarkeit der Spezifikationsprache widerspiegelt. Diesen Eindruck unterstützen auch die Ergebnisse des kontrollierten Experiments. Hier wurde die Spezifikationsprache als eher einfach und angenehm anwendbar eingestuft.

Abbildung 7.9 zeigt die Bewertung der Animation. Die Ergebnisse sind statistisch gesehen nur unter Vorbehalt interpretierbar, da während der gesamten Projektgruppenzeit nur eine Person für die Animationsspezifikation abgestellt wurde. Der Rest hat sporadisch mit Animationsmustern und der Animation gearbeitet. Trotzdem ist eine ansehnliche Animation entstanden. Aus diesem Grund wird die Anwendung von Animationsmustern auch eher als schwieriger und innerhalb der Gruppe inkonsistenter bewertet als die Anwendung visueller Muster, die sicherlich schwieriger ist, da in der Regel mehrere Berechnungsrollen angewendet werden müssen.

Frage		Durchschnitt	Standard- abweichung
Die Simulationsstruktursicht hilft beim Debugging ... (1) überhaupt nicht ... (5) sehr gut		3,7	1,4
Der integrierte DSIM Debugger hilft beim Debugging ... (1) überhaupt nicht ... (5) sehr gut		3,2	0,9
Die Ereignishistorie hilft beim Debugging ... (1) überhaupt nicht ... (5) sehr gut		-	-
Die automatisch generierte Animation hilft beim Debugging ... (1) überhaupt nicht ... (5) sehr gut		2,4	1,0
Die automatischen oder selbstgeschriebenen Logging- Ausgaben helfen beim Debugging ... (1) überhaupt nicht ... (5) sehr gut		4,2	1,1

Abbildung 7.10: Fragebogen Teil 4: Debuggingmöglichkeiten in DEViL

Dynamische Animationsobjekte wurden nicht verstanden, was daran liegt, dass nur ein solches Objekt in der Animation definiert wurde. Der Rest der Animation war strukturorientiert. Insgesamt lässt sich für die Animation jedoch sagen, dass das Konzept verstanden wurde und zu keiner Einschränkung bei der Spezifikation einer beliebigen Animation führt. Abbildung 7.10 zeigt, dass Logging-Ausgaben zum Debuggen einer Simulation das Mittel der Wahl sind. Die Ausgaben werden als durchweg positiv bewertet. Die Simulationsstruktursicht wird im Zusammenhang mit der schrittweisen Simulation häufig benutzt und zeichnet sich besonderes durch die einfache Anwendbarkeit aus. Auch der integrierte Debugger für DSIM scheint sich gut zu eignen. Hier ist eventuell noch Nachbesserungsbedarf vorhanden, da der Debugger bisher hauptsächlich den Kontrollfluss zeigt und Werte bzw. Ausdrücke nur eingeschränkt dargestellt werden. Somit ist die Bewertung hier als besonders positiv zu betrachten.

Für die Ereignishistorie liegen keine Informationen vor, da diese Art der Darstellung bei vielen Objekten, die zur gleichen Zeit bewegt werden einen unübersichtlichen Graph erzeugt. Diese Darstellungsart ist nur für kleinere Sprachinstanzen mit wenigen Bewegungsoperationen gut geeignet. Vielen Projektgruppenmitgliedern war nicht klar, dass die Animation eine formale Abbildung ihrer Simulation ist. Sie wurde deshalb zu Debugging-Zwecken nicht eingesetzt, womit sich die hier schwache Bewertung erklärt.

### 7.4.3 Expertenbewertung

Zwei der oben vorgestellten Sprachen, eine einfache Petri-Netz-Implementierung sowie das "Mensch-ärgere-Dich-nicht"-Spiel wurden bereits von anderen Generatorsystemen spezifiziert. Ich möchte hier nun kurz die Implementierungen vergleichen. Mit dem Moses-System [33] wurde ein Petri-Netz-Interpreter implementiert. Die



```

1 class PNInterpreter[tm, G] is
2   define
3     P = {v | v ∈ vertices(G), v("type") = "Place"},
4     T = {v | v ∈ vertices(G), v("type") = "Transition"} ;
5   input {"fire"} : step ;
6   output {"schedule"} ;
7
8   function M arity 1 ;
9
10  initialize :
11  once
12    [OutputPorts(this)("schedule")] ← ⊥@tm,
13    do forall p ∈ P :
14      M(v) := v("initialTokens")
15    end
16  end
17
18  rule step[p, now, v] :
19  once
20    choose t ∈ {t | t ∈ T,
21      ∀e ∈ edges(G) :
22        (dst(e) = t ⇒ e("Weight") ≤ M(src(e)))} :
23
24    do forall e ∈ edges(G) :
25      if t = dst(e) then
26        M(src(e)) := M(src(e)) - e("Weight")
27      end.
28      if t = src(e) then
29        M(dst(e)) := M(dst(e)) + e("Weight")
30      end
31    end.
32    [OutputPorts(this)("schedule")] ← ⊥@now
33  end
34 end
35 end

```

Abbildung 7.11: Definition eines Petri-Netz Interpreters in Moses

Verhaltensspezifikation basiert auf der *Abstract State Machine Language* (ASML) und ist in Abbildung 7.11 zu sehen. ASML ist eine formale Sprache, die insbesondere für die Spezifikation von parallelen Prozessen benutzt werden kann. Sie ist klassenbasiert, unterstützt Vererbung und hat eine mathematische Notation. Die Verhaltensspezifikation in Moses benötigt in etwa genauso viele LOC wie die in DEViL, ist jedoch durch den hohen Grad an Abstraktion nicht so gut zu lesen.

In [91] ist eine Fallstudie beschrieben, bei der die Generatorsysteme Tiger, Dia-  
Meta, AToM<sup>3</sup> sowie Systeme wie Fujaba, XL, AGG/ROOTS und Groove das  
"Mensch-ärgere-Dich-nicht"-Spiel implementieren sollen. Teil der Spezifikation sollen  
die Visualisierung, also das Spielbrett, sowie Spielerstrategien sein. Neben der  
interaktiven Variante, d. h. der Spieler bewegt seine Spielfigur, soll es möglich sein,  
dass der Computer ein Spiel autonom und vollständig bis zum Ende spielt.

Das Papier lässt viel Raum für die verschiedenartigen Implementierungen. Au-

ßerdem werden keine quantitativen Angaben über LOC oder die Anzahl der implementierten Regeln gemacht, deshalb ist auch kein direkter Vergleich untereinander bzw. mit dem von DEViL generierten Editor möglich. Dies wird im Papier ähnlich bewertet. Trotzdem lassen sich einige Beobachtungen machen und auf das hier entworfene System übertragen. Alle Systeme können das *“Mensch-ärger-Dich-nicht”*-Spiel geeignet visualisieren. Ein System sogar in einer gerenderten 3-D-Ansicht. Die Systeme basieren alle auf der Anwendung von Graphtransaktionsregeln. Korrekte Züge werden somit entweder a priori vollständig berechnet oder ausprobiert und ggf. wieder rückgängig gemacht. Im DEViL Editor wird zunächst geprüft ob eine Spielfigur ziehen kann. Die Züge an sich werden durch das Zählen der Felder realisiert. In den Systemen aus dem Papier gab es noch die Varianten, die Felder zu numerieren oder so genannte *Single-Step*-Regeln anzuwenden. Dabei wird der Regel ein zusätzlicher Zähler hinzugefügt, der sukzessive dekrementiert wird.

Bei der Beschreibung des Verhaltens setzen fünf von acht Systemen auf den abstrakten Syntaxgraph. Nur AToM<sup>3</sup> und Tiger benutzen die konkrete Syntax. In XL genauso wie in DEViL - mit DSIM - wird eine textuelle Sprache zur Verhaltensspezifikation genutzt.

Bei der Interaktion gibt es die Varianten, dass die berechneten möglichen Regelmengen automatisch angewendet werden oder bei Nicht-Determinismus dem Benutzer eine Auswahl gegeben wird. Außerdem existiert die Variante, dass der Benutzer die Regelanwendung selbst erstellen muss. Hier sei erwähnt, dass die von DEViL generierten Editoren sowohl automatisch, als auch interaktiv durch Dialoge arbeiten können. Für den *“Mensch-ärger-Dich-nicht”*-Editor wurde die Variante mit drei Computerspielern und einem menschlichen Spieler gewählt. Aus diesem Grund lässt sich der Editor auch nicht vollständig autonom simulieren. Dies könnte jedoch recht leicht nachträglich implementiert werden. Die Systeme aus dem Papier können teilweise vollständig autonom simulieren und brauchen für ein Spiel zwischen 50 und 2600 Millisekunden.

Auf Seiten der Animation drängt sich der Vergleich mit Autorensystemen wie *Adobe Flash CS Professional* [48] auf. Mit Adobe Flash lassen sich komplexe Animationen erstellen. Dazu werden auf der Zeichenfläche, in Adobe Flash *Stage* genannt, graphische Primitive zu so genannten *Symbolen* zusammengesetzt. Symbole entsprechen im DEViL Sinn semantischen Sprachkonstrukten von denen beliebig viele Instanzen existieren können, die sich in Position und weiteren ausgewählten Parametern unterscheiden können. Dies entspricht etwa den Repräsentanten in DEViLs editierbarer Struktur. Eine Animation lässt sich durch Interpolation von Keyframes erreichen, die in Adobe Flash auf einer Zeitleiste positioniert werden können. Die Interpolation kann dabei ebenfalls durch Parameter angepasst werden um z. B. Easing-Effekte zu erreichen. Auf der Zeitleiste können beliebig viele, ggf. auch überlappende, Animationen in so genannten *Layers* positioniert werden. Somit können gleichzeitige Animationen erreicht werden. Die jeweiligen Layer können zu

so genannten *Movie-Clips* exportiert werden. Movie-Clips sind nach außen durch ihre Attribute parametrisierbar. Insbesondere lassen sie sich durch ActionScript starten und stoppen. Movie-Clips können wiederum beliebig viele Movie-Clips enthalten. So lassen sich z. B. bewegte Hintergründe realisieren. Benutzerinteraktion kann in ActionScript hinzugefügt werden, das letztlich die gesamte Steuerung der Animation realisiert und ausprogrammiert werden muss.

Insgesamt lässt sich sagen, dass DEViL ähnlich viele Animationsarten unterstützt wie Flash. Animationen können deklarativ durch Keyframes definiert werden, was in DEViL den diskreten Simulationsschritten entspricht. Adobe Flash ist animationsorientiert, d. h. ausgehend von einer Animation wird, falls nötig, eine Steuerung entworfen. Der Ansatz in DEViL ist simulationsorientiert, d. h. auf Basis einer Simulation wird eine Animation generiert, die im Wesentlichen das gewünschte Ergebnis abbildet und nur noch wenig durch die Simulation parametrisiert werden muss. Wie die Beispielimplementierungen gezeigt haben, reicht das Überschreiben von Animationsmustern häufig aus. Die Simulation muss nur in wenigen Sprachen Werte für die Animation vorgeben, z. B. wenn die Rotationsrichtung des Cursors in Logo berechnet werden muss.

Abschließend lässt sich sagen, dass die von DEViL generierten Editoren durchaus mit den anderen Systemen mithalten können. Vor allem kann eine Animation ohne Zusatzaufwand generiert werden. Dies leisten die anderen Systeme nicht.

### 7.4.4 Kontrolliertes Experiment

Um den Arbeitsaufwand für die Verhaltensspezifikation eines kleinen Editors abzuschätzen sollten vier Testkandidaten das Verhalten des Sokoban-Editors implementieren. Die Aufgabe lautete:

“In diesem Experiment sollen Sie das Verhalten für einen Sokoban Editor spezifizieren. Der Sokoban ist eine Spielfigur, die sich durch einen Benutzer innerhalb eines zweidimensionalen kachelbasierten Spielfeldes entlang der vier Hauptrichtungen Norden, Süden, Westen und Osten bewegen lässt. Das Spielfeld besteht aus einer Reihe von Wandkacheln, die das Spielfeld begrenzen und Hindernisse darstellen sowie einer Menge von Paketen und Zielpositionen für die Pakete, so genannte *Goals*. Ziel des Spiels ist es, alle Pakete auf die Goals zu bewegen. Die Pakete verschwinden, wenn sie auf ein Goal bewegt sind. Der Sokoban kann Pakete durch Schieben bewegen. Sind alle Pakete erfolgreich entfernt, so ist das Spiel gewonnen und es soll eine entsprechende Log-Ausgabe erzeugt werden. Testen Sie das Spiel wenn Sie fertig sind.”

Vorgegeben war ein lauffähiger Editor, der semantische Struktur und visuelle Darstellung des Spielfeldes und der Spielfiguren enthielt. Zu entwerfen war die DSIM-

Spezifikation, d. h. Simulationsstruktur, Ereignisse und Verhalten. Den Testkandidaten wurde Zugriff auf die DEViL-Dokumentationen sowie auf die implementierten Beispiele gegeben. Eine Vorgabe bzgl. einer Implementierungsstrategie oder eines systematischen Vorgehens wurde nicht gemacht. Alle Testkandidaten hatten im Rahmen der Projektgruppe mit DSIM und DEViL im Allgemeinen gearbeitet. Keiner der Probanden kannte das Sokoban-Spiel bisher.

In Listing 7.1 ist ein Ausschnitt aus der Verhaltensspezifikation mit den wichtigsten Ereignissen zu sehen. Das Ereignis `coordinateSokoban` berechnet die nächste Position des Sokoban. Es erhält die Richtung basierend auf der Tastatureingabe und bewegt den Sokoban auf das entsprechende Nachbarfeld sofern es frei ist. Ist es nicht frei, so kann es sein, dass sich dort ein Paket (`Box`) befindet. Dann wird versucht das Paket in die entsprechende Richtung zu bewegen (Ereignis `tryMoveBox`). Dies gelingt nur, wenn in der Bewegungsrichtung kein anderes Element, d. h. eine Wand oder eine andere Box, vorhanden ist. Die vollständige Spezifikation der Beispielimplementierung umfasst 157 LOC. Darin enthalten sind auch Mechanismen den Spielstand und das Level zu laden, die nicht Teil der Aufgabe waren. Die Probanden mussten etwa 75 Zeilen Code selbst entwerfen. Dazu mussten im Wesentlichen die Benutzerinteraktion und einige Nachbarschaftsfunktionen verwendet werden. Um herauszufinden, ob das Spiel gewonnen wurde, also keine Pakete mehr auf dem Spielfeld vorhanden sind, waren zwei Varianten möglich: entweder ein XPath-Ausdruck benutzen, der die Pakete zählt oder von Hand über alle Pakete mit dem `FOREACH`-Quantor iterieren und in einer Variable mitzählen. Beide Varianten sind gemessen an den LOC etwa gleich umfangreich.

Die zweite Aufgabe im Rahmen eines kontrollierten Experiments bestand darin, die Standardanimation für einen vorhandenen Editor zu ändern und dynamische Animationsobjekte einzusetzen. Dazu sollte der Pac-man Editor benutzt werden. Folgende Aufgabe wurde gestellt:

“Ändern Sie die Standardanimation für den Pac-man Editor so ab, dass der Pac-man bei der Bewegung zunächst in die Bewegungsrichtung gedreht wird und sich anschließend entsprechend bewegt. Wenn der Pac-man eine Powerpill frisst, soll diese zunächst kurz aufblinken bevor sie verschwindet. Wird der Pac-man durch eine der Geister-Figuren gefangen, so soll als dynamisches Animationsobjekt ein Totenkopf an der entsprechenden Stelle erscheinen, der sich wiederum leicht bewegt. Die Rotationswinkel bzw. die Rotationsrichtung des Pac-man sind bereits in den Variablen `angle` bzw. `clockwise` durch die Simulation vorberechnet.”

Um diese Aufgabe zu erledigen mussten die Probanden sieben Animationsmuster mit insgesamt 16 Kontrollattributen anwenden. Diese Aufgabe mag als sehr klein erscheinen, ist jedoch durchaus realistisch, denn wie in Tabelle 7.4.1 zu sehen ist, reicht die Standardanimation in der Regel aus. Nur kleine Anpassungen sind noch nötig.

```

1 LOOP {
2   FOREACH sokoban IN [Sokoban]{
3     ...
4     IF(ACTION_EVENT(inputRight) AND (ACTION_EVENT_VALUE(inputRight) == VLInt(3))){
5       FIRE coordinateSokoban(sokoban, 3) @ TIME_DIRECT;
6     }
7     ...
8   }
9 }
10 EVENTS {
11   coordinateSokoban(Sokoban sokoban, VLInt direction){
12     Tile go = NEIGHBOUR_TILE(default, sokoban.tile, direction);
13     Tile from = sokoban.tile;
14     IF(SIZE(go.item) == 0) {
15       Item i = REMOVE(from.item, FIRST);
16       INSERT(go.item, i, FIRST);
17       FIRE computeRotation(sokoban,direction) @ TIME_DIRECT;
18     } ELSE {
19       Item i = GET(go.item, FIRST);
20       IF(INSTANCEOF(i OF [Box])) {
21         Box currentBox = CAST(i TO [Box]);
22         FIRE tryMoveBox(currentBox, direction)@ TIME_DIRECT;
23       }
24     }
25     FIRE gameLostOrWon();
26   }
27   tryMoveBox(Box box, VLInt direction) {
28     Tile boxIsOn = box.tile;
29     VLInt x = TILE_GETX(default, boxIsOn);
30     VLInt y = TILE_GETY(default, boxIsOn);
31
32     Tile next =NEIGHBOUR_TILE(default, boxIsOn, direction);
33
34     Sokoban sokoban = #[0]Sokoban;
35     Tile tile = sokoban.tile;
36     IF(NOTNULL(next)){
37       IF(SIZE(next.item) == 0) {
38         Item box = REMOVE(boxIsOn.item, FIRST);
39         INSERT(next.item, box, FIRST);
40         REMOVE(tile.item, FIRST);
41         INSERT(boxIsOn.item, sokoban, FIRST);
42       } ELSE {
43         Item item = GET(next.item, FIRST);
44         IF(INSTANCEOF(item OF [Goal])) {
45           REMOVE(tile.item, FIRST);
46           REMOVE(box);
47           INSERT(boxIsOn.item, sokoban, FIRST);
48         }
49       }
50     }
51   }
52 }

```

Listing 7.1: Ausschnitt aus der Sokoban Verhaltensspezifikation

## KAPITEL 7. EVALUATION

<b>Sokoban Verhaltensimplementierung</b>		<b>Durchschnitt</b>	<b>Standard- abweichung</b>
Die Aufgabe war ... (1) sehr schwer ... (5) sehr einfach		4,0	0,7
Die Bearbeitung der Aufgabe empfand ich als ... (1) sehr unangenehm ... (5) sehr angenehm		4,5	0,8
Bearbeitungszeit (in min.)		53	14
<b>Pac-man Musteranwendung</b>			
Die Aufgabe war ... (1) sehr schwer ... (5) sehr einfach		4,0	0,8
Die Bearbeitung der Aufgabe empfand ich als ... (1) sehr unangenehm ... (5) sehr angenehm		5,0	0,0
Wie zufrieden sind Sie mit der Animation ... (1) sehr unzufrieden ... (5) sehr zufrieden		5,0	0,0
Bearbeitungszeit (in min.)		17	3

**Abbildung 7.12:** Kontrolliertes Experiment: Implementierung der Verhaltensspezifikation in Sokoban und Musteranwendung in Pac-man

In Abbildung 7.12 sind die Ergebnisse der Aufgaben zu sehen. Alle Probanden haben es geschafft, beide Aufgaben zu lösen. Dabei war die Herangehensweise sehr unterschiedlich. Die Verhaltensspezifikation wurde zum Teil zunächst recht chaotisch implementiert. Dies hat sich auch in einer Lösung widergespiegelt, die mit 241 LOC mehr als dreimal so groß wie die Musterlösung war. Trotzdem wurde der Schwierigkeitsgrad der Aufgabe als recht niedrig und die Bearbeitung der Aufgabe als sehr angenehm empfunden. Dies ist insbesondere erstaunlich, da kontrollierte Experimente einer Prüfungssituation ähneln und Probanden Aufgaben somit eher als schwierig einstufen.

Abbildung 7.12 zeigt, dass die Bearbeitungszeit im Schnitt bei 53 Minuten lag, wobei der beste Proband bereits nach 37 Minuten einen voll funktionsfähigen Editor vorweisen konnte. Der Zeitbedarf scheint für die Aufgabe zunächst recht hoch auszufallen, allerdings ist darin bereits die Zeit für das Testen und für das Nachschlagen in der Dokumentation oder in Beispielen enthalten.

Ein interessanter Aspekt war, dass alle Probanden zunächst das Verhalten vollständig implementieren wollten. Zwischentests oder Phasen in denen die Übersetzbarkeit der Spezifikation getestet wurde, wurden ausgelassen. Ich vermute, dass dies an der Einfachheit der Spezifikationssprache liegt und an der Selbstverständlichkeit mit der diese benutzt wurde. Einige Konstrukte der Sprache wurden jedoch recht umständlich angewendet, was zwar nicht zu Fehlern führte, jedoch die Spezifikation zum Teil unnötig kompliziert werden ließ.

Die Anwendung der Animationsmuster wurde sehr schnell erledigt und die Aufgabe als angenehm empfunden. Die Probanden waren durchweg zufrieden mit der generierten Animation. Erstaunlich war, dass nur einer der Probanden bereits mit dynamischen Animationsobjekten gearbeitet hatte. Die restlichen Probanden mussten die Anwendung erst erlernen, was jedoch aufgrund der Ähnlichkeit zu den herkömmlichen Animationsmustern recht einfach und schnell geschah. Hier zeigt sich die gute Konsistenz der entworfenen Animationskomponente.

In einer abschließenden kleinen Diskussion konnten die Probanden noch mitteilen, was ihnen besonders schwer oder besonders einfach gefallen ist. Dabei empfand ein Proband die Anwendung der Ereignisse als schwierig, da diese keine Rückgabewerte haben und ein Zwischenspeichern von berechneten Werten in der Simulationsstruktur als zu umständlich erachtet wurde. Dies ist ein Aspekt der mir auch in der Projektgruppe aufgefallen ist: Ereignisse werden als Methoden betrachtet und nur ungern in die Ereigniswarteschlange eingefügt. Statt dessen werden Ereignisse häufig mit dem `TIME_DIRECT`-Konstrukt ausgeführt. Ich vermute, dass ein Grund in der Ausbildung liegt, da hier Java als erste Programmiersprache eingesetzt wird. Trotzdem legen die Antworten aus Tabelle 7.7 nahe, dass sich die Probanden alle gut mit dem ereignisbasierten Konzept auseinandergesetzt haben und sich auch sicher in der Anwendung sind.

Ein weiterer Proband bemängelte, dass ihm häufig nicht klar sei, mit welchem Knotentyp, d. h. SUB-Knoten, Objektknoten oder (erweiterter) Attributknoten, er momentan in DSIM arbeitet. Ich vermute, dass sich dies darauf zurückführen lässt, dass das semantische Modell bereits vorgegeben war und die Struktur somit nicht wirklich verinnerlicht wurde. Trotzdem konnte der Proband in relativ kurzer Zeit die Verhaltensspezifikation implementieren, deshalb vermute ich, dass das Problem als größer empfunden wurde als es tatsächlich war. In Abbildung 7.13 sind einige abschließende Fragen, die nach Beendigung des kontrollierten Experiments gestellt wurden aufgelistet. Wie bereits vermutet, sind die Antworten wesentlich besser, als in der Projektgruppe. Insbesondere die Einschätzung von DSIM ist durchweg positiv. Meine Vermutung, dass sich der algorithmische Schwierigkeitsgrad der Verkehrssimulation auf die Einschätzung der Spezifikationssprache negativ auswirkt, wurde somit bestätigt.

## 7.5 Geschwindigkeit

Schmidt hat bereits die Geschwindigkeit generierter Editoren untersucht [101]. Er hat dabei die Reaktionszeit verschiedener Repräsentationen untersucht. Er konnte zeigen, dass die Reaktionszeit zur Sichtaktualisierung für realistische Sprachen und Programmgrößen akzeptabel, d. h. unter 300 Millisekunden, liegt. Dabei war zu beobachten, dass die Aktualisierungszeit der Sichten proportional zu ihrer struktu-

Frage		Durchschnitt	Standardabweichung
Die Anwendung von XPath Ausdrücken zum Matchen von Strukturen verstehe ich zu ... (1) null Prozent ... (5) hundert Prozent		4,7	0,4
Wie Änderungen an der Simulationsstruktur vorgenommen werden und welche Operationen dazu existieren ist mir ... (1) völlig unklar ... (5) völlig klar		4,8	0,4
Statische Fehlermeldungen in DSIM sind ... (1) schwer verständlich ... (5) leicht verständlich		3,3	0,4
DSIM schränkt mich bei der Verhaltensspezifikation ... (1) stark ein ... (5) überhaupt nicht ein		4,3	0,8
DSIM schränkt mich beim Entwurf einer maßgeschneiderten Simulationsstruktur ... (1) stark ein ... (5) überhaupt nicht ein		4,8	0,4
Den Umgang mit DSIM empfinde ich als ... (1) unangenehm ... (5) angenehm		4,8	0,4
Die Erweiterung der Simulationsstruktur verstehe ich zu ... (1) null Prozent ... (5) hundert Prozent		4,5	0,5
Die Anwendung von Pfadausdrücken zur Navigation in der Simulationsstruktur verstehe ich zu ... (1) null Prozent ... (5) hundert Prozent		4,5	0,5

Abbildung 7.13: Kontrolliertes Experiment: Abschließende Fragen

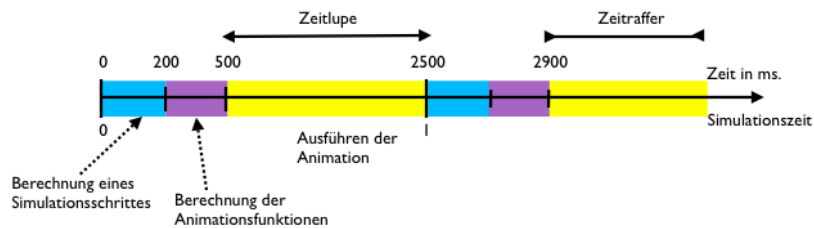
rellen Größe ist. Das heißt, je umfangreicher eine Sicht ist, umso länger dauert es, die Sicht darzustellen. Allgemein kann man sagen, dass die Darstellung je Strukturknoten im Schnitt 3,5 bis 5 Millisekunden in Anspruch nimmt. Diese Zeit steigt, wenn im Mengennmuster die Überlappungsfreiheit berechnet wird auf ca. 7 Millisekunden an. Wird das Baummuster verwendet, so liegt die Aktualisierungszeit je Strukturknoten bei ca. 17 Millisekunden. Der Grund ist, dass zur Darstellungsberechnung beim Baummuster das Layoutwerkzeug *Dot* [39] benutzt wird. *Dot* wird als separater Prozess aufgerufen, außerdem besteht die Layoutberechnung in *Dot* aus einem mehrstufigen Verfahren, somit ist ein erhöhter Rechenaufwand erforderlich.

An den implementierten Beispielsprachen ist zu sehen, dass die Simulations- und Animationsgeschwindigkeit hauptsächlich von der Strukturgröße abhängig ist.

Abbildung 7.14 verdeutlicht die Abbildung von Simulation, Animationsberechnung und Animation auf die Realzeit. Ein Simulations- bzw. Animationsschritt besteht dabei aus der Berechnung der Simulation, die durch die Verhaltensspezifikation in DSIM implementiert wurde und der Berechnung der Animationsfunktionen. Diese werden anhand der ausgeführten Änderungsoperationen und den Baumdurchläufen berechnet, die notwendig sind um die graphische Attribute aller Objekte berechnen zu können.

Die Animation an sich läuft in allen implementierten Sprachen mit mindestens





**Abbildung 7.14:** Abbildung von abstrakter Simulations- und Animationszeit auf Echtzeit

24 Frames pro Sekunde, also flüssig ab. Dies ist sogar der Fall wenn mehr als 50 Objekte gleichzeitig animiert werden. Zusätzlich hat der Anwender hier noch die Option die Animation durch einen skalaren Faktor zu beschleunigen oder zu verlangsamen, d. h. er kann die Animation in Zeitlupe oder in Zeitraffer ablaufen lassen.

Problematisch und als unangenehm wird die Animation empfunden, wenn die Anteile von Simulations- und Animationsberechnung zu groß werden. Burnett et al. nennen in [14] eine Grenze von etwa 300 Millisekunden. Wird diese Zeit überschritten, so ist eine deutliche Unterbrechung in der gesamten Animation zu bemerken. Dies tritt bei den implementierten Beispielanwendungen nur bei Sprachen auf, die kachelbasierte Darstellungen nutzen. Beispiele sind Pac-man, Sokoban und die Verkehrssimulation. Durch die sehr kompakte Repräsentation lassen sich auf sehr engem Raum viele Strukturobjekte platzieren. Die Simulation muss hier für bestimmte Operationen recht häufig den gesamten Strukturbaum durchlaufen und der Attributauswerter benötigt zur Berechnung der Darstellung ebenfalls recht lange.

In Tabelle 7.2 habe ich einige Editoren mit unterschiedlicher Strukturgröße simuliert und die Zeit gemessen, die für Simulations- und Animationsfunktionsberechnung benötigt wird. Als Vertreter für nicht-kachelbasierte Strukturen habe ich ein recht komplexes Petri-Netz mit 51 Strukturknoten genommen. Die Berechnung von Simulation und Animation hat im Schnitt 127 Millisekunden gedauert. In der Animation war nahezu keine Unterbrechung erkennbar. Als Beispiel für kachelbasierte Sprachen habe ich die Pac-man und die Verkehrssimulation genommen. Ich habe festgestellt, dass sich Pac-man noch sehr gut auf einem recht großen und realistischen Feld (18x10 Kacheln) spielen lässt. Dies ist erstaunlich, da die Berechnung für Simulation und Animation mit im Schnitt zusammen ca. 550 Millisekunden schon deutlich über der 300 Millisekunden Schwelle liegt. Benutzerinteraktion wirkt sich trotzdem noch unmittelbar aus. Hier wird es erst ab einer Grenze von ca. 500 Strukturknoten problematisch.

Die Verkehrssimulation ist strukturell wesentlich komplexer. Hier wirken sich Geschwindigkeitsprobleme bereits wesentlich früher negativ aus und es kommt zu Wartepausen zwischen den Animationen. Der Grund dafür ist, dass die Simulation

**Tabelle 7.2:** Geschwindigkeiten der Berechnung von Simulation und Animation einiger Editoren in Abhängigkeit zur Strukturgröße (in ms.)

	Knotenanzahl	Feldgröße (in Kacheln)	Simulation	Animationsberechnung
Petri-Netze	51		93	34
Pac-man	164	10x10	388	43
Pac-man	255	18x10	494	66
Pac-man	536	25x15	843	133
Verkehrssimulation	1575	10x10	6157	643

wesentlich aufwändiger ist. Regeln, die der Benutzer der Verkehrssimulation selbst definieren kann, müssen ausgewertet und auf die Straßenkarte angewendet werden. Um dem Benutzer bei der Anwendung der Regeln Konstruktionszeit zu ersparen, werden die Regeln intern und während der Simulation gedreht, so kann eine allgemeine Rechts-vor-Links Regel durch eine Regelkonstruktion abgedeckt werden, obwohl sie prinzipiell für alle vier Himmelsrichtungen entworfen werden müsste. Außerdem werden viele XPath-Ausdrücke ausgewertet, was durch das Traversieren des Strukturbaums ebenfalls sehr kostenintensiv ist.

Trotzdem gibt es auch hier ein subjektives Empfinden für Geschwindigkeit. Da der Benutzer bereits weiß, dass die Simulation sehr aufwändig ist, wird er auch toleranter gegenüber der langsamen Ausführung.

Der Hauptgrund für die Geschwindigkeitsprobleme bei kachelbasierten Darstellungen liegt in der Brücke zwischen C und Tcl. Der Attributauswerter, der die Darstellung berechnet ist in C realisiert, das DEViL-Frontend jedoch in Tcl.<sup>1</sup> Somit muss für jedes graphische Primitiv der Tcl-Interpreter aufgerufen werden. Versuche wie Batch-Processing, Interpreter-Optimierung oder eine Konstruktion der Primitive direkt in OpenGL haben keine signifikanten Geschwindigkeitsvorteile gebracht. Eine interessante Variante könnte sein, die Differenz zwischen zwei adjazenten

---

<sup>1</sup>Die Wahl von Tcl ist eine historische Konsequenz. Tcl ist eine sehr einfache Skriptsprache und es existierte bereits sehr früh eine Anbindung an C, die notwendig war um die in C generierten Attributauswerter des Eli Systems zu nutzen.

Darstellungen zu berechnen. Gerade bei kachelbasierten Darstellungen ändern sich häufig nur wenige Zellen, eine komplette Neuberechnung der Darstellung ist nur selten nötig. Allerdings ist die Differenzberechnung in Strukturbäumen sicherlich nicht einfach zu realisieren und wird ebenfalls einigen Overhead erzeugen. Hinzu kommt noch eine weitere strukturelle Ebene, die die Komplexität des Gesamtsystems erhöht.

Auf Dauer scheint somit ein Wechsel der Plattform, evtl. zu Eclipse, unumgänglich. Dies würde die langsame Skriptsprache Tcl umgehen und es wäre auch der Einsatz von Threads denkbar. So könnte während der Animationsphase bereits ein Thread den nächsten Simulationsschritt berechnen. Tcl erlaubt keine Threads, deshalb schied diese Variante in der aktuellen Implementierung aus.



## 8 Resümee und Ausblick

Ziel dieser Arbeit war es, das Generatorsystem DEViL so zu erweitern, dass anspruchsvolle Simulationen und Animationen für visuelle Sprachen spezifiziert werden können. Ich habe dabei versucht, vorhandene und bewährte Konzepte zu nutzen und zu erweitern. Dies hat den Vorteil, dass auch Anwender, die bisher nur Sprachen ohne Simulation und Animation entworfen haben, leicht Simulationen spezifizieren können. Ein weiteres Kriterium war es, das Sprachspektrum nicht einzuschränken. Ich konnte zeigen, dass alle Struktureditoren die DEViL generieren kann, auch mit Simulation und Animation ausgestattet werden können. Insbesondere stellt es kein Problem dar, wenn eine Sprache nachträglich erweitert werden soll. Ich habe visuelle Sprachen auf ihre Eigenschaften untersucht und kategorisiert. Die Vielfalt der implementierten und hier vorgestellten Sprachen ist groß. Sowohl Sprachen mit universeller und inhärenter Zustandsübergangsnotation können einfach simuliert und automatisch animiert werden, als auch Sprachen die eine Erweiterung der Simulationsstruktur benötigen. Wird der Strukturunterschied zwischen Simulation und Animation noch größer, so kann DEViLs Strukturkopplungsprinzip benutzt werden und eine Animation ebenfalls automatisch abgeleitet werden.

Auf Seiten der Simulation habe ich existierende Konzepte zur Modellspezifikation wiederverwendet um Simulationsstrukturen erstellen zu können. Ich habe mich hierbei am klassenbasierten Entwurf von DSSL orientiert. Die bekannten Pfadausdrücke können sowohl in der semantischen als auch in der Simulationsstruktur angewendet werden.

Zur Spezifikation von Verhalten habe ich eine auf visuelle Sprachen maßgeschneiderte Sprache entwickelt, die es leicht ermöglicht auf Strukturobjekte der Sprache zuzugreifen und diese zu verändern. Mit dem integrierten XPath-Auswerter können so sogar Sprachen implementiert werden, deren Verhalten in Konstrukten der Sprache selbst definiert ist und durch den Anwender beliebig kombiniert werden können. Die Untersuchung von existierenden Simulationsmethodiken führte zur Implementierung von speziellen Funktionen, die den Kontext eines Sprachkonstrukts kennen. So können auch transportorientierte Simulationen leicht entworfen werden.

Zur Manipulation der Simulationsstruktur steht eine kleine aber wohldefinierte Schnittstelle - die Änderungsoperationen - bereit. An diese Schnittstelle knüpft die Animation an. Die Simulationsstruktur sowie die Änderungsoperationen bilden

die Baumfragmentstruktur aus der automatisch eine Animation abgeleitet werden kann. Diese reicht, wie die Evaluation gezeigt hat, in vielen Fällen bereits aus um eine anspruchsvolle Animation zu erreichen. Ist dies nicht der Fall, so kann die Animation deklarativ durch *Animationsmuster* angepasst werden. Dies geschieht durch einfaches Überschreiben der Standardanimation. Ich habe dazu eine Bibliothek von Animationsmustern entworfen, die nahezu beliebig miteinander kombinierbar sind und durch Kontrollattribute einfach und schnell zu parametrisieren sind. Die Animationsfunktionen der Musterbibliothek basieren auf Recherchen in der Literatur und deren Anwendung in anderen Systemen, wie z. B. Authoringwerkzeugen. Durch die schmale Anknüpfungsstelle zwischen Simulation und Animation ist die Animation zusätzlich eine formal korrekte Abbildung und kann sogar zu Debugging-Zwecken benutzt werden. Reicht die Bibliothek an Animationsmustern nicht aus, können weitere Muster einfach implementiert werden. Sie müssen lediglich an eine Simulationsänderungsoperation gebunden werden und eine bestimmte Erweiterungsstelle ausfüllen.

Bisher wurde in anderen existierenden Generatorsystemen der Animation nur wenig Aufmerksamkeit gewidmet, sie war quasi nur Beiwerk zur Simulation. In dieser Arbeit konnte ich jedoch zeigen, dass eine automatisch generierte Animation wenig Zusatzaufwand auf Spezifikationsebene bedeutet, jedoch sehr eindrucksvoll ist. Dies ist besonders vorteilhaft, wenn im Rapid-Prototyping Prozess schnell Ergebnisse erzielt werden müssen. Die hier vorgestellte Animation ist einzigartig in der Form, dass Strukturobjekte nicht nur graphisch interpoliert werden, sondern sich vollständig ineinander morphen lassen, d. h. ihr gesamtes Layout verändert sich flüssig. Dies ist nicht nur der Fall für Objekte, die durch die Änderungsoperationen modifiziert wurden, sondern auch für quasi unbeteiligte Objekte, die sich aufgrund von Seiteneffekten durch andere Objekte verändern. Dies ist in bisher keinem anderen existierenden Generatorsystem in dieser Form implementiert.

Weitere Konzepte wie statische bzw. dynamische Animationsobjekte sowie Mechanismen zur Positionierung von Sprachkonstrukten innerhalb der Zeichenfläche oder entlang von Pfaden sowie *Easing* für sich bewegende Objekte unterstützen den Sprachspezifizierer eine anspruchsvolle Animation zu entwerfen. So können Animationen erreicht werden, die auf allen Ebenen der Cox-Klassifikation [22] einzuordnen sind: Es sind direkte Abbildungen der Simulation auf die Animation möglich (*direct*), es können aber auch zusätzliche Eigenschaften berechnet und in der Animation eingeblendet werden (*structural*). Durch eingebaute Logging- und Auswertungsmechanismen können Meta-Informationen berechnet werden (*synthesized, analytical*) und die automatisch generierte weiche und zeitlich beliebig darstellbare Animation hilft beim Systemverständnis (*explanatory*).

Auch Staskos Prinzipien [112] an Animationssysteme werden erfüllt: So sind die Animationsmuster eine einfache deklarative Spezifikationsmethode, bei der der Anwender vom Implementierungsniveau abstrahieren kann (*Ease of Design*). Anima-

---

tionen können automatisch generiert werden und helfen somit im *Rapid-Prototyping*. Sollte die Musterbibliothek nicht ausreichen, so kann der Anwender selbst beliebige Muster implementieren. Er muss dazu das Muster nur registrieren, es an eine Simulationsänderungsaktion binden und eine bereits vorhandene Erweiterungsstelle implementieren (*Ability to fine-tune*).

Eine weitere Anwendung für die Animationsmuster könnte direkt bei der Konstruktion eines visuellen Programms sein. Benutzerinteraktionen könnten so direkt in Animationen umgewandelt werden, d. h. ein Einfügen eines Programmkonstrukts durch den Anwender könnte unmittelbar eine Animation auslösen. Die momentan diskreten Änderungen würden so plausibler dargestellt werden.

Erweiterungspotential gibt es sicherlich noch im Bereich der Darstellung für visuelle Sprachen, die einen großen Bereich der Zeichenfläche einnehmen. Zooming zu interessanten Stellen oder die Bewegung der Kamera in eine bestimmte Position sind hier Aspekte, die eventuell auch durch Animationsmuster spezifiziert werden können. Andere Visualisierungsvarianten [37] wie isometrische oder *Fish-eye* Ansichten könnten hier ebenso eingesetzt werden.

Auf Seiten der Verhaltensspezifikation ist es auch denkbar, dass eine visuelle Sprache eingesetzt werden kann. Ich denke aber, dass diese durch die Pfadausdrücke recht textlastig werden würde. Eine sinnvolle Erweiterung sehe ich im Bereich von Sprachen bei denen der Benutzer selbst Regeln definieren kann, die dann auf einen anderen Teilbaum angewendet werden. Hier ist die bisherige textlastige Spezifikation recht aufwändig. Eine visuelle Form der Spezifikation durch *Programming-by-Example* wie in Agentsheets oder anderen auf Regeln basierenden Systemen ist hier sicherlich vorteilhaft. Die Evaluation hat gezeigt, dass die integrierten Debugging Möglichkeiten helfen, eine korrekte Simulation zu erreichen. Interessant könnte noch die Erweiterung von Ko und Myers [60] sein: Hier können Benutzer interaktiv "Warum"-Fragen zu Verhalten und Sprachkonstrukten stellen. Eine Vorstufe ist mit dem DSIM-Debugger und den Overlays bereits integriert. Die Overlays geben Auskunft darüber, warum ein Sprachkonstrukt animiert wird und was der Auslöser war.

In der letzten Zeit ist eine neue Tendenz im Bereich der visuellen Sprachen zu erkennen: der Einsatz von 3-dimensionalen Repräsentationen. Erste Sprachvarianten sind schon seit längerer Zeit bekannt [75, 120, 52]. Durch die Verwendung von OpenGL und die mittlerweile große Verbreitung an schneller Graphikhardware könnten so einige typische Probleme visueller Sprachen, wie der Platzbedarf, zumindest teilweise umgangen werden können. In *Agentcubes* [93], einer Erweiterung von Agentsheets mit dem kachelbasierte Simulationen durch visuelle Regeln programmiert werden können, wird dazu die zweidimensionale Darstellung in der dritten Dimension "aufgebläht". Das Resultat ist ein dreidimensionales Spielfeld, in dem der Benutzer die Spielfigur aus der Spielerperspektive steuert. Auf Seiten der

Spezifikation müssen dazu nur die graphischen Primitive in einem speziellen Editor graphisch angepasst werden, sodass sie eine dreidimensionale Kontur bekommen.

GEF3D [123] ist ein Framework um auf Basis von Eclipse Editoren mit 2-, 2,5- und 3-D-Unterstützung zu generieren. Dabei werden herkömmliche zweidimensionale Diagramme auf eine Ebene projiziert (2,5-D) und danach ebenfalls zu dreidimensionalen Diagrammen aufgebläht. Beziehungen zwischen Diagrammen, unter Umständen auch verschiedenartigen Diagrammen, können so dreidimensional dargestellt werden (*inter-model*-Perspektive). Auf Seiten der Programmvisualisierung wird in *CodeCity* [126] eine 3-dimensionale Darstellung benutzt um Softwarestrukturen und Größen darzustellen. Um solche Darstellungen herleiten zu können muss im Wesentlichen nur die Darstellung graphischer Primitive angepasst werden. Die zu Grunde liegende Editorstruktur muss nicht angepasst werden. So können auch vorhandene Editoren ohne große Änderungen in eine dreidimensionale Darstellung überführt werden. Eine ähnliche Variante auf das DEViL-System übertragen, könnte Teilbäume der editierbaren Struktur als 3-dimensionale Teile einer Sprache auffassen. Beziehungen, also Referenzattribute, würden dann die Beziehungen im Raum darstellen.

In Anbetracht der in der Evaluation besprochenen Geschwindigkeitsprobleme und neuer Darstellungsmöglichkeiten ist die Zukunft des DEViL-Systems eher auf der Eclipse-Plattform zu sehen. Cebeci hat in seiner Diplomarbeit [17] gezeigt, dass die Konzepte von DEViL zu großen Teilen übernommen werden können. Plugins für Eclipse und die große Entwicklergemeinschaft sprechen dafür, dass 3-dimensionale Repräsentationen aber auch Simulation und Animation ebenfalls in Eclipse realisiert werden können.



# Literaturverzeichnis

- [1] Price B. A., Baecker, R. M., and Small I. S. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
- [2] Apache Software Foundation. *Xalan-C++*, 2010. <http://xml.apache.org/xalan-c/>.
- [3] R. Baecker. Sorting out sorting, 1981. ACM SIGGRAPH, Videotape.
- [4] R. Baecker and I. Small. Animation at the interface. In B. Laurel, editor, *The Art of Human-Computer Interface Design*, pages 251–267. Addison-Wesley, 1990.
- [5] Ronald Baecker and Aaron Marcus. Printing and publishing c programs. In Marc H. Brown John Stasko, John Domingue and Blaine A. Price, editors, *Software Visualization*, pages 45–61. MIT Press, 1998.
- [6] Roswitha Bardohl. GenGed: A generic graphical editor for visual languages based on algebraic graph grammars. In *1998 IEEE Symposium on Visual Languages*, pages 48–55, September 1998.
- [7] Mordechai Ben-ari. Program visualization in theory and practice. *Informatik/Informatique*, 2:8–11, 2001.
- [8] Jacques Bertin. *Semiology of graphics*. University of Wisconsin Press, 1983.
- [9] E. Biermann, C. Ermel, J. Hurrelmann, and K. Ehrig. Flexible visualization of automatic simulation based on structured graph transformation. In *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008.*, pages 21–28, Sept. 2008.
- [10] Ronald Baecker Blaine Price and Ian Small. An introduction to software visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, pages 3–27. MIT Press, 1998.
- [11] Marc H. Brown and John Hersherberger. Program auralization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, pages 137–145. MIT Press, 1998.
- [12] M.H. Brown and R. Sedgewick. A system for algorithm animation. In *Proceedings of ACM SIGGRAPH*, pages 177–186, 1984.
- [13] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley, Aug. 2003.

- [14] Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, 1995.
- [15] E. Börger, E. Riccobene, and J. Schmid. Capturing requirements by abstract state machines: The light control case study. *Journal of Universal Computer Science*, 6(7):597–620, 2000.
- [16] Paul Carlson, Margaret Burnett, and Jonathan Cadiz. A seamless integration of algorithm animation into a declarative visual programming language. In *Proceedings of Advanced Visual Interfaces (AVI'96)*, 1996.
- [17] Yascha Cebeci. Entwurf eines Prototyps zur Generierung von graphischen Struktureditoren für das Eclipse Framework. Diplomarbeit, Universität Paderborn, 2009.
- [18] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In *ACM Symposium on User Interface Software and Technology*, pages 45–55, 1993.
- [19] Kai Chen, Janos Sztipanovits, Sherif Abdelwahed, and Ethan Jackson. Semantic anchoring with model transformations. In *European Conference on Model Driven Architecture -Foundations and Applications (ECMDA-FA)*, Nuremberg, Germany, November 2005.
- [20] Bastien Chopard, Alexandre Dupuis, Re Dupuis, and Pascal Luthi. A cellular automata model for urban traffic and its application to the city of geneva. In *Proceedings of Traffic and Granular*, pages 154–168. Springer, 1997.
- [21] MetaCase Consulting. *MetaEdit+ User's Guide*, 2002. <http://www.metacase.com/fs.asp?vasen=vasen.html&paa=products.html>.
- [22] K.C. Cox and G.-C. Roman. Abstraction in algorithm animation. In *1992 IEEE Workshop on Visual Languages*, pages 18–24, September 1992.
- [23] Bastian Cramer. Generierung von graphischen Struktureditoren aus visuellen Spezifikationen. Diplomarbeit, Universität Paderborn, 2005. <http://ag-kastens.uni-paderborn.de/paper/cramer2005.pdf>.
- [24] Ole-Johan Dahl and Kristen Nygaard. Simula - an algol-based simulation language. In *Communications of the ACM*, 1966.
- [25] Richard C. Davis and James A. Landay. Forms of expression for designing visual languages for animation. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 327–328, Washington, DC, USA, 2005. IEEE Computer Society.
- [26] Beman Dawes and David Abrahams. Boost Homepage, 2010. <http://www.boost.org/>.
- [27] Juan de Lara and Hans Vangheluwe. Using atom3 as a meta-case tool. In *4th International Conference on Enterprise Information Systems (ICEIS)*, pages 642–649, April 2002.

- [28] Sergey Dmitriev. Language oriented programming: The next programming paradigm. JetBrains 'onBoard' electronic monthly magazine, 2004. <http://www.onboard.jetbrains.com/is1/articles/04/10/lop>.
- [29] Jürgen Ebert, Roger Süttenbach, and Ingar Uhe. Meta-case in practice: A case for kogge. In Antoni Olivé and Joan Pastor, editors, *Advanced Information Systems Engineering*, volume 1250 of *Lecture Notes in Computer Science*, pages 203–216. Springer Berlin / Heidelberg, 1997.
- [30] Sol Efroni, David Harel, and Irun R. Cohen. Reactive animation: Realistic modeling of complex dynamic systems. *Computer*, 38:38–47, 2005.
- [31] Claudia Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, Universität Berlin, 2006.
- [32] Claudia Ermel, Roswitha Bardohl, and Hartmut Ehrig. Generation of animation views for petri nets in genged. In Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg, and Herbert Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 83–108. Springer, 2003.
- [33] Robert Esser and Jörn W. Janneck. Moses - a tool suite for visual modeling of discrete-event systems. In *HCC '01: Proceedings of the IEEE 2001 Symposium on Human Centric Computing Languages and Environments (HCC'01)*, page 272, Washington, DC, USA, 2001. IEEE Computer Society.
- [34] Osman Balci et. al. A Web-Based Visual Simulation Architecture. *International Journal of Modelling and Simulation*, 29(2):137–148, 2009.
- [35] Andrew Hale Feinstein and Hugh M. Cannon. Constructs of simulation evaluation. *Simulation and Gaming*, 33(4):425–439, 2002.
- [36] Paul A. Fishwick. *Simulation Model Design and Execution*. Prentice Hall, 1995.
- [37] Vitaly Friedman. Data visualization: Modern approaches, 2010. <http://www.smashingmagazine.com/2007/08/02/data-visualization-modern-approaches/>.
- [38] R. Heckel G. Engels. Graph transformation and visual modeling techniques. *Bulletin of the European Association for Theoretical Computer Science, EATCS*, (71), June 2000.
- [39] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.
- [40] E. P. Glinert. Towards “second generation” interactive, graphical programming environments. In *Proc. of IEEE 2nd Fall Comp. Conf.*, pages 292–299, 1987.
- [41] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

- [42] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.
- [43] Yuri Gurevich. Evolving algebras 1993: Lipari guide. pages 9–36, 1995.
- [44] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.
- [45] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.
- [46] Kevin J. Healy and Richard A. Kilgore. Silk : A java-based process simulation language. In *Winter Simulation Conference*, pages 475–482, 1997.
- [47] Magnus Lie Hetland. *Practical Python*. Apress, Berkeley, CA, USA, 2002.
- [48] Adobe Systems Incorporated. Adobe flash cs professional, 2010. <http://www.adobe.com/products/flash/>.
- [49] Uwe Kastens Jan Wolter, Bastian Cramer. Animation of tile-based games automatically derived from simulation specifications. In *CoRTA'2010, 4th Compilers, Programming Languages, Related Technologies and Applications*, September 2010.
- [50] Jörn Wilhelm Janneck. *Syntax and Semantics of Graphs*. PhD thesis, ETH Zürich, 2000.
- [51] Patrick W. Jordan. *An Introduction to Usability*. Taylor & Francis, 1998. (paper) 0-7484-0794-4 (cloth).
- [52] K. Kahn and V. A. Saraswat. Complete visualizations of concurrent programs and their executions. In *1990 IEEE Workshop on Visual Languages*, October 1990.
- [53] Uwe Kastens. LIGA: A language independent generator for attribute evaluators. Technischer Bericht, Reihe Informatik tr-ri-89-63, Universität Paderborn Fachbereich Mathematik-Informatik, 1989.
- [54] Uwe Kastens and Carsten Schmidt. VL-Eli: A generator for visual languages. In *Proceedings of Second Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, number 2027 in Electronic Notes in Theoretical Computer Science, Grenoble, France, 2002. Band 65, Elsevier Science Publishers.
- [55] Yoshikazu Kato, Etsuya Shibayama, and Shin Takahashi. Effect lines for specifying animation effects. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 27–34, Washington, DC, USA, 2004. IEEE Computer Society.
- [56] Jonathan M. Kaye, Ph. D, David Castillo, Sandy Clark, James Gish, Marissa Maiella, Jonathan M. Kaye, David Castillo, Jaimie Wetzels, Maura Theriault, Fair Huntoon, Sarena Douglass, Mary Ellen Black, and Larry Main. Castillo flash mx for interactive simulation, 2002.

- [57] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming, 1997.
- [58] Mikael Kindborg and Kevin McGee. Comic strip programs: Beyond graphical rewrite rules.
- [59] Donald E. Knuth. *The art of Computer Programming - Fundamental Algorithms*. Addison-Wesley, 1973.
- [60] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the java whyline. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 1569–1578, New York, NY, USA, 2009. ACM.
- [61] Wolfgang Kreutzer, Jane Hopkins, and Marcel van Mierlo. Simjava—a framework for modeling queueing networks in java. In *WSC '97: Proceedings of the 29th conference on Winter simulation*, pages 483–488, Washington, DC, USA, 1997. IEEE Computer Society.
- [62] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.
- [63] John Lasseter. Principles of traditional animation applied to 3d computer animation. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 35–44, New York, NY, USA, 1987. ACM.
- [64] Meyers Lexikonredaktion. *Duden Informatik, Ein Fachlexikon für Studium und Praxis, 3. Auflage*. Dudenverlag, 2001.
- [65] Henry Lieberman. The tv turtle a logo graphics system for raster displays. In *The papers of the ACM symposium on Graphic languages*, pages 66–72, New York, NY, USA, 1976. ACM.
- [66] Microsoft. Die AsmL Sprache, 2006. <http://research.microsoft.com/fse/asml/>.
- [67] Mark Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, August 2002.
- [68] Mark Minas. Generating meta-model-based freehand editors. In Albert Zündorf and Dániel Varró, editors, *Proc. of the 3rd International Workshop on Graph Based Tools (GraBaTs'06), Natal (Brazil), September 21-22, 2006, Satellite event of the 3rd International Conference on Graph Transformation*, volume 1 of *Electronic Communications of the EASST*, 2006.
- [69] Mark Minas and Johann Gottschall. Specifying animated diagram languages. In *Proc. International Workshop on Theory of Visual Languages (TVL'97), Capri, Italy, September 1997*.
- [70] Sougata Mukherjea and John T. Stasko. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *ACM Trans. Comput.-Hum. Interact.*, 1(3):215–244, 1994.

- [71] B. A. Myers. Taxonomies of visual programming and program visualization. *Visual Languages and Computing*, 1(1):97–123, 1990.
- [72] Brad A. Myers, Robert C. Miller, Rich Mcdaniel, and Alan Ferreny. Easily adding animations to interfaces using constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '96)*, pages 119–128, 1996.
- [73] Wolfgang Müller. *Executable Graphics for VHDL-Based Systems Design*. PhD thesis, Universität Paderborn, 1996.
- [74] K. Nagel and M. Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique I*, 2:2221–2229, December 1992.
- [75] Marc-Alexander Najork and M. Kaplan. Programming in three dimensions. *Journal of Visual Languages and Computing*, 7:219–242, 1994.
- [76] A. Narayanan, L. Ford, D. Manuel, D. Tallis, and M. Yazdani. Language animation. 2007.
- [77] Bonnie A. Nardi and Craig L. Zarter. Beyond models and metaphors: Visual formalisms in user interface design. *Journal on Visual Languages and Computing*, 4(1):5–33, 1993.
- [78] Donald A. Norman and Stephen W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1986.
- [79] Nuno Ernesto Salgado Oliveira. Improving Program Comprehension Tools for Domain Specific Languages. Master thesis, University of Minho, 2010.
- [80] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.
- [81] Volker Paelke. *Design of Interactive 3D Illustrations*. PhD thesis, Universität Paderborn, 2002.
- [82] O. Park and R. Hopkins. Instructional conditions for using dynamic visual displays : a review. *Instructional Science*, 21:427–449, 1993.
- [83] C. Dennis Pegden. Introduction to siman. In *Proceedings of the 1983 Winter Simulation Conference*, 1983.
- [84] C. Dennis Pegden, Robert E. Shannon, and Randall P. Sadowski. *Introduction to Simulation Using SIMAN*. McGraw Hill, 1995.
- [85] Robert M. O’Keefe Peter C. Bell. Visual interactive simulation: A methodological perspective. In *Annals of Operations Research*, pages 321–342, Dezember 1994.
- [86] Marian Petre. Why looking isn’t always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995.
- [87] Emmanuel Pietriga. A toolkit for addressing hci issues in visual language environments. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual*

- Languages and Human-Centric Computing*, volume 0, pages 145–152, Los Alamitos, CA, USA, 2005. IEEE Computer Society.
- [88] Istvan Rath, David Vago, and Daniel Varro. Design-time simulation of domain-specific models by incremental pattern matching. In *VLHCC '08: Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 219–222, Washington, DC, USA, 2008. IEEE Computer Society.
- [89] István Ráth and Dániel Varró. Challenges for advanced domain-specific modeling frameworks. In *International Workshop on Domain Specific Program Development (DSPD 2006)*, Nantes, France, July 2006.
- [90] Horst Rathbauer. *Angewandte Simulation mit GPSS World für Windows*. Logos Verlag, Berlin, Deutschland, 2003.
- [91] Arend Rensink, Alexander Dotor, Claudia Ermel, Stefan Jurack, Ole Kniemeyer, Juan de Lara, Sonja Maier, Tom Staijen, and Albert Zündorf. Ludo: A case study for graph transformation tools. pages 493–513. 2008.
- [92] A. Repenning and T. Sumner. Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer*, 28(3):17–26, 1995.
- [93] Alexander Repenning and Andri Ioannidou. Agentcubes: raising the ceiling of end-user development in education through incremental 3d. In *IEEE Symposium on Visual Languages and Humancentric Computing, 2006*. IEEE Press, 2006.
- [94] Stephen V. Rice and Ana Marjanski. The simscript iii programming language for modular object-oriented simulation. In *Proceedings of the 2005 Winter Simulation Conference*, 1988.
- [95] Lloyd P. Rieber. On visual formalisms. *Animation as feedback in a computer-based simulation: Representation matters*, 44:5–22, March 1996.
- [96] Chell A. Roberts and Yasser M. Dessouky. An overview of object-oriented simulation. In *Simulation*, pages 272–279. IEEE Computer Society, September 2001.
- [97] Gruia-Catalin Roman and Kenneth C. Cox. Program visualization: The art of mapping programs to pictures. In *In Proceedings of the 14th International Conference on Software Engineering*, pages 412–420. ACM Press, 1992.
- [98] Gruia-Catalin Roman, Kenneth C. Cox, C. Donald Wilcox, and Jerome Y. Plun. Pavane: a system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.
- [99] Robert G. Sargent. The use of graphical models in model validation. In *WSC '86: Proceedings of the 18th conference on Winter simulation*, pages 237–241, New York, NY, USA, 1986. ACM.
- [100] Stefan Schiffer. *Visuelle Programmierung - Grundlagen und Einsatzmöglichkeiten*. Addison Wesley, 1998.

- [101] Carsten Schmidt. *Generierung von anspruchsvollen Struktureditoren*. PhD thesis, Universität Paderborn, 2006.
- [102] Carsten Schmidt, Bastian Cramer, and Uwe Kastens. Usability evaluation of a system for implementation of visual languages. In *VLHCC '07: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 231–238, Washington, DC, USA, 2007. IEEE Computer Society.
- [103] Carsten Schmidt and Uwe Kastens. Implementation of visual languages using pattern-based specifications. *Software - Practice and Experience*, 33:1471–1505, December 2003.
- [104] Carsten Schmidt and Christian Schindler. *Muster-basierte Generierung von Struktur-Editoren für visuelle Sprachen*. Diplomarbeit, Universität Paderborn, Germany, January 2000.
- [105] Carsten Schmidt and Michael Thies. PaderWAVE Abschlussbericht. Technical report, Universität Paderborn, 2005. <http://ag-kastens.uni-paderborn.de/lehre/paderwave/material/PaderWAVE-Abschlussbericht.pdf>.
- [106] Thomas J. Schriber. *An introduction to simulation using GPSS/H*. John Wiley & Sons, Inc., New York, NY, USA, 1991.
- [107] Herbert D. Schwetman. Introduction to process-oriented simulation and csim. In *Proceedings of the 1990 Winter Simulation Conference*, 1990.
- [108] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold, 1988.
- [109] D. C. Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhauser, Basel, 1977.
- [110] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++: An aspect-oriented extension to c++. In *In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 53–60, 2002.
- [111] IEEE Software Staff. What's so bad about rule-based programming? *IEEE Software*, 8(5):103, 1991.
- [112] John T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 307–314, New York, NY, USA, 1991. ACM.
- [113] John T. Stasko and D. Scott McCrickard. Real clock time animation support for developing software visualisations. *Australian Computer Journal*, 27(4):118–128, 1995.
- [114] J.T. Stasko. TANGO: A Framework and System for Algorithm Animation. Technical report, Brown University, 1989. Report No. CS-89-30.
- [115] Torsten Strobl and Mark Minas. Implementing an animated lambda-calculus. In *Workshop on Visual Languages and Logic, satellite of 2009 IEEE Symposium on*



- Visual Languages and Human-Centric Computing, Corvallis, OR, USA, September 20, 2009*, volume 510 of *CEUR Workshop Proceedings*, 2009.
- [116] Torsten Strobl and Mark Minas. Specifying and generating editing environments for interactive animated visual models. In Jochen Küster and Emilio Tuosto, editors, *Proceedings of the 9th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010), March 20-21 2010, Paphos, Cyprus*, volume 29 of *Electronic Communications of the EASST*, 2010.
- [117] Torsten Strobl, Mark Minas, Andreas Pleuß, and Arnd Vitzthum. From the behavior model of an animated visual language to its editing environment based on graph transformation. In *Preproceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)*, 2010.
- [118] Eugene Syriani and Hans Vangheluwe. Programmed graph rewriting with devs. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 136–151. Springer Berlin / Heidelberg, 2008.
- [119] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *AGTIVE*, pages 481–488, 1999.
- [120] F. Van Reeth and E. Flerackers. Three-dimensional graphical programming in cael. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 389–391, 24-27 1993.
- [121] Hans Vangheluwe, Juan de Lara, and PJ Mosterman. An introduction to multi-paradigm modelling and simulation. 2002.
- [122] M. J. Varanda and Pedro Rangel Henriques. Visualization / animation of programs based on abstract representations and formal mappings. In *HCC'01 - 2001 IEEE Symposia on Human-Centric Computing Languages and Environments*. IEEE, September 2001.
- [123] Jens von Pilgrim and Kristian Duske. Gef3d: a framework for two-, two-and-a-half-, and three-dimensional graphical editors. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 95–104, New York, NY, USA, 2008. ACM.
- [124] G. M. Vose and G. Williams. Labview: Laboratory virtual instrument engineering workbench. *BYTE*, pages 84–92, September 1986.
- [125] Jos Warmer and Anneke Kleppe. *The Object Constraint Language. Precise Modeling with UML*. Addison-Wesley, November 1999.
- [126] Richard Wettel and Michele Lanza. Codacity: 3d visualization of large-scale software. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 921–922, New York, NY, USA, 2008. ACM.
- [127] Philipp Wissneth. Generierung von Strukturkopplungen für visuelle Entwicklungsumgebungen. Diplomarbeit, Universität Paderborn, 2009.

- [128] Jan Wolter. Simulation und Animation eines regelbasierten Spieles. Studienarbeit, Universität Paderborn, 2009.
- [129] XML Path Language (XPath) Version 1.0, W3C Recommendation, November 1999. <http://www.w3c.org/TR/xpath>.

# Abbildungsverzeichnis

2.1	Harels Higraphs . . . . .	11
2.2	Framework zur Klassifizierung von Softwarevisualisierung . . . . .	16
2.3	Effect Lines . . . . .	19
2.4	Klassifikation von Animation . . . . .	21
2.5	Animationsarten . . . . .	23
2.6	Ereignisorientierte Simulation . . . . .	26
2.7	Prozessorientierte Simulation . . . . .	26
2.8	Blockmodell in SIMAN . . . . .	28
2.9	Operationen auf TANGO Datentypen . . . . .	33
2.10	Polka . . . . .	36
2.11	Pictorial Janus Syntax . . . . .	38
2.12	Pictorial Janus Beispiel . . . . .	38
2.13	Visualisierung in Alma . . . . .	40
2.14	MetaEdit+ . . . . .	42
2.15	Spezifikation in GenGed . . . . .	45
2.16	Spezifikation von Simulation und Animation in DiaMeta . . . . .	47
2.17	VL Definition in Moses . . . . .	48
2.18	Definition eines Petri-Netz Interpreters in Moses . . . . .	49
2.19	Gekoppeltes DEVS-Modell in AToM <sup>3</sup> . . . . .	51
2.20	DEViL Spezifikationskonzept . . . . .	55
2.21	Strukturen in DEViL . . . . .	56
2.22	Spezifikation von Struktur und Repräsentation . . . . .	58
2.23	Grammatikabbildung . . . . .	61
2.24	Abbildung auf die Repräsentationsstruktur . . . . .	64
2.25	Pfadausdruck . . . . .	65
3.1	Daten- vs. Codeorientierte Simulation . . . . .	72
3.2	Visuelle Sprachen vom Typ 1 . . . . .	79
3.3	Visuelle Sprachen vom Typ 2 . . . . .	81
3.4	Visuelle Sprachen vom Typ 3 (gekoppelt) . . . . .	83
3.5	Visuelle Sprachen vom Typ 3 . . . . .	84
3.6	Klassifikation visueller Sprachen . . . . .	85
4.1	Erweiterter Spezifikationsprozess . . . . .	89
4.2	Einbettung des Simulators . . . . .	94
4.3	Zeitliches Verhalten des Simulators . . . . .	95

4.4	DSIM funktional betrachtet . . . . .	99
4.5	Interaktionsdialog . . . . .	109
5.1	Animationskonzept . . . . .	115
5.2	Graphische Repräsentation der Simulation . . . . .	118
5.3	Ausdehnung einer generischen Zeichnung . . . . .	120
5.4	Interpolationsaktionen . . . . .	121
5.5	AVP . . . . .	124
5.6	Animationsmuster . . . . .	125
5.7	Berechnung der Interpolation . . . . .	128
5.8	Beispiele statischer und dynamischer Animationsobjekte . . . . .	130
5.9	Positionierungsproblem . . . . .	131
5.10	Positionierungsproblem: Definition von Crosslines . . . . .	132
5.11	Informationsverlust in der Simulation . . . . .	133
5.12	Generierter Struktureditor . . . . .	135
6.1	Ein Ereignisgraph . . . . .	143
6.2	DSIM Debugger . . . . .	143
6.3	Animationssicht mit Metainformationen . . . . .	144
6.4	Auslastungsgraph . . . . .	145
6.5	Darstellung von Leerlaufzeiten in der Simulation . . . . .	146
7.1	Usability im Verlauf der Zeit . . . . .	152
7.2	Beispielimplementierungen . . . . .	157
7.3	Beispielimplementierungen . . . . .	158
7.4	Beispielimplementierungen . . . . .	159
7.5	Likert Skala . . . . .	166
7.6	Aktueller Kenntnisstand . . . . .	167
7.7	Fragen zu Simulation und Simulationsspezifikation . . . . .	168
7.8	Fragen zu Simulation und Simulationsspezifikation . . . . .	168
7.9	Fragen zur Animation . . . . .	169
7.10	Fragen zu Debuggingmöglichkeiten . . . . .	170
7.11	Definition eines Petri-Netz Interpreters in Moses . . . . .	171
7.12	Kontrolliertes Experiment . . . . .	176
7.13	Kontrolliertes Experiment: Abschließende Fragen . . . . .	178
7.14	Abbildung von abstrakter Simulations- und Animationszeit auf Echtzeit	179

# Tabellenverzeichnis

7.1	Spezifikationsaufwand in LOC für Editoren mit Simulations- und Animationsunterstützung. Zahlen in Klammern sind LOC für C-Funktionen. . . . .	164
7.2	Geschwindigkeiten der Berechnung von Simulation und Animation einiger Editoren in Abhängigkeit zur Strukturgröße (in ms.) . . . . .	180



# Listings

2.1	Spezifikation mit GPSS aus [90]	29
2.2	Spezifikation mit SIMSCRIPT III	30
2.3	Spezifikation in TANGO.	34
2.4	Spezifikation in POLKA.	35
2.5	Spezifikation von Visualisierungsregeln in Alma.	40
2.6	Spezifikation von Ersetzungsregeln in Alma.	40
2.7	Ausschnitt aus der Basisstruktur	63
2.8	Eine abgeleitete Struktur	63
4.1	Spezifikation der Simulationsstruktur für Petri-Netze in DSIM	100
4.2	Ausschnitt der Spezifikation der Simulationsstruktur für Ludo in DSIM	101
4.3	Ausschnitt der Spezifikation des LOOP-Blocks für Petri-Netze in DSIM	103
4.4	Ausschnitt einer Spezifikation eines LOOP-Blocks mit Schleifenvarianten und Ausdrücken	104
4.5	Spezifikation von Ereignissen für einen Petri-Netz-Editor	104
4.6	Spezifikation eines Ereignisses für den Pac-man-Editor	106
4.7	Spezifikation von Kachelzugriffsfunktionen im Pac-man Editor	106
4.8	Log-Ausgaben	107
4.9	Benutzerinteraktion	108
4.10	Mustererkennung mit XPath	110
5.1	Annotation von DSIM durch Animationsereignisse	116
5.2	Anwendung der AVPs	122
5.3	Spezifikation eines dynamischen Animationsobjekts	129
5.4	Spezifikation eines statischen Animationsobjekts	129
5.5	Informationsverlust zwischen Simulation und Animation in einer Petri-Netz Spezifikation	132
5.6	Zugriff auf die Verursacherliste mit dem POSITION-Makro, dass das erste Element der Verursacherliste auswählt.	134
5.7	Zugriff auf das auslösende Ereignis innerhalb einer Animationsspezifikation	134
6.1	Ein Aspekt für die Warteschlangensimulation	147
7.1	Ausschnitt aus der Sokoban Verhaltensspezifikation	175