Bachelorarbeit

Simulation und Animation eines regelbasierten Spieles

Jan Wolter
Matrikelnummer: 6384444
E-Mail: jwolter@mail.uni-paderborn.de

Paderborn, den 29. September 2009

Betreuer:	Dipl. Inform. Bastian Cramer
Gutachter:	Prof. Dr. Uwe Kastens
	Prof. Dr. Stefan Böttcher

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Paderborn, den 29. September 2009

Jan Wolter

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **3**

2.1. Regelbasierte Spiele 3
2.1.1. Spiel des Lebens 6
2.1.2. Vier gewinnt 7
2.2. Das Spiel Pac-Man 8
2.3. DEViL 9
2.3.1. Abstrakte Struktur 11
2.3.2. Visuelle Darstellung 13
2.3.3. Semantische Analyse und Code-Generierung 16
2.3.4. Simulation und Animation 17

3. Sprachentwurf **22**

3.1. Der Editor 22
3.1.1. Überblick 22
3.1.2. Das Modell der abstrakten Struktur 24
3.1.3. Die visuelle Darstellung 26
3.2. Die Simulation und die Animation 28
3.2.1. Überblick 28
3.2.2. Die Simulation 30
3.2.3. Die Animation 34

4. Aspekte der Implementierung **36**

4.1. Initialisierung 36
4.2. Konsistenzbedingungen 38
4.3. Synchronisation 40
4.4. Strategien 41
4.4.1. Zufall 41
4.4.2. Schrittweise Annäherung 42
4.4.3. Bergsteigen 42
4.5. Aspektorientierte Programmierung 44
4.5.1. Beschreibung des Paradigmas 44

4.5.2. Einsatz bei der Implementierung 46

5. Evaluation **48**
5.1. Komplexitäts- und Performanzuntersuchungen 48
5.2. Regelbasierte Syntax 51
5.2.1. Klassifikation und Verallgemeinerung der C-Funktionen . . 51
5.2.2. Mapping von Bezeichnern 53
5.2.3. Syntaxvorschläge 54

6. Resümee und Ausblick **56**

A. Konkretisierte Syntaxvorschläge **58**
A.1. Syntax der Abfragen und Aufruf der C-Funktionen 58
A.2. Regelbasierte Syntax 59
A.2.1. Menge der `RuleStmt`-Produktionen 60
A.2.2. Beispiele für Regeln in DSIM 60
A.3. Zwei Beispiele 61
A.3.1. Geister-Strategien 61
A.3.2. `coordinatePacman`-Ereignis 62

B. Anmerkungen zur Spezifikation **63**

Abbildungsverzeichnis **64**

Quelltextverzeichnis **65**

Tabellenverzeichnis **66**

Literaturverzeichnis **67**

1. Einleitung

Visuelle Sprachen gewinnen bei der Modellierung von Softwaresystemen zunehmend an Bedeutung. Der große Vorteil visueller Sprachen ist, dass sie sich Metaphern einer bestimmten Domäne bedienen können. So ist es mittels grafischer Editoren auch Nicht-Experten möglich, Software für spezielle Anwendungsgebiete zu entwerfen. Ein bekanntes Beispiel hierfür ist die *Unified Modelling Language* (UML). Es existieren Editoren, die es ermöglichen Diagramme nach den von UML vorgegebenen standardisierten Konstrukten zu entwickeln, die dann durch Code-Generierung in eine objektorientierte Programmiersprache transformiert werden können.

Mittlerweile gibt es nicht nur Editoren, die sich auf eine statische Darstellung von Diagrammen und visuellen Sprachelementen beschränken. Die Entwicklung geht dahin visuelle Sprachen zu simulieren und zu animieren. So lässt es sich realisieren, innerhalb desselben Diagramms eine Abfolge modifizierender Schritte darzustellen, die das Diagramm umgestalten. Ein gutes Beispiel dafür ist ein Diagramm eines Zustandsdiagramms, bei dem die Zustände, je nachdem ob sie gerade aktiv oder inaktiv sind, ihre Farbe wechseln und so dem Benutzer ihren aktuellen Status visualisieren. Im Zusammenhang mit Zustandsdiagrammen kann der Vorteil von animierten Repräsentationen, das Verständnis von nebenläufigen Systemen zu fördern, besonders ausgespielt werden. Sehr hilfreich kann auch die Animation von Petri-Netzen sein, bei der der Flug der Token entlang der Transition visualisiert wird.

Um Editoren für visuelle Sprachen schnell und effektiv zu erstellen, existieren Generatoren, die diese aus formalen Spezifikationen einer visuellen Sprache generieren. In der Fachgruppe „Programmiersprachen und Übersetzer“ der Universität Paderborn wird mit *DEViL* (Development Environment for Visual Languages) ein solches Generatorsystem entwickelt, welches auch die Simulation und die Animation von visuellen Sprachen erlaubt. Die mit DEViL erstellten Struktureditoren erlauben es, gemäß den Regeln der zugrunde liegenden Spezifikation, grafische Strukturen zu erstellen. Diese lassen sich, sofern es sich um korrekte Strukturen handelt, was mittels semantischer Analyse überprüft wird, in eine textbasierte Sprache übersetzen. Außerdem ist es möglich, regelgerechte Strukturen zu simulieren und zu animieren.

Eine Sprache für regelbasierte Spiele ist im Kontext von DEViL ganz neu und stellt sich, speziell im Hinblick auf die Simulation und die Animation, als sehr spannend dar. In dieser Arbeit soll ein grafischer Editor zur Erstellung eines regelbasierten Spieles erstellt werden. Die Wahl als Beispiel für ein solches regelbasiertes Spiel ist auf das Kultspiel *Pac-Man* gefallen. So muss es zunächst möglich sein, mit dem Editor Spielwelten für Pac-Man aufzubauen. In jeder der erstellten Spielwelten soll im Anschluss das Pac-Man-Spiel simuliert und geeignet animiert werden. Dafür ist es zunächst notwendig eine visuelle Sprache, die Spielfelder von Pac-Man-Spielen beschreibt, zu entwickeln. Danach muss eine adäquate Simulation mit zusätzlicher Animation implementiert werden. Dabei ist besonders interessant, dass die Simulation von Benutzerstimuli, die den Pac-Man steuern, abhängig ist. So gibt es neben den kontinuierlichen Ereignissen der Geister auch modale Ereignisse von außen. Dieser Aspekt ist gegenüber den bisher in DEViL realisierten Simulationen neu.

Die für das Spiel notwendigen Regeln müssen mit einer Simulationsspezifikationssprache definiert werden. Da diese bisher keine Form der regelbasierten Anweisungen unterstützt, werden Vorschläge gemacht, wie eine Erweiterung um eine regelbasierte Syntax aussehen kann.

In der vorliegenden Arbeit werden zunächst regelbasierte Spiele und insbesondere das Spiel Pac-Man genauer beschrieben. Außerdem wird das Spezifikationskonzept von DEViL ausführlich erläutert. Danach wird der Sprachentwurf für den implementierten Editor vorgestellt. Dies teilt sich in zwei Teile: Den Entwurf des Editors zur Erstellung der Spielwelten und die darauf aufbauende Konzeption der Simulation und der Animation. Kapitel 4 befasst sich mit besonderen Aspekten der Implementierung, so zum Beispiel den Strategien, die die im Spiel vorkommenden Geister verfolgen. In Kapitel 5 wird das realisierte Spiel evaluiert und aus den dort vorgestellten Ergebnissen eine Möglichkeit vorgestellt, wie eine regelbasierte Syntax realisiert werden kann. Kapitel 6 enthält eine kurze Zusammenfassung der Ergebnisse und einen Ausblick auf mögliche Erweiterungen.

2. Grundlagen

In diesem Kapitel sollen die Grundlagen beschrieben werden, die nötig sind, um einen grafischen Editor zur Erstellung von regelbasierten Spielen mit dem Generatorsystem DEViL zu implementieren.

Im ersten Abschnitt werden die allgemeinen Ideen regelbasierter Spiele und zwei Beispiele vorgestellt. Im darauf folgenden Teil wird das regelbasierte Spiel Pac-Man beschrieben, welches im Rahmen dieser Arbeit implementiert werden soll.

Im letzten Abschnitt dieses Kapitels wird das Generatorsystem DEViL ausführlich vorgestellt. Zunächst wird die abstrakte Struktur und die Darstellung einer visuellen Sprache vorgestellt, bevor auf die semantische Analyse und Code-Generierung eingegangen wird. Diese Konzepte haben sich schon seit mehreren Jahren bei der Spezifikation verschiedenster Editoren bewährt. Abschließend folgt eine ausführliche Beschreibung der erst seit kurzem vorhandenen Möglichkeit visuelle Sprachen zu simulieren und zu animieren.

2.1. Regelbasierte Spiele

In diesem Kapitel soll zunächst das mit den regelbasierten Spielen verwandte Konzept der regelbasierten Systeme vorgestellt werden. Danach wird am Beispiel des Entwicklungswerkzeugs *Agentsheets* [Agea] gezeigt, wie Regeln im Kontext von regelbasierten Spielen formuliert werden. Danach folgen in zwei Unterkapiteln Beispiele für regelbasierte Spiele.

Wie der Name schon sagt, handelt es sich bei regelbasierten Spielen um Spiele, die auf Regeln basieren. Dies ist genauso viel- wie nichtssagend, denn ein Spiel ohne Regeln ist fast ein Widerspruch in sich. Um zu verstehen, was genau dahinter steckt, bietet sich das formale Konzept der regelbasierten Systeme [FH] an.

Regelbasierte Systeme sind auf Probleme anwendbar, bei denen das ganze Wissen über den Problembereich in Form von Wenn-Dann Regeln ausgedrückt werden kann. Gibt es zu viele Regeln, kann das System schnell unübersichtlich und schwer zu handhaben sein. Um ein regelbasiertes System für ein gegebenes Problem zu beschreiben, bedarf es folgender drei Dinge:

Wissensbasis Eine Menge von Fakten, die alles relevante im Anfangszustand des Problems beschreibt.

Regelmenge Sie sollte alle Aktionen umfassen, die innerhalb der Möglichkeiten des Problems auftreten können. Irrelevante Sachverhalte sollten weggelassen werden.

Inferenzmaschine Sie bestimmt eine mögliche Lösung, indem sie die Wissensbasis und die Regelmenge miteinander verknüpft.

Im Game Programming Wiki [Gam] findet sich ein anschauliches Beispiel für das eben beschriebene. Es geht um eine Maschine, die das Verhalten einer Maus imitiert. In der Wissensbasis findet sich die Aussage „Käse riecht gut“ und in der Regelmenge befindet sich genau eine Regel: „Wenn `X` gut riecht und `X` sichtbar ist, dann laufe in Richtung `X`.“ Die Inferenzmaschine für die Maus wird auf Grundlage des Wissens und der entsprechenden Regel in Richtung Käse laufen.

Solche Art von Entscheidungen auf Grundlage bestimmten Wissens müssen auch in vielen Spielen gefällt werden. Ein bekanntes Entwicklungswerkzeug zur Erstellung von Spielwelten samt aller notwendiger Regeln und anschließender Simulation ist Agentsheets. Agentsheets wurde Anfang der neunziger Jahre von Alexander Repenning entwickelt und seitdem kontinuierlich erweitert. Die mit Agentsheets erstellten Spielwelten [RC93] sind kachelbasiert und jede dieser Kacheln kann einen Agenten enthalten. Ein Agent wird als eine feingranulare, autonome Einheit mit Verhalten verstanden. Im Gegensatz zu vielen agentenbasierten Systemen, spielen die Agenten in Agentsheets nicht die Rolle von Vermittlern zwischen dem Benutzer und der Anwendung. Stattdessen sind die Agentsheets-Agenten die Anwendung. Das Hauptaugenmerk einer mit Agentsheets erstellten Anwendung liegt auf der Interaktion zwischen dem Benutzer und den Agenten.

Jeder Agent wird durch fünf verschiedene Eigenschaften beschrieben: ein bildliches Symbol, Sensoren, Aktoren, Verhalten und Zustände. Sensoren aktivieren Methoden des Agenten, die Verhalten beschreiben. Dies wird durch den Benutzer z.B. durch das Klicken auf den Agenten ausgelöst. Für alle eingebauten Agentenklassen gibt es ein Standardverhalten, das die Reaktion für alle Sensoren beschreibt. Die Aktoren werden benutzt, um Nachrichten zwischen Agenten zu verschicken. Dies geschieht entweder über Kacheln des Spielfeldes oder über direkte Verbindungen. Ein Zustand beschreibt die Beschaffenheit eines Agenten, die durch das bildliche Symbol visualisiert wird. So wird z.B. ein Agent, der einen elektrischen Schalter beschreibt, je nach Zustand (geöffnet oder geschlossen) durch zwei verschiedene Symbole dargestellt.

Wie mit Agentsheets regelbasierte Programmierung stattfindet, lässt sich in [GIL+95] nachlesen, wo die regelbasierte Programmierumgebung *LEGOsheets*, die

Abbildung 2.1.: LEGOsheets-Arbeitsfläche und Regel-Editor [GIL+95].

es erlaubt einen programmierbaren LEGO Baustein zu simulieren und zu handhaben, beschrieben wird. Der programmierbare LEGO Baustein wird in ein kleines, aus LEGO Bausteinen zusammengebautes, Fahrzeug integriert und dessen Motoren und Sensoren werden mit ihm verbunden. Mit LEGOsheets, deren Arbeitsfläche den programmierbaren Baustein visualisiert, lassen sich Programme definieren, die auf den Baustein geladen werden können und so das Fahrzeug steuern. So lassen sich der Arbeitsfläche Motoren und Sensoren hinzufügen, die erst mit ihren physikalischen Pendants verbunden werden, um danach ihr Verhalten nach einem regelbasierten Ansatz festzulegen. Die Regeln in LEGOsheets bestehen aus drei Teilen: einem optionalen initialen Wert, einer Menge von Bedingungen und einer Standard-Aktion. Bei den Bedingungen handelt es sich um Wenn-Dann Konstrukte von denen in jedem Ausführungsschritt genau eines ausgeführt wird. Ist keine der Bedingungen erfüllt, kommt die Standard-Aktion zur Anwendung. Die Regel aus Abbildung 2.1 lautet wie folgt: „If: TOUCH3 = 1 Then: -8. Default: REFLECT1-200." Wenn der dritte Berührungssensor aktiviert ist, soll der Motor mit Geschwindigkeit -8 laufen, ansonsten ergibt sich die Geschwindigkeit aus einem Ausdruck in Abhängigkeit von dem ersten Reflektionssensor.

In [RCT95] und [Rep95] wird das Erstellen von Regeln weiter in den visuellen Kontext eingebettet, indem die Erstellung von Regeln nicht mehr textbasiert,

wie noch in dem LEGOsheets-Beispiel, sondern durch Anordnung von grafischen Symbolen erfolgt. In beiden Publikationen geht es um die Modellierung einer visuellen Sprache zur Simulation des Straßenverkehrs. In [RCT95] wird eine visuelle Sprache zur Erstellung von Wenn-Dann Regeln eingeführt. Die so erstellten Regeln lesen sich als eine Folge von kleinen Bildern, die durch eine bestimmte Semantik Regeln der Art „Wenn meine Darstellung ein nach Osten gerichtetes Auto ist und die Straße einen Knick von Osten nach Süden macht, dann fahre nach Osten und danach nach Süden." beschreiben können. In [Rep95] wird eine visuelle Sprache vorgestellt, mit der sich zum einen Karten, bestehend aus Straßen, Ampeln und Autos, erstellen lassen und zum anderen visuelle Regeln formulieren lassen, um z.B. beschreiben zu können, wie sich Autos fortbewegen sollen. Um dies alles so allgemeingültig wie möglich zu halten, wird jedes Straßenelement mit Verbindungspfeilen versehen, sodass nur zueinander passende Elemente miteinander verbunden werden können.

2.1.1. Spiel des Lebens

Das *Spiel des Lebens*[1] wurde 1970 vom englischen Mathematiker John Horton Conway entworfen und ist das bekannteste Beispiel für *zelluläre Automaten*. Der Verlauf des Spieles wird ausschließlich durch seine Startkonfiguration bestimmt und bedarf danach keiner weiteren Benutzereingabe.

Das Spielfeld besteht aus Zeilen und Spalten und ist im Idealfall unendlich groß. Jede Zelle des matrixartigen Spielfeldes kann genau einen von zwei Zuständen annehmen, entweder ist sie *tot* oder *lebendig*. Zu Beginn des Spieles platziert der Spieler lebendige Zellen auf dem Spielfeld und legt so die Startkonfiguration fest. Jede Zelle hat genau acht Nachbarzellen, mit der sie in jedem Spielschritt nach vier einfachen Regeln interagiert:

1. Jede lebendige Zelle mit weniger als zwei lebendigen Nachbarn stirbt an Vereinsamung.

2. Jede lebendige Zelle mit mehr als drei lebendigen Nachbarn stirbt an Überbevölkerung.

3. Jede lebendige Zelle mit zwei oder drei lebendigen Nachbarn überlebt in die nächste Runde.

4. Jede tote Zelle mit genau drei lebendigen Nachbarn wird in der nächsten Runde lebendig.

[1] engl. Conway's Game of Life. Siehe auch [Köl00].

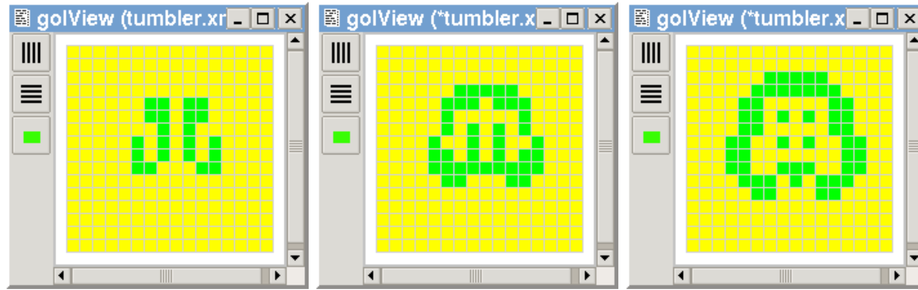

Abbildung 2.2.: Simulationsfolge aus dem Spiel des Lebens.

Mit DEViL wurde für das Spiel des Lebens ein Editor generiert. In Abbildung 2.2 ist eine Simulationsfolge daraus abgebildet. Die lebendigen Zellen sind in grün, die toten in gelb dargestellt. Außerdem sieht man jeweils links die Toolbar-Knöpfe, mit denen sich neue Zeilen und Spalten sowie initial lebendige Kacheln einfügen lassen.

2.1.2. Vier gewinnt

Vier gewinnt [All88] ist ein Zweipersonen-Spiel, welches auf einem, zu Beginn leeren, Spielfeld mit sieben Spalten und sechs Zeilen gespielt wird. Jeder der Spieler besitzt anfangs 21 Spielsteine. Die Menge der Steine des einen Spielers sind gelb und die des anderen sind rot. Die 6×7 Matrix des Spielfeldes wird von oben mit Spielsteinen gefüllt. Die Spieler wählen abwechselnd eine Spalte des Spielfeldes aus, in die ein Spielstein geworfen wird. Dieser fällt dann in die tiefstmögliche Zeile, die noch nicht von einem anderen Spielstein belegt ist. Gewonnen hat derjenige Spieler, der zuerst eine Linie aus vier seiner Spielsteine, entweder in horizontaler, vertikaler oder diagonaler Richtung, gebildet hat. In Abbildung 2.3 hat der Spieler mit den gelben Spielsteinen gewonnen, da diese eine zusammenhängende Linie in horizontaler Richtung bilden.

Das Aufstellen einer vollständigen Regelmenge für Vier gewinnt ist sehr komplex, wie Victor Allis in [All88] gezeigt hat. Exemplarisch sollte allerdings folgende Regel in jeder Implementierung von Vier gewinnt zur Anwendung kommen: „Kann der Gegner seinen roten Spielstein in Zelle x setzen, um so zu gewinnen, dann setze einen gelben Spielstein in Zelle x“. Allis hat nachgewiesen, dass der Spieler, der das Spiel eröffnet, auch gegen die beste Strategie des Gegners, immer das Spiel gewinnen kann. Daraus lässt sich folgende Regel ableiten: „Wenn das Spielfeld leer ist, dann werfe den Spielstein immer in die mittlere Spalte“.

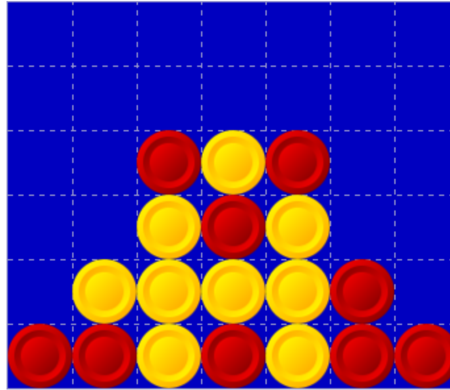

Abbildung 2.3.: Der Spieler mit den gelben Spielsteinen hat gewonnen.

2.2. Das Spiel Pac-Man

Pac-Man[2] ist ein sehr bekanntes Arcade-Computerspiel[3], welches 1980 von Toro Iwatani für das Unterhaltungsunternehmen Namco unter dem Originalnamen *Puck Man* entwickelt wurde. Ein Jahr später wurde es in den USA unter dem leicht abgewandelten Namen Pac-Man lizenziert und wurde zu einem der populärsten Computerspiele der achtziger Jahre. Aufgrund der großen Beliebtheit wurde es viele Male für Heimcomputer, Spielekonsolen oder Pocket-PCs nachprogrammiert und weiterentwickelt. Abbildung 2.4 zeigt ein Bildschirmfoto der originalen Pac-Man-Version von Namco.

Pac-Man wird klassischerweise im Einzelspielermodus gespielt, bei dem der menschliche Spieler die Aufgabe hat, den Pac-Man durch das Labyrinth zu manövrieren und dabei möglichst viele *Pillen* zu essen, ohne von den Geistern gefangen zu werden. Die Pillen sind zu Beginn des Spieles in jedem begehbaren Feld vorhanden und werden durch das Erreichen eines Feldes durch Pac-Man gegessen. Insgesamt gibt es vier verschiedene Geister, die von der Mitte des Spielfeldes aus starten und es sich zur Aufgabe gemacht haben Pac-Man zu fangen. Dies ist erfolgreich, wenn sie das Feld betreten in dem sich Pac-Man gerade befindet. In diesem Fall verliert Pac-Man eines seiner zu Beginn vorhandenen drei Leben. Jeder der vier Geister, die in der amerikanischen Pac-Man-Version Blinky, Pinky, Inky und Clyde heißen, hat eine andere Strategie, um den vom Benutzer gesteuerten Pac-Man zu fangen. Neben den gewöhnlichen Pillen, durch die der Spieler Punkte erwerben kann, gibt es auch noch, meist in der Nähe der Ecken des Spielfeldes, vier *Kraftpillen*, die Pac-Man mehr Punkte und außerdem die Fähigkeit seinerseits die Geister zu

[2] Vgl. [Luc05] und [GR03].

[3] Arcade-Computerspiele werden seit den 1970er Jahren (meistens in den USA) in Spielhallen kostenpflichtig angeboten.

Abbildung 2.4.: Bildschirmfoto des originalen Pac-Man-Spieles.

jagen, einbringen. Isst Pac-Man eine Kraftpille, färben sich die Geister für eine gewisse Zeit dunkelblau, ändern ihre bisherige Richtung und werden deutlich langsamer. Hat Pac-Man einen Geist gegessen, bekommt er einen gesonderten Punktebonus und der Geist erwacht nach kurzer Zeit in der Spielfeldmitte zu neuem Leben. Zusätzlich zu den bisher gesehenen drei Möglichkeiten für Pac-Man Punkte zu sammeln, gibt es noch eine weitere: Manchmal erscheint an einer zufälligen Position ein Symbol einer Frucht (z.B. einer Kirsche), welches Pac-Man Extrapunkte einbringt.

Das Spiel ist beendet, wenn alle vorhandenen Pillen von Pac-Man gegessen wurden oder er seine drei Leben verloren hat. Für den Fall, dass alle Pillen gegessen wurden, wird ein neuer Level erreicht, welcher etwas schwieriger zu spielen ist als der vorangegangene. Dies wird unter anderem durch sich schneller bewegende Geister erreicht. Beim Pac-Man-Spiel von Namco war es möglich 255 Level zu spielen. Hat Pac-Man alle Leben verloren, verharrt der Spieler in dem entsprechenden Level und hat in einem neuen Spiel die Chance ein Level weiter zu kommen.

2.3. DEViL

DEViL ist ein Generatorsystem [SC09a], welches aus Spezifikationen von hohem Niveau grafische Struktureditoren generiert. Diese Editoren besitzen eine Multi-Fenster Umgebung und viele weitere, schon aus anderen Editoren bekannte, Funk-

tionalitäten wie Copy-And-Paste, Drucken, Speichern und Laden von Beispielen. Außerdem ist es durch beliebig viele Sichten möglich, das Modell der Sprache zu betrachten. Um einen Eindruck davon zu bekommen, ist in Abbildung 2.5 ein Bildschirmfoto eines solchen Editors (in diesem Fall einer zum Erstellen von generischen Zeichnungen[4]) zu sehen.

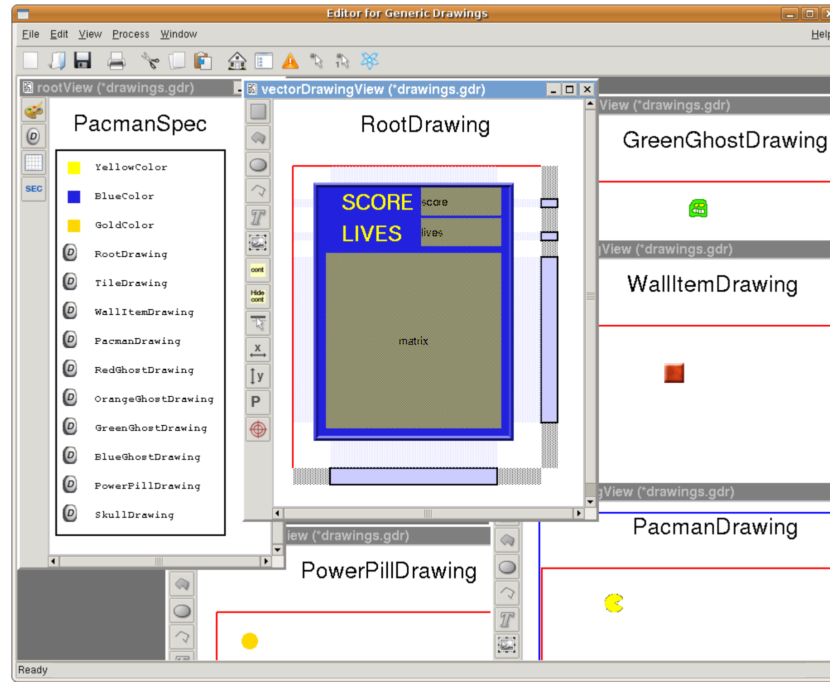

Abbildung 2.5.: Multi-Fenster Umgebung des Editors für generische Zeichnungen.

Grafische Struktureditoren lassen sich in DEViL durch die Spezifikation der *abstrakten Struktur* und der *visuellen Sichten* erzeugen. Durch die Angabe der abstrakten Struktur wird das Modell der visuellen Sprache in Form einer Klassenhierarchie spezifiziert. Die visuellen Sichten stellen einen, auf der abstrakten Struktur basierenden Baum oder Teilbaum, visuell dar. Jeder Knoten des Baumes repräsentiert dabei die Instanz einer Klasse der abstrakten Struktur. Die visuelle Repräsentation der Bäume wird durch sogenannte vordefinierte *visuelle Muster*, wie z.B. Listen, Mengen oder Matrizen, erreicht.

Von DEViL wird das Übersetzergenerator-Framework Eli [KPJ98] benutzt, wodurch es möglich ist, alle Funktionen des Eli Systems zu nutzen. So lässt sich eine semantische Analyse spezifizieren, die überprüft, ob eine im Editor erstellte Struktur semantisch korrekt ist. Außerdem lässt sich auf Basis einer semantisch

[4]Vgl. hierzu Abschnitt 2.3.2.

korrekten Struktur Code generieren, um so eine Source-to-Source Übersetzung von einer visuellen Sprache in eine beliebige textuelle Zielsprache zu realisieren.

Mit DEViL lassen sich nicht nur Editoren spezifizieren, die eine visuelle Sprache statisch darstellen. Es ist auch möglich, visuelle Sprachen zu simulieren und zu animieren. Dabei beschreibt eine diskrete Transformation der visuellen Sprache die Simulation und die Visualisierung, der bei der Simulation auftretenden diskreten Schritte, die Animation. Bei der Animation werden sogenannte *animierte visuelle Muster*, die mit den visuellen Mustern vergleichbar sind, verwendet.

In den folgenden Abschnitten werde ich genauer auf die oben kurz beschriebenen Konzepte eingehen. Zur besseren Verständlichkeit werde ich an mehreren Stellen Quelltext-Beispiele anbringen.

2.3.1. Abstrakte Struktur

Um mit DEViL eine visuelle Sprache zu spezifizieren, ist es zunächst notwendig, das semantische Modell der visuellen Sprache zu spezifizieren. Dieses Modell wird durch die Spezifikationssprache DSSL (DEViL Structure Specification Language) beschrieben. Diese erlaubt es, eine abstrakte Struktur zu spezifizieren, die sich am Modell von objektorientierten Programmiersprachen orientiert und Konzepte wie Klassen, Assoziation, Aggregation, Vererbung und Definition von Attributen verwendet.

In Quelltext 2.1 ist ein Beispiel für die Spezifikation einer abstrakten Struktur in DSSL zu sehen. In der abstrakten Struktur muss mindestens eine Klasse `Root` enthalten sein, die die Wurzel der abstrakten Struktur repräsentiert. Jede Klasse beschreibt durch Attribute die Struktur eines bestimmten Sprachelements. Attribute können drei verschiedenen Typen angehören:

VAL-Attribute speichern Werte, die entweder vordefinierten (z.B. String, Integer, Boolean oder Point) oder selbst definierten Datentypen angehören.

REF-Attribute speichern Referenzen zu anderen Sprachobjekten.

SUB-Attribute speichern Unterbäume, Listen von Unterbäumen oder können leer sein.

```
CLASS Root{
  score: SUB Score?;
  lives: SUB Lives?;
  matrix: SUB Matrix?;
  sound: VAL VLBoolean INIT "1" EDITWITH "Radio -values
    {off 0 on 1}";
}
```

```
CLASS Score{
  score: VAL VLInt INIT "0" EDITWITH "None";
}
CLASS Lives{
  lives: VAL VLInt INIT "3" EDITWITH "None";
}
CLASS Matrix{
  rows: SUB Row*;
  columns: SUB Column*;
}
CLASS Row{
  tiles: SUB Tile*;
}
CLASS Column{ }
CLASS Tile{
  columnRef: REF Column EDITWITH "None";
  item: SUB Item?;
}
ABSTRACT CLASS Item{
  name: VAL VLString;
}
CLASS WallItem INHERITS Item{ }
CLASS Pacman INHERITS Item{
  direction: VAL VLInt INIT "2" EDITWITH "None";
  angle: VAL VLInt INIT "0" EDITWITH "None";
  clockwise: VAL VLBoolean EDITWITH "None";
}
CLASS Ghost INHERITS Item{
  strategy: VAL VLInt EDITWITH "Radio -values
    {Random 1 {Incremental Approach} 2 {Hill Climbing} 3}";
  eatable: VAL VLBoolean INIT "0" EDITWITH "None";
}
CLASS PowerPill INHERITS Item{
  price: VAL VLInt INIT "100" EDITWITH
    "Scale -from 0 -to 500";
}
```

Quelltext 2.1: Spezifikation der abstrakten Struktur.

Um den VAL-Attributen Initialwerte zuzuweisen, gibt es die INIT-Klausel. Um sicherzustellen, dass die abstrakte Struktur bestimmte Bedingungen erfüllt, gibt es die CHECK-Klausel, die für VAL-, REF- und SUB-Attribute erlaubt ist. So kann z.B. überprüft werden, ob die Werte eines bestimmten Attributs in einem vorgegebenen Bereich liegen. Ist dies nicht der Fall, wird eine Fehlermeldung ausgegeben. Es gibt bereits eine Menge von vordefinierten Prüffunktionen. Falls diese nicht ausreichen, lassen sich mit der Programmiersprache Tcl eigene Funktionen schreiben. Für

komplexere Initialisierungen lassen sich ebenfalls eigene Funktionen schreiben, die nach der INIT-Klausel aufgerufen werden können.

Um Eigenschaften, die verschiedene Klassen gemeinsam haben, zu kapseln, stellt DSSL ein objektorientiertes Vererbungskonzept zur Verfügung. Zur Definition gemeinsamer Eigenschaften gibt es abstrakte Klassen, von denen geerbt werden darf. DSSL erlaubt Mehrfachvererbung, allerdings dürfen Unterklassen nur von abstrakten Oberklassen erben. Instanzen abstrakter Klassen sind nicht erlaubt. Im Quelltext 2.1 wird die Eigenschaft des Namens, den die Klassen `WallItem`, `Pacman` und `Ghost` gemeinsam besitzen, in der abstrakten Klasse `Item` gekapselt.

Das Modell der abstrakten Struktur ist der einzige Teil der Spezifikation, der zwingend erforderlich ist. Aus diesem generiert DEViL, für die interne Weiterverarbeitung mit Werkzeugen aus dem Eli System, eine *kontextfreie Grammatik*. Nur durch Angabe der abstrakten Struktur kann bereits ein einfacher Editor generiert werden, der eine Baumansicht implementiert und es über diese erlaubt, die abstrakte Struktur zu manipulieren.

2.3.2. Visuelle Darstellung

Um die abstrakte Struktur oder Teile daraus grafisch darzustellen, gibt es sogenannte Sichten, die es erlauben, Änderungen an der abstrakten Struktur vorzunehmen. Es kann beliebig viele Sichten geben, die die abstrakte Struktur jeweils unterschiedlich visualisieren. Jedes Objekt der abstrakten Struktur kann als grafisches Element der Sicht nur als Ganzes selektiert, verschoben oder gelöscht werden.

Bei der Deklaration der Sichten wird auch festgelegt, welche Toolbar-Knöpfe die Sicht enthält. Diese werden am linken Rand des Sichtfensters angezeigt. Über die Toolbar-Knöpfe, die entweder durch Text oder eine Grafik dargestellt werden, können neue Sprachkonstrukte per *Drag-and-Drop* eingefügt werden.

```
VIEW pacmView ROOT Root{
  Matrix(body(rows, columns));
}
```

Quelltext 2.2: Definition der Sicht.

Aus jeder Definition einer Sicht werden Grammatikproduktionen generiert, die die Grundlage für die grafische Darstellung bilden. Quelltext 2.2 zeigt die Definition einer Sicht. Diese lässt sich als Baum interpretieren: `Matrix` hat den Unterbaum `body` und `body` hat die Unterbäume `rows` und `columns`. Der Wurzelknoten muss einer Klasse aus der abstrakten Struktur und die Blätter müssen Attributen dieser Klasse entsprechen. Die inneren Knoten dürfen beliebig benannt werden. Über-

prüft man diese Anforderungen mit der abstrakten Struktur aus Quelltext 2.1, sieht man, dass sie erfüllt sind.

Um eine erweiterte grafische Repräsentation zu erreichen, können sogenannte visuelle Muster auf das semantische Modell der Sprache angewandt werden. Visuelle Muster charakterisieren spezifische visuelle Darstellungskonzepte und beschreiben, wie Strukturbäume visuell repräsentiert werden sollen. So kann z.B. spezifiziert werden, dass ein Teilbaum durch das abstrakte Konzept „Liste" und eine Untermenge als Listenelemente repräsentiert wird. DEViL nutzt eine Reihe von häufig vorkommenden visuellen Mustern, wie z.B. Listen, Mengen, Formularen, Tabellen oder Matrizen. Genauere Informationen sind in der entsprechenden Bibliothek [SC09b] nachzulesen.

```
SYMBOL pacmView_Matrix_body INHERITS VPFormElement, VPMatrix
COMPUTE
  SYNT.formElementName = "tiles";
  SYNT.minCellSize = VLPoint(24,24);
  SYNT.bgFillColor = "black";
END;
```

Quelltext 2.3: Anwendung visueller Muster durch eine attributierte Grammatik.

Implementierungen visueller Muster werden angewendet, indem Grammatiksymbole von bestimmten Berechnungsrollen erben, wobei Mehrfachvererbung erlaubt ist. Um dies zu realisieren, werden *attributierte Grammatiken* angewendet, die Berechnungen im abstrakten Strukturbaum durchführen. Anschließend wird durch den Attributauswerter *LIGA* die abschließende grafische Repräsentation berechnet. Die attributierten Grammatiken werden in der Sprache *LIDO* spezifiziert. Sowohl LIDO als auch LIGA sind beide Bestandteil des Eli Systems. Die Berechnungsrollen jedes Musters implementieren bestimmte Schnittstellen und setzen andere Schnittstellen voraus. Durch diese Tatsache können die visuellen Muster fast beliebig miteinander kombiniert werden. In Quelltext 2.3 ist zu sehen, wie das Grammatiksymbol `pacmView_Matrix_body` die Berechnungsrollen der visuellen Muster `VPFormElement` und `VPMatrix` erbt und wie die Attribute `formElementName`, `minCellSize` und `bgFillColor` bestimmt werden.

Neben dem Konzept der visuellen Muster bietet DEViL noch den Editor *gded*[5] an, mit dem sich sogenannte *generische Zeichnungen* erstellen lassen. Generische Zeichnungen definieren die grafische Repräsentation von Formulardarstellungen. Bei ihnen handelt es sich um grafische Spezifikationen, mit denen sich bestimmte Ausprägungen von Formulardarstellungen einfach und komfortabel beschreiben lassen. Um die generischen Zeichnungen zu referenzieren, können die visuellen

[5] Vgl. Abbildung 2.5 auf Seite 10.

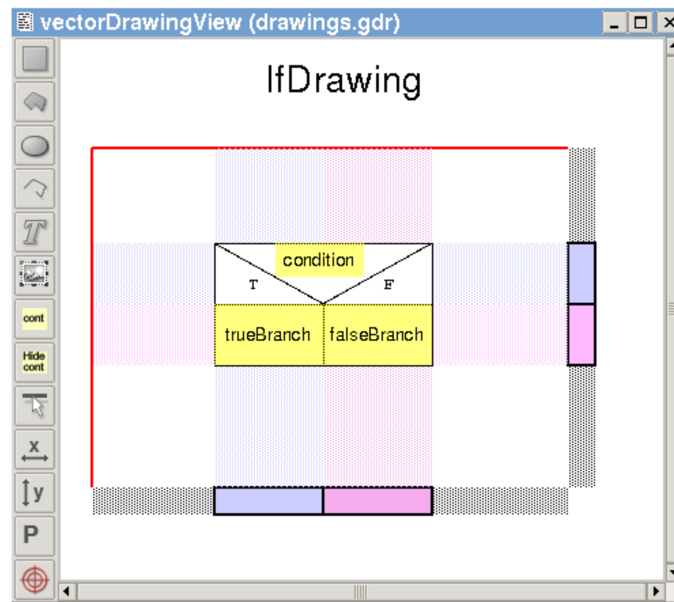

Abbildung 2.6.: Generische Zeichnung eines If-Konstrukts in Nassi-Shneiderman-Diagrammen.

Muster `VPForm` oder `VPFormList` angewendet werden. In Abbildung 2.6 ist die generische Zeichnung eines If-Konstrukts in *Nassi-Shneiderman-Diagrammen* zu sehen. Diese Zeichnung enthält neben grafischen Primitiven wie Linien und Rechtecken noch die Textprimitive `T` und `F`. In gelb sind die sogenannten Container dargestellt, die bestimmen, wie die Unterelemente des Formulars positioniert werden. Die seitlich dargestellten Dehnungsintervalle bestimmen, wie die Zeichnung transformiert werden soll, um Platz für den Containerinhalt zu schaffen.

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

Quelltext 2.4: Zuordnung einer generischen Zeichnung.

Generische Zeichnungen werden in einer Datei mit der Endung „.gdr“ gespeichert und, wenn eine solche Datei im Spezifikationsverzeichnis vorhanden ist, automatisch bei der Generierung des Editors berücksichtigt. Quelltext 2.4 zeigt, wie die Zeichnung `RootDrawing` einem Grammatiksymbol zugeordnet wird.

2.3.3. Semantische Analyse und Code-Generierung

Um die im spezifizierten Editor erstellten visuellen Ausdrücke zu analysieren oder in eine textuelle Zielrepräsentation zu übersetzen, lassen sich sogenannte *Prozessoren* definieren. Zur Definition dieser Prozessoren werden wieder attributierte Grammatiken, die in der Sprache LIDO beschrieben werden, verwendet, wodurch alle Werkzeuge zur Verfügung stehen, die auch vom Eli System genutzt werden.

```
Relation(from:string, to:string):
  \draw[O]->[C] ([from]) -- ([to]);
```

Quelltext 2.5: IPTG-Textmuster zur Code-Generierung.

Sowohl für die semantische Analyse als auch für die Code-Generierung wird ein Attributauswerter benutzt, der aus der Spezifikation der attributierten Grammatik konstruiert wurde. Die semantische Analyse wird zeitlich vor der Code-Generierung durchgeführt. Eine mögliche Anwendung der semantischen Analyse wäre z.B. bei einem Editor für Zustandsdiagramme die Überprüfung, ob der Startzustand genau eine ausgehende und keine eingehende Transition besitzt. Wird dabei eine Verletzung der Vorgaben festgestellt, wird eine Fehlermeldung ausgegeben, andernfalls wird die Code-Generierung gestartet.

```
SYMBOL codegen_Relation
COMPUTE
  SYNT.code = PTGRelation(
    VLString(vlGet(THIS.objId,"THIS.from.VALUE.name.VALUE")),
    VLString(vlGet(THIS.objId,"THIS.to.VALUE.name.VALUE"))
  );
END;
```

Quelltext 2.6: Spezifikation der Code-Generierung.

Das wichtigste Werkzeug bei der Generierung von Quelltext ist *IPTG* [Kas], welches auf dem Werkzeug PTG aus dem Eli System basiert. IPTG stellt eine Funktionalität bereit, durch die sich jegliche textuelle Sprache als Zielsprache unterstützen lässt. Die Struktur der Zielsprache wird in IPTG durch Textmuster beschrieben. Aus den Textmustern, die Parameter besitzen können, generiert IPTG eine Menge von Funktionen, durch deren Aufruf die Zielstruktur Schritt für Schritt zusammengesetzt werden kann. In Quelltext 2.5 ist ein IPTG-Textmuster dargestellt. Das Muster `Relation` hat zwei formale Parameter, die in ihrem Rumpf eingesetzt werden. Bei den Zeichensequenzen `[O]` und `[C]` handelt es sich um Notationen, die die eckigen öffnenden und schließenden Klammern beschreiben. Dies ist notwendig, da die eigentlichen Symbole in IPTG schon durch die Kennzeichnung der Einfügestellen besetzt sind. In Quelltext 2.6 wird die aus dem Textmuster `Relation`

generierte Funktion `PTGRelation` mit den zwei Parametern, die durch *Pfadausdrücke* bestimmt werden, aufgerufen.

2.3.4. Simulation und Animation

Seit kurzem ist es mit DEViL möglich visuelle Sprachen zu simulieren und zu animieren[6]. Die Simulation einer visuellen Sprache versteht man dabei als eine Menge von Transformationen des semantischen Modells der visuellen Sprache. Die Simulation ist eine reine Ausführungssemantik der visuellen Sprache und von der grafischen Repräsentation wird an dieser Stelle vollkommen abstrahiert. DEViL stellt zur Beschreibung der Simulation die Simulationsspezifikationssprache DSIM zur Verfügung. Erst die Animation kann die diskreten Transformationen durch eine kontinuierliche Animation geeignet visualisieren. Hierfür werden zur Steuerung Simulationsänderungsoperationen genutzt. Besonders komfortabel ist die Tatsache, dass aus einer Simulationsspezifikation automatisch eine Animation generiert werden kann.

Um den Unterschied zwischen Simulation und Animation zu verdeutlichen, stelle man sich einen grafischen Editor für Petri-Netze vor. In einem Petri-Netz mit Kantengewicht eins und unbegrenzter Kapazität der Stellen darf eine Transition schalten, wenn sich in allen Eingangsstellen mindestens ein Token befindet. Dann muss die Tokenanzahl in den Vorbedingungen der Transition dekrementiert und in allen Nachbedingungen inkrementiert werden. Dies beschreibt die Simulation. Erst die Animation sorgt für den Flug der Token von der Eingangsstelle über die Transition hin zur Ausgangsstelle.

Der Simulator

Zur Simulation bedarf es eines Simulators, der auf einem maßgeschneiderten *Simulationsmodell* arbeitet, welches die Grundlage der Simulation darstellt. In Abbildung 2.7 ist das grundlegende Konzept des Simulators abgebildet. Der Simulator simuliert das Simulationsmodell, welches auf dem semantischen Modell der visuellen Sprache beruht und ggf. zur Erfüllung bestimmter Anforderungen durch Zufallszahlen oder Warteschlangen erweitert wird.

Der Simulator ist vom semantischen Modell der visuellen Sprache soweit wie möglich entkoppelt, damit an vorhandenen Spezifikationen keine Fehler durch Seiteneffekte entstehen und damit bei Editoren, deren Spezifikation keine Simulationsdefinition enthält, der Simulator quasi nicht vorhanden ist. Bei der Entwicklung einer visuellen Sprache in einem Team kann es vorkommen, dass Teile, die für die Simulation wichtig sind, noch nicht in der visuellen Sprache vorhanden

[6] Vgl. [Cra08] und [CK09].

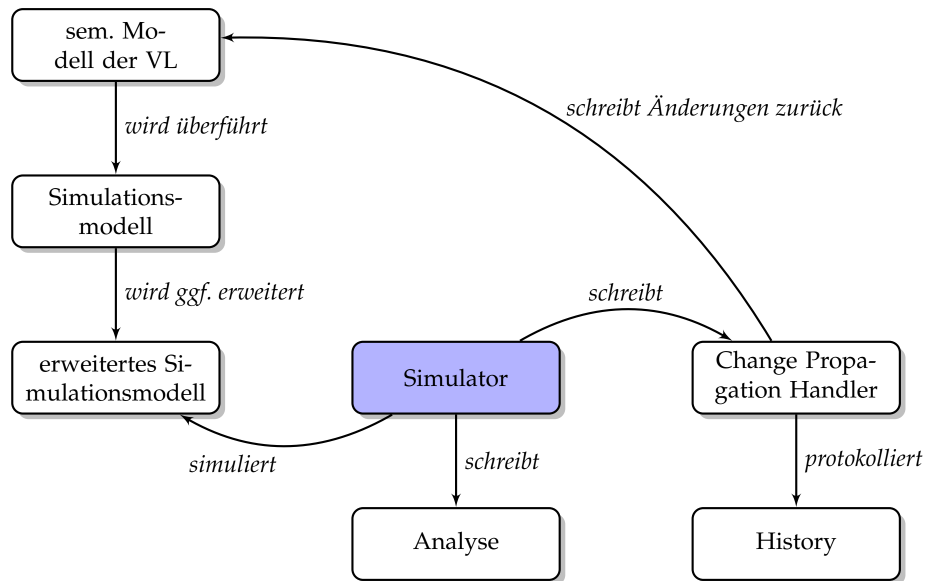

Abbildung 2.7.: Das Konzept des Simulators [Cra08].

sind. Da das Simulationsmodell vollständig von der visuellen Sprache entkoppelt ist, können diese Teile zunächst ins Simulationsmodell eingefügt werden und nachträglich in die visuelle Sprache übernommen werden.

Änderungen werden vom Simulator zunächst in sein eigenes Simulationsmodell geschrieben und dem *Change Propagation Handler* (CPH) werden die Änderungen gegenüber dem Modell der visuellen Sprache mitgeteilt. Der CPH schreibt diese Änderungen bei Bedarf in das semantische Modell der Sprache zurück. Das Intervall, mit dem diese Rückschreibeoperationen geschehen, kann flexibel gewählt werden. Standardmäßig beträgt das Intervall 1, sodass jeder Simulationschritt direkt in die visuelle Sprache transformiert wird. Weiterhin protokolliert der CPH alle Modifikationen in einer History, sodass die Simulation schrittweise rückgängig gemacht werden kann. Der Simulator schreibt außerdem Simulationsergebnisse, in ein Analysemodul in dem Simulationsschritte protokolliert werden. So ist es möglich, Auslastungsgraphen zu Zufallszahlen oder Warteschlangen zu generieren.

Die Sprache DSIM

Die eingangs erwähnte Simulationsspezifikationssprache DSIM wird benutzt, um die Simulation der visuellen Sprache zu spezifizieren. Die Sprache DSIM besteht aus vier Teilen: der Definition des *Simulationsmodells*, der *Simulationsschleife*, dem *Ereignisblock* und dem optionalen *Konfigurationsblock*.

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

Quelltext 2.7: Simulationsmodell für Petri-Netze.

In dem Simulationsmodell kann das semantische visuelle Modell der Sprache modifiziert und erweitert werden. Alle Klassen des semantischen Modells können, müssen aber nicht, im Simulationsmodell vorkommen. Wird eine Klasse aus dem semantischen Modell der visuellen Sprache in das Simulationsmodell übernommen, so ist sie ein komplettes Abbild ihres Pendants im Modell der visuellen Sprache. Die Erweiterung des Simulationsmodells wird durch die Definition von zusätzlichen Simulationsklassen und simulationsspezifischen Attributen wie Warteschlangen, Mengen, Zufallsvariablen oder Werten realisiert. Am häufigsten werden hiervon die Mengen benutzt, die eine Sammlung von Strukturobjekten, die eine bestimmte Bedingung erfüllen, definieren. Die Bedingung selber wird durch die Angabe eines Pfadausdrucks spezifiziert. In Quelltext 2.7 ist das Simulationsmodell für Petri-Netze zu sehen.

```
LOOP{
  FOREACH t IN Transition RANDOM{
    IFEVERY SIZE(t.incomingPlaces->tokens) > 0 {
      FIRE preTransitionFire(t) @ TIME_DIRECT;
      FIRE postTransitionFire(t);
   }
  }
}
```

Quelltext 2.8: Simulationsschleife.

In der Simulationsschleife, die vom Simulator in einer Endlosschleife durchlaufen wird, wird die eigentliche Simulation spezifiziert. Von dort aus kann auf das Simulationsmodell zugegriffen werden und Ereignisse, die im Ereignisblock deklariert wurden, können in eine Ereigniswarteschlange eingefügt werden. Um auf einzelne oder auf Mengen von Sprachkonstrukten zuzugreifen, bietet DSIM vier verschiedene Quantoren (`IFSOME`, `IFEVERY`, `FOREACH`, `FORONE`) an. Eine Anwendung dafür ist in Quelltext 2.8 zu sehen. Befindet sich in allen Eingangsstellen mindestens ein Token, wird das Ereignis `preTransitionFire`, gekennzeich-

net durch das Schlüsselwort `@ TIME_DIRECT` sofort ausgeführt und das Ereignis `postTransitionFire` wird in die Warteschlange des Simulators eingefügt.

Im Ereignisblock werden Ereignisse spezifiziert, die in der Simulationsschleife aufgerufen werden können. Ereignisse modifizieren die Instanz des Simulationsmodells. So werden in Quelltext 2.9 die beiden Ereignisse `preTransitionFire` und `postTransitionFire` definiert, die schon in der Simulationsschleife aufgerufen wurden.

Der Konfigurationsblock wird verwendet, um simulationsspezifische Merkmale zu kapseln. So können dort C-Header Dateien eingebunden werden, um in der DSIM-Spezifikation auf eigene C-Funktionen zugreifen zu können. Außerdem ist es dort möglich, die Simulationswurzel festzulegen. Möchte man, abweichend vom Standard, nicht die gesamte semantische Struktur ausgehend von `Root` simulieren, kann man eine andere Simulationswurzel angeben, die nur einen bestimmten Teilbaum in das Simulationsmodell einliest. Darüber hinaus können an dieser Stelle Tastaturereignisse definiert werden, durch die es in der Simulationsschleife möglich ist bestimmte Abläufe zu starten, je nach dem welche Eingabe der Benutzer über die Tastatur tätigt.

```
EVENTS{
  preTransitionFire(Transition transition) {
    FOREACH place IN transition.incomingPlaces {
      Token token = REMOVE(place->tokens);
    }
  }
  postTransitionFire(Transition transition) {
    FOREACH place IN transition.outgoingPlaces {
      INSERT(place->tokens, "Token");
    }
  }
}
```

Quelltext 2.9: Ereignisblock.

Das Animationsframework

Für die Animation wird auf das Konzept der linearen grafischen Interpolation zurückgegriffen. Da der grafische Zustand von zwei aufeinanderfolgenden Simulationsschritten, der durch zwei Attributauswerterläufe berechnet werden kann, wohl bekannt ist, kann die Größe und Position aller Strukturobjekte gespeichert werden. Der nächste Schritt ist die grafische Interpolation (also Umgestaltung oder Überführung) zwischen diesen beiden Schnappschüssen. Die Sprache DSIM beinhaltet vier Simulationsänderungsoperationen (`CREATE`, `REMOVE`, `MOVE` und `COPY`), die

das Animationsframework steuern und die Ausführung von Standardanimationsaktionen bewirken. Außerdem gibt es noch die Simulationsänderungsoperation `INSERT`, die allerdings als Abkürzung für die `MOVE`-Operation verstanden werden kann, wenn ein bereits existierender Teilbaum eingefügt werden soll. Wird ein komplett neues Objekt eingefügt, ist die Operation `INSERT` mit `CREATE` äquivalent. Es gibt insgesamt vier Standardanimationsaktionen, die zu den Simulationsänderungsoperationen getriggert werden. Dies sind im Einzelnen:

`CREATE` Das erstellte Objekt wird langsam bis zu seiner endgültigen Größe vergrößert.

`REMOVE` Das gelöschte Objekt wird langsam verkleinert, bis es nicht mehr zu sehen ist.

`MOVE` Das erstellte Objekt wird von seiner ursprünglichen Position bis zur endgültigen Position bewegt.

`MORPH` Eine Änderungsoperation an einem anderen Knoten hat dazu geführt, dass sich der Platzbedarf des aktuellen Objektes geändert hat. Es wird gemorpht.

Formulardarstellungen, die auf generischen Zeichnungen beruhen, vergrößern sich beim Einfügen von neuen Elementen nicht linear. Um diesen nichtlinearen Platzbedarf zu realisieren, kann DEViL automatisch generische Zeichnungen transformieren. Dies führt zu einer *weichen* Animation.

Die Standardanimationsaktionen sind in den meisten Fällen verwendbar, allerdings gibt es Ausnahmen, in denen die Standardanimation die Wünsche des Benutzers nicht erfüllt. Denken wir z.B. an das eingangs beschriebene Beispiel mit dem Schalten einer Transition in Petri-Netzen. Die Standardanimation würde so aussehen, dass in den Vorbedingungen die Token bis zur Unsichtbarkeit einschrumpfen und in den Nachbedingungen neue Token bis zur vollständigen Größe wachsen würden. Um eine angepasste Animation, die den Flug der Token beschreibt, zu erreichen, muss man die Standardanimationsabbildung überschreiben. Dies wird mit den sogenannten *animierten visuellen Mustern* (AVP[7]) erreicht. Die AVPs sind mit den eingangs beschriebenen visuellen Mustern vergleichbar. Wenn eine Änderungsoperation an einem Sprachkonstruktknoten ausgelöst wird, überprüft das Animationsframework, ob an diesem Knoten ein AVP vom selben Typ vorhanden ist. Ist dies der Fall, wird das AVP, ansonsten werden die Standardanimationsaktionen benutzt. Es existiert eine Bibliothek [SC09b] für animierte visuelle Muster, die es u.a. ermöglicht, Objekte blinken, rotieren oder bewegen zu lassen.

[7] animated visual pattern.

3. Sprachentwurf

In diesem Kapitel soll der systematische Entwurf eines Editors für regelbasierte Spiele am Beispiel Pac-Man beschrieben werden. Dazu werde ich zunächst den implementierten Editor vorstellen. Danach folgt die Darstellung des Modells der abstrakten Struktur und der visuellen Darstellung. Im Anschluss gehe ich auf die für Editoren von regelbasierten Spielen besonders wichtige Simulation und Animation ein. Dies teilt sich wieder in drei Teile: Zunächst gebe ich einen Überblick und danach gehe ich genauer zuerst auf die Simulation und dann auf die Animation ein.

3.1. Der Editor

Zunächst folgt ein recht anschaulicher Überblick über die einzelnen Elemente des Editors, durch den verdeutlicht werden soll, wie eine Spielwelt sukzessive aufgebaut wird. Danach wird das dem Editor zugrunde liegende Modell der abstrakten Struktur und die Realisierung der visuellen Darstellung beschrieben. Diese beiden Aspekte sorgen dafür, dass sich die Spielwelten, so wie es im Überblicksteil beschrieben wird, erstellen lassen.

3.1.1. Überblick

Der mit DEViL erstellte Editor bietet Standardfunktionen wie das Erstellen, Öffnen, Speichern oder Schließen von Dateien, die Pac-Man-Welten beschreiben, an. Diese Dateien werden in eine Datei mit der Endung „.pacman" geschrieben. Der Editor ist als *Multiple Document Interface* (MDI) Anwendung realisiert, die es erlaubt, mehrere Dateien, die in separaten Unterfenstern angezeigt werden, gleichzeitig geöffnet zu haben. So ist ein schnelleres Editieren mehrerer Dateien zur gleichen Zeit möglich. In Abbildung 3.1 ist ein Bildschirmfoto des Editors zu sehen, in dem zwei Dateien geöffnet sind, die Pac-Man-Spielwelten darstellen.

Wird eine neue Datei erstellt, erscheint in einem neuen Fenster ein, in den blauen Hintergrund eingebettetes, Spielfeld von 10×10 Kacheln Größe. Oberhalb des Spielfeldes wird der Score und die noch verbliebenen Leben des Pac-Man angezeigt. Zu Beginn ist der Score null und Pac-Man hat drei Leben. An der linken Seite des Fensters sind sechs verschiedene Toolbar-Knöpfe zu sehen, mit denen sich

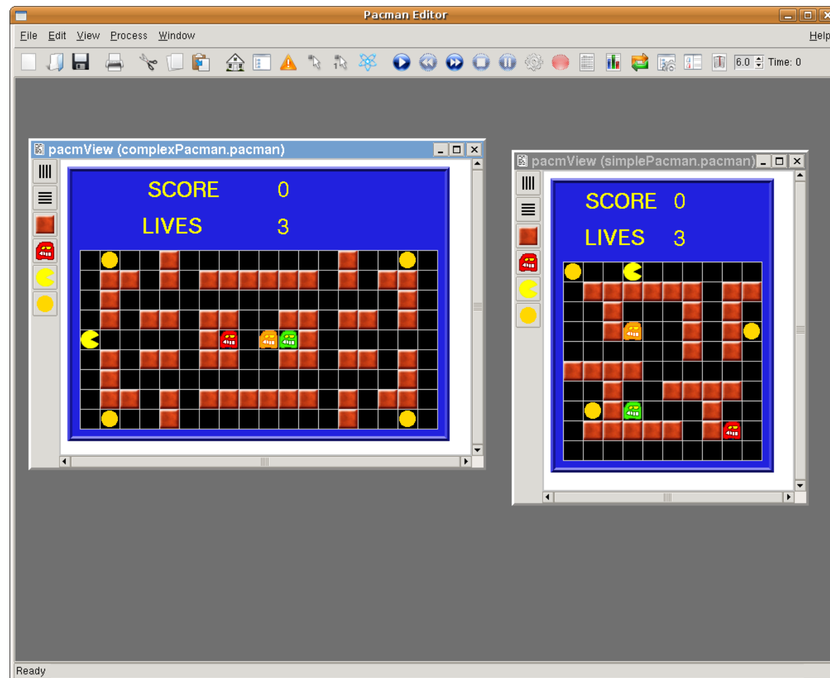

Abbildung 3.1.: Editor für Pac-Man-Spielwelten.

neue Spalten, Zeilen, Wandelemente, Geister, Pac-Man und Kraftpillen einfügen lassen. Wird einer dieser Knöpfe vom Benutzer ausgewählt und bewegt er danach die Maus auf das Spielfeld, werden dort die möglichen Einfügestellen für das ausgewählte Element angezeigt. Wird eine neue Zeile bzw. Spalte ausgewählt, befinden sich die möglichen Einfügestellen zwischen den bisher vorhandenen Zeilen bzw. Spalten. Wird eines der anderen Elemente ausgewählt, so befinden sich die möglichen Einfügestellen in jeder Kachel, die noch kein Element enthält. Ein schon eingefügtes Element kann durch Drag-and-Drop in eine andere freie Einfügestelle verschoben werden. Um ein gültiges Spielfeld zu erzeugen, muss sich auf diesem mindestens eine Kraftpille und ein Geist befinden. Außerdem wird gefordert, dass sich genau ein Pac-Man auf dem Spielfeld befindet. Eine genauere Beschreibung solcher Konsistenzprüfungen erfolgt in Abschnitt 4.2.

Das Editieren von Attributen eines Sprachkonstrukts ist entweder über einen Doppelklick auf die grafische Repräsentation des Konstrukts oder über das Kontextmenü möglich. In den Abbildungen 3.2 und 3.3 ist jeweils der Dialog zum Editieren der Attribute der Geister und der Kraftpillen zu sehen. Bei beiden kann der Name individuell festgelegt werden, wobei jedem Geist eine eigene Strategie zum Einfangen des Pac-Man zugewiesen werden kann. Jeder Kraftpille kann ein gewisser Wert zugewiesen werden, um den sich während des Spieles der Score er-

höht, wenn diese Kraftpille von Pac-Man gegessen wird. Durch einen Doppelklick auf den blauen Hintergrund des Spielfeldes ist es möglich, das Attribut `sound` der Klasse `Root` auf `on` oder `off` zu setzen, um so festzulegen, ob während der Simulation bei bestimmten Aktionen Geräusche abgespielt werden sollen.

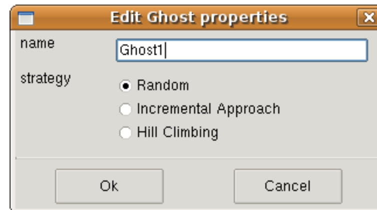

Abbildung 3.2.: Editieren der Geister-Attribute.

Jedem neu in das Spielfeld eingefügten Geist wird zufällig eine von drei möglichen Strategien zugewiesen. Je nach Strategie nimmt der Geist eine andere Farbe an: Der Geist mit der zufälligen Strategie ist grün, der mit der Strategie der schrittweisen Annäherung orange und der mit der Strategie des Bergsteigens rot. Diese initiale zufällige Festlegung kann, wie oben beschrieben, über das Editieren der Attribute verändert werden. Die Realisierung der zufälligen Zuweisung wird in Kapitel 4 näher erläutert. Dort werden außerdem die drei Algorithmen, die die Strategien der Geister beschreiben, sowie deren Implementierung vorgestellt.

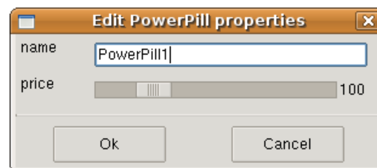

Abbildung 3.3.: Editieren der Kraftpillen-Attribute.

3.1.2. Das Modell der abstrakten Struktur

Die grundlegenden Konzepte zum Verständnis der abstrakten Struktur wurden bereits in Abschnitt 2.3.1 vorgestellt. Die in DSSL implementierte abstrakte Struktur des in dieser Arbeit entwickelten Editors diente bereits in Quelltext 2.1 auf Seite 11 als Beispiel für das allgemeine Verständnis der abstrakten Struktur. In Abbildung 3.4 ist das daraus abgeleitete Klassendiagramm dargestellt.

Die Wurzel der abstrakten Struktur wird durch die Klasse `Root` repräsentiert. Diese besitzt drei SUB-Attribute, die jeweils eine oder keine Instanz der Klassen `Score`, `Lives` bzw. `Matrix` enthält. `Score` wird verwendet, um die während des Spieles erzielten Punkte zu zählen, `Lives` verwaltet die Anzahl der noch vorhan-

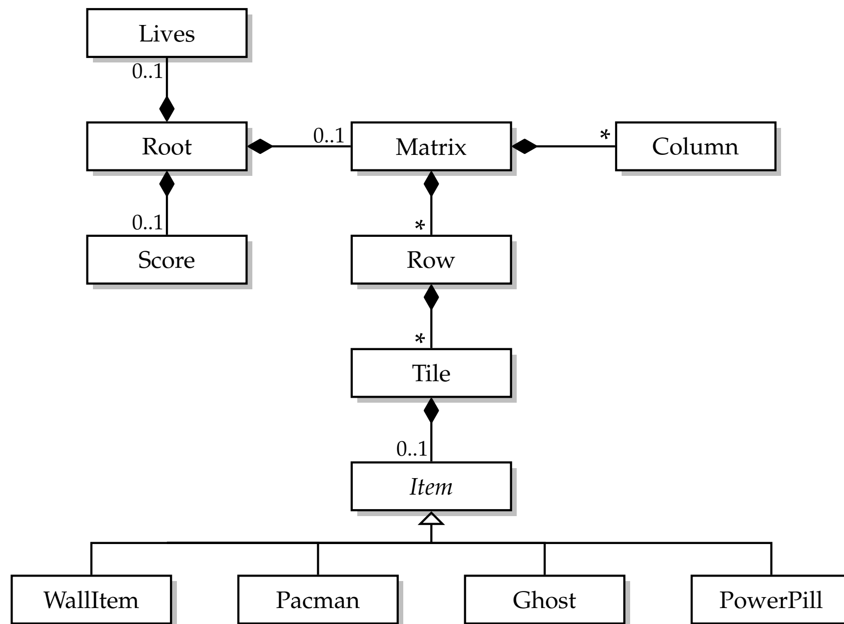

Abbildung 3.4.: Klassendiagramm des Modells der abstrakten Struktur.

denen Leben des Pac-Man und `Matrix` repräsentiert das kachelartige Spielfeld. Die Klasse `Matrix` hat wiederum zwei Attribute vom Typ SUB, die es aufgrund ihrer Multiplizität erlauben, beliebig viele Instanzen der Klassen `Row` und `Column` aufzunehmen. Um in der Struktur die Kacheln zu modellieren, besitzt die Klasse `Row` ein SUB-Attribut, welches beliebig viele Objekte der Klasse `Tile` besitzen kann. Jede Kachel, also eine Instanz der Klasse `Tile`, besitzt neben einem Referenz-Attribut auf die mit ihr assoziierte Instanz der Klasse `Column` ein SUB-Attribut, welches ein oder kein Objekt enthalten kann, das die Rolle von `Item` spielt.

Bei `Item` handelt es sich um eine abstrakte Klasse, im Klassendiagramm durch die kursive Schrift gekennzeichnet, von der keine Objekte existieren können. `Item` wird als Oberklasse der Unterklassen `WallItem`, `Pacman`, `Ghost` und `PowerPill` eingesetzt, um die gemeinsame Eigenschaft des Namens zu kapseln. Bei dieser Vererbungsbeziehung kommt das Vererbungsparadigma *Abstraktion* [Kas09] zur Anwendung. Alle eben aufgezählten vier Unterklassen erfüllen die Beziehung „ist spezielle Art von“ `Item` und sind außerdem disjunkt. Dies macht auch nur Sinn, da jede Kachel maximal ein Element beinhalten darf, welches entweder ein Wandelement, einen Pac-Man, einen Geist oder eine Kraftpille symbolisiert.

Die Erstellung des Klassendiagramms erfolgt als Komposition, also als existenzabhängige Teil-Ganzes Beziehung, der einzelnen Klassen der abstrakten Struktur. Dies bedeutet, dass ein Teil-Objekt `Tile` vollständig im Besitz des Ganzen-Objekt

Row ist. Wird das Row-Objekt gelöscht, so impliziert dies automatisch die Löschung der zu ihm assoziierten Tile-Objekte.

Spezielle Initialisierungsaspekte, die bei der Instanziierung bestimmter Objekte durchgeführt werden, werden in Abschnitt 4.1 genauer behandelt.

3.1.3. Die visuelle Darstellung

Für die visuelle Darstellung der Pac-Man-Spielwelten wurde eine Sicht (*pacmView*) entworfen. Die Sicht wurde, wie immer bei Editoren die mit DEViL entwickelt werden, mit attributierten Grammatiken spezifiziert. Um die visuelle Darstellung, wie sie anhand der fertigen Anwendung schon in Abbildung 3.1 zu sehen ist, zu realisieren, wurde zunächst eine generische Zeichnung erstellt. Diese ist in Abbildung 3.5 dargestellt. Im oberen Bereich, jeweils rechts neben dem Schriftzug SCORE bzw. LIVES sind zwei Container (score und lives) eingefügt worden, in denen der aktuelle Score, also die von Pac-Man gesammelten Punkte, bzw. die noch vorhandenen Leben angezeigt werden. Im unteren, den meisten Platz einnehmenden Bereich, ist ebenfalls ein Container (matrix) eingefügt worden, in dem das Spielfeld dargestellt wird. Für alle Container wurden Dehnungsintervalle definiert, die beschreiben wie die Zeichnung transformiert werden soll, um Platz für den Containerinhalt zu schaffen.

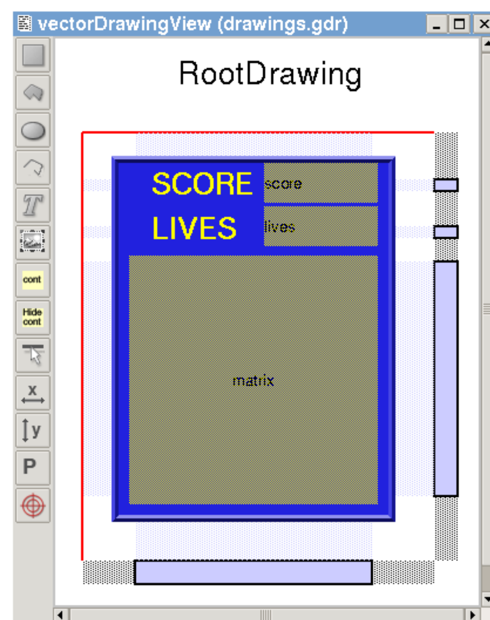

Abbildung 3.5.: Generische Zeichnung für die Sicht pacmView.

Da generische Zeichnungen die grafische Repräsentation von Formulardarstellungen definieren, muss zur Referenzierung der erstellten generischen Zeichnung das visuelle Muster `VPForm` angewendet werden. Weil die in Abbildung 3.5 dargestellte Zeichnung die Hauptsicht des Editors definiert, muss das, die Wurzel der abstrakten Struktur beschreibende, Grammatiksymbol die Berechnungsrollen des visuellen Musters `VPForm` erben. Genau dies wurde bereits im Grundlagenkapitel in Quelltext 2.4 auf Seite 15 als Beispiel dafür angeführt, wie eine generische Zeichnung einem Grammatiksymbol zugeordnet wird.

Um die drei in der generischen Zeichnung definierten Container zu referenzieren, müssen die entsprechenden Grammatiksymbole die Berechnungsrollen des visuellen Musters `VPContainer` erben. Innerhalb der Container `score` und `lives` kommt rekursiv das visuelle Muster `VPValTextPrimitive` zur Anwendung, um die Zeichenketten darzustellen, die sich aus den Werten des Attributs `score` der Klasse `Score` und des Attributs `lives` der Klasse `Lives` ergeben. Innerhalb des Containers `matrix` wird das visuelle Muster `VPMatrix` verwendet, um das kachelartige Spielfeld zu realisieren.

```
SYMBOL pacmView_Ghost INHERITS VPContainerElement, VPForm
COMPUTE
  SYNT.drawing = IF(EQ(THIS.pers_eatable,VLBoolean("1")),
    VisDrawingPtr(ADDROF(BlueGhostDrawing)),
    IF(EQ(THIS.pers_strategy,VLInt("1")),
      VisDrawingPtr(ADDROF(GreenGhostDrawing)),
      IF(EQ(THIS.pers_strategy,VLInt("2")),
        VisDrawingPtr(ADDROF(OrangeGhostDrawing)),
        VisDrawingPtr(ADDROF(RedGhostDrawing))))
  );
  SYNT.bgFillColor = "black";
  SYNT.pad = VLPoint(1,1);
  SYNT.balloon = IF(EQ(THIS.pers_strategy,VLInt("1")),
    VLString("Ghost, Random"),
    IF(EQ(THIS.pers_strategy,VLInt("2")),
      VLString("Ghost, Incremental Approach"),
      VLString("Ghost, Hill Climbing"))
  );
END;
```

Quelltext 3.1: Kontextabhängige Zuordnung generischer Zeichnungen.

In Quelltext 3.1 ist die kontextabhängige Zuordnung einer generischen Zeichnung an ein Grammatiksymbol dargestellt. Dort erfolgt die Zuordnung einer generischen Zeichnung an das Grammatiksymbol `pacmView_Ghost` in Abhängigkeit der Attribute `eatable` und `strategy` der Klasse `Ghost`, um so einem Geist, je nachdem, ob er gerade von Pac-Man gegessen werden kann und je nach gewählter

Strategie, ein anderes Aussehen zuzuweisen. Ist der Wert des Attributs `eatable` *true*, so wird ein blauer Geist dargestellt, ist er hingegen *false*, erfolgt die Zuweisung der generischen Zeichnung in Abhängigkeit des Attributs `strategy`. Ist der Wert des Attributs `strategy` 1, wird der Geist grün dargestellt, ist er 2, wird der Geist orange dargestellt und ist er 3, so wird der Geist rot dargestellt. In dem Quelltext ist außerdem zu sehen, wie dem Attribut `bgFillColor` die Farbe schwarz zugewiesen wird, damit das Spielfeld in dieser Farbe angezeigt wird. Durch die Bestimmung des Attributs `pad` wird ein zusätzlicher Rand definiert, der dafür sorgt, dass die generische Zeichnung, die einen Geist visualisiert, in der Kachel zentraler dargestellt wird. Durch die Zuweisung einer Zeichenkette an das Attribut `balloon` wird dafür gesorgt, dass ein Tooltip, erzeugt wird, wenn der Benutzer mit der Maus auf einen Geist zeigt. Dieser Tooltip zeigt an, dass es sich um einen Geist handelt und zeigt die ihm zugeordnete Strategie an. Dabei handelt es sich wieder um eine kontextabhängige Zuordnung, da das Attribut `strategy` vom Typ `VLInt` ist und erst über die kontextabhängige Zuordnung eine textuelle Benennung der Strategien möglich ist.

3.2. Die Simulation und die Animation

In diesem Abschnitt soll die Simulation und die Animation der im vorherigen Abschnitt vorgestellten visuellen Sprache beschrieben werden. Dafür werde ich zunächst in einem Überblicksteil schildern, wie die Simulation und die Animation des erstellten Spieles abläuft. Danach werde ich genauer vorstellen, wie die Simulation mit der Simulationsdefinitionssprache DSIM spezifiziert wurde. Da die aus der Simulation automatisch generierte Animation nicht ganz dem Anforderungsprofil eines Spieles genügte, musste die Standardanimationsabbildung mittels animierter visueller Muster überschrieben werden. Die Beschreibung dieses Sachverhalts inklusive einiger Quellcodebeispiele erfolgt im letzten Teil dieses Abschnitts.

3.2.1. Überblick

Nachdem ein gültiges Spielfeld erstellt wurde, kann das Spiel durch Start der Simulation begonnen werden. Dies geschieht durch einen Klick auf den Simulations-Startbutton in der Werkzeugleiste des Editors. Wurde das Spiel gestartet, kann der Benutzer mit den vier Pfeiltasten der Tastatur den Pac-Man steuern. Unmittelbar mit dem Start des Spieles setzen sich auch die Geister nach unterschiedlichen Strategien in Bewegung und versuchen den Pac-Man zu fangen. Die dafür angewandten Strategien werden genauer in Abschnitt 4.4 beschrieben.

Der Benutzer sollte das Ziel haben, so schnell wie möglich alle Kraftpillen zu essen, die sich auf dem Spielfeld befinden. Um dies zu tun, muss er den Pac-Man auf die Kachel der Kraftpille bewegen. Bevor diese verschwindet und der Pac-Man die Kachel besetzt, blinkt sie noch kurz ein paar mal auf und der Score des Pac-Man erhöht sich um den Wert der Kraftpille. Wurde eine Kraftpille gegessen, färben sich alle Geister blau und bewegen sich für eine vom Wert der gegessenen Kraftpille abhängigen Zeitraum nur nach der für Pac-Man ungefährlichsten zufälligen Strategie fort. Sind die Geister blau gefärbt, kann Pac-Man auch die Geister jagen und bei Erfolg seinen Score um 1000 Punkte erhöhen. Nach Ablauf des Zeitraums nehmen alle noch vorhandenen Geister wieder ihre ursprüngliche Farbe an und es werden so viele neue Geister an zufälligen Positionen erzeugt, wie vorher von Pac-Man gegessen wurden.

Abbildung 3.6.: Pac-Man hat das Spiel verloren.

Wird Pac-Man hingegen von einem Geist gegessen, erscheint ein weißer Totenkopf, der von der Kachel aus aufsteigt, an der er von einem Geist gegessen wurde. Dieser Zustand ist in Abbildung 3.6 dargestellt. Danach erscheint ein Dialog, in dem auf das verlorene Spiel hingewiesen wird und eventuell, in Abhängigkeit der noch vorhandenen Leben, ein neues Spiel angeboten wird. Das Spiel ist gewonnen, wenn alle Kraftpillen von Pac-Man gegessen wurden. In diesem Fall wird auch ein entsprechender Dialog angezeigt. Die Implementierungsdetails bzgl. der Dialoge sind in Abschnitt 4.5.2 nachzulesen.

Da in DEViL die Musik-Bibliothek *FMOD* [Fir] integriert wurde, ist es möglich Geräusche und Musik abzuspielen. Wie bereits in Abschnitt 3.1.1 beschrieben, kann der Benutzer vor dem Start des Spieles festlegen, ob Geräusche abgespielt werden sollen oder nicht. Ist dies der Fall, werden bei besonderen Ereignissen, wie z.B. beim Essen einer Kraftpille durch Pac-Man oder beim Essen des Pac-Man durch einen Geist, spezielle Geräusche abgespielt.

3.2.2. Die Simulation

Um die Simulation zu realisieren, wurde ein Konfigurationsblock definiert, der simulationsspezifische Merkmale, wie z.B. Tastaturereignisse und den Zugriff auf eigene C-Funktionen, kapselt. Im Simulationsmodell wurde das semantische visuelle Modell der Sprache um Attribute erweitert, die nur im Kontext der Simulation genutzt werden. Innerhalb des Ereignisblocks wurden Ereignisse definiert, die innerhalb der Simulationsschleife aufgerufen werden können. Innerhalb dieser Simulationsschleife wird die eigentliche Simulation spezifiziert, die auf Informationen aus dem Konfigurationsblock, Ereignisse aus dem Ereignisblock und erweiterte Attribute aus dem Simulationsmodell zurückgreift.

```
CONFIGURATION {
  #include "tileFunctions.h"
  REGISTER "<Up>" AS inputUp CONVERT "1" VLInt;
  REGISTER "<Right>" AS inputRight CONVERT "2" VLInt;
  REGISTER "<Down>" AS inputDown CONVERT "3" VLInt;
  REGISTER "<Left>" AS inputLeft CONVERT "4" VLInt;
}
```

Quelltext 3.2: Konfigurationsblock.

Die Tastatureingabe ist für das implementierte Spiel besonders wichtig. Für diesen Zweck wurden im Definitionsblock vier Tastaturereignisse definiert, auf die in der Simulationsschleife durch das Schlüsselwort `ACTION_EVENT` zugegriffen werden kann. In Quellcode 3.2 ist die Spezifikation des Konfigurationsblocks zu sehen. Zunächst wird der C-Header `tileFunctions.h` eingebunden, um in der Simulationsschleife auf eigene C-Funktionen zugreifen zu können. Die implementierten C-Funktionen werde ich in Abschnitt 4.4 genauer vorstellen. Außerdem wurden die schon eben beschriebenen vier Tastaturereignisse definiert.

```
MODEL {
  CLASS Ghost{
    VALUE tile OF Tile: "THIS.PARENT.PARENT";
  }
  ...
}
```

Quelltext 3.3: Ausschnitt aus dem Simulationsmodell.

Während der Simulation werden bestimmte Attribute einer Klasse gebraucht, die nicht im semantischen Modell der Sprache enthalten sind. So möchte man z.B. zu jedem Geist seine zugehörige Kachel kennen. Um dies zu realisieren bietet es sich an, das Simulationsmodell zu erweitern. In Quellcode 3.3 ist ein Ausschnitt

aus dem Simulationsmodell dargestellt. Es zeigt die Erweiterung der Klasse `Ghost` um das Attribut `tile`, welches mittels eines Pfadausdrucks bestimmt wird und die zum Geist gehörige Kachel beinhaltet.

Neben der Klasse `Ghost` wurde die Klasse `Pacman` auf die selbe Weise erweitert, um auf seine Kachel in der Simulation zugreifen zu können. Dies ist z.B. der Fall, wenn sich der Pac-Man oder ein Geist von einer Kachel auf eine andere bewegt. In diesem Fall muss dem Ereignis, welches die Bewegung definiert, die Start- und die Zielkachel übergeben werden.

Außerdem wurde die Klasse `Root` um das Attribut `numberOfEatenGhosts` erweitert. `numberOfEatenGhosts` wird während der Simulation inkrementiert, wenn Pac-Man einen Geist gegessen hat. Ist der Zeitraum, in dem Pac-Man die Geister essen kann, abgelaufen, werden `numberOfEatenGhosts`-viele Geister neu instanziiert.

Die Klasse `Tile` wurde um ein Attribut `pill` erweitert, über welches sich auf die Kraftpille zugreifen lässt, die sich auf der Kachel befinden kann. Außerdem wurde ein Attribut `position` hinzugefügt, welches für jede Kachel ihre kartesische Koordinate speichert. Die Initialisierung dieses Attributs wird in 4.1 ausführlich beschrieben. Darüber hinaus wurde die Klasse `Tile` um die Attribute `diffVal` und `visited` erweitert, die beide für die in 4.4.3 beschriebene Strategie des Bergsteigens benötigt werden.

Im Ereignisblock werden alle Ereignisse spezifiziert, die während eines Durchlaufs der Simulationsschleife aufgerufen werden können und dabei die Instanz des Simulationsmodells modifizieren. Im Folgenden möchte ich ausgewählte Ereignisse kurz beschreiben:

`coordinatePacman` Koordiniert als globales Ereignis die Fortbewegung des Pac-Man und ruft hierfür auch andere Ereignisse auf. Dieses wichtige Ereignis wird im nachfolgenden Quelltext genauer vorgestellt.

`goPacman` Bewegt den Pac-Man von seiner Ausgangs- zu seiner Zielkachel. Außerdem wird durch Aufruf des `incrementScore`-Ereignisses der Score um eins erhöht.

`eatPacman` Sorgt dafür, dass Pac-Man gegessen wird, indem sich der Geist von seiner Ausgangs- zu seiner Zielkachel bewegt und vorher der Pac-Man von der Zielkachel entfernt wird. Außerdem wird an dieser Stelle ein Geräusch abgespielt, welches den Tod des Pac-Man akustisch untermalt.

`computeRotation` Da sich der Pac-Man bei einer Richtungsänderung in diese Richtung dreht, muss die dafür notwendige Rotation berechnet werden. Schaut Pac-Man erst nach rechts und ändert dann seine Richtung nach unten, muss

eine Rotation um 90° im Uhrzeigersinn erfolgen. Dieses Ereignis wird immer aus anderen Ereignissen heraus aufgerufen, bei denen es darum geht den Pac-Man zu bewegen.

`eatableGhosts` Dieses Ereignis wird aufgerufen, wenn Pac-Man eine Kraftpille gegessen hat, um das Attribut `eatable` aller Geister auf *true* zu setzen und ihnen die schwächste Strategie zuzuweisen. Außerdem wird das Ereignis `notEatableGhosts` aufgerufen, welches allerdings erst für eine gewisse Zeit in die Ereigniswarteschlange eingefügt wird.

`notEatableGhosts` Setzt das Attribut `eatable` wieder auf *false* und weist allen Geistern ihre ursprüngliche Strategie zu. Außerdem wird von hier aus das Ereignis `addNewGhost` so oft aufgerufen, wie Geister von Pac-Man gegessen wurden.

`addNewGhost` Fügt auf einer zufälligen leeren Kachel einen neuen Geist ein.

`gameWon` Wird aufgerufen, wenn Pac-Man eine Kraftpille gegessen hat. Das Verhalten dieses Ereignisses wird allerdings erst durch aspektorientierte Programmierung spezifiziert. Dies wird genauer in 4.5.2 beschrieben.

```
EVENTS {
  coordinatePacman(Pacman pacman, int direction){
    Tile go = neighbourHasPowerpill(pacman.tile, direction);
    IF(NOTNULL(go)){
      FIRE eatPowerpill(pacman.tile, go, pacman, direction) @
          TIME_DIRECT;
    }{
      Tile go = neighbourHasGhost(pacman.tile, direction);
      IF(NOTNULL(go)){
        FIRE eatGhost(pacman.tile, go, pacman, direction) @
            TIME_DIRECT;
      }{
        Tile go = getNeighbour(pacman.tile, direction);
        IF(NOTNULL(go)){
          FIRE goPacman(pacman.tile, go, pacman, direction) @
              TIME_DIRECT;
        }
      }
    }
  }
  ...
}
```

Quelltext 3.4: Ausschnitt aus dem Ereignisblock.

In Quelltext 3.4 ist ein Ausschnitt aus dem Ereignisblock mit der Spezifikation des `coordinatePacman`-Ereignisses, welches als Parameter den Pac-Man und eine Bewegungsrichtung übergeben bekommt, zu sehen. Zunächst wird der Kachel `to` entweder die Nachbarkachel, falls diese eine Kraftpille enthält, oder `NULL` zugewiesen. Dies geschieht mittels der C-Funktion `neighbourHasPowerpill`, der die Ausgangskachel und eine Richtungsangabe übergeben wird. Befindet sich tatsächlich eine Kraftpille auf der Kachel, wird das Ereignis `eatPowerpill` ausgeführt. Dies geschieht, gekennzeichnet durch das Schlüsselwort `@ TIME_DIRECT`, unmittelbar, da das Ereignis nicht erst in die Simulationswarteschlange eingefügt wird. Wurde hingegen `NULL` zurückgegeben, wird in den anderen Block gesprungen, indem mittels einer C-Funktion geprüft wird, ob sich auf der Nachbarkachel ein Geist befindet, dessen Attribut `eatable` zur Zeit *true* ist. Ist dies der Fall, wird das Ereignis `eatGhost` ausgeführt. Wird allerdings auch hier `NULL` zurückgegeben, wird mittels einer weiteren C-Funktion die Nachbarkachel zurückgeliefert, auf die der Benutzer den Pac-Man mittels Tastatureingabe navigieren will. Diese Funktion gibt nur `NULL` zurück, wenn sich auf der Nachbarkachel ein Wandelement befindet oder wenn es keinen solchen Nachbarn gibt, weil sonst der Rand des Spielfeldes überschritten würde.

```
LOOP{
  FORONE pacman IN Pacman{
    IF(ACTION_EVENT[inputUp] == VLInt(1)){
      FIRE coordinatePacman(pacman, 1.0) @ TIME_DIRECT;
    }
    IF(ACTION_EVENT[inputRight] == VLInt(2)){
      FIRE coordinatePacman(pacman, 2.0) @ TIME_DIRECT;
    }
    IF(ACTION_EVENT[inputDown] == VLInt(3)){
      FIRE coordinatePacman(pacman, 3.0) @ TIME_DIRECT;
    }
    IF(ACTION_EVENT[inputLeft] == VLInt(4)){
      FIRE coordinatePacman(pacman, 4.0) @ TIME_DIRECT;
    }
  }
  ...
}
```

Quelltext 3.5: Ausschnitt aus der Simulationsschleife.

Mit Hilfe der spezifizierten Ereignisse, Definitionen und Erweiterungen des Modells lässt sich innerhalb der Simulationsschleife, die vom Simulator in einer Endlosschleife durchlaufen wird, die eigentliche Simulation spezifizieren. Die Simulation teilt sich in zwei Bereiche: Zum einen in die Simulation für den Pac-Man und zum anderen in die Simulation der Geister. Da es im Spiel immer nur einen

Pac-Man gibt, kann auf diesen mit dem Quantor FORONE zugegriffen werden. Da es mehrere Geister geben kann, wird auf diese mittels des Quantors FOREACH zugegriffen.

In Quelltext 3.5 ist ein Ausschnitt aus der Simulationsschleife mit einem Teil des spezifizierten Codes zur Beschreibung des Verhaltens des Pac-Man dargestellt. Dort wird jeweils mittels des Schlüsselworts ACTION_EVENT auf die im Konfigurationsblock definierten Tastaturereignisse zugegriffen und mit einer Ganzzahl, die die Bewegungsrichtung angibt, verglichen. Dann wird jeweils das bereits in Quelltext 3.4 genau beschriebene coordinatePacman-Ereignis mit der entsprechenden Bewegungsrichtung aufgerufen.

3.2.3. Die Animation

Die aus der Simulationsspezifikation automatisch generierte Standardanimation für die Bewegung von Pac-Man von einer Ausgangs- in eine Zielkachel, sieht so aus, dass sich der Pac-Man sehr schnell von der Ausgangskachel auf die Zielkachel bewegt und erst auf der Zielkachel in die gewählte Bewegungsrichtung blickt. Viel schöner ist es, wenn der Pac-Man noch in der Ausgangskachel in die vom Benutzer gewählte Richtung blicken würde und sich dann auf die Zielkachel bewegt.

Um dies zu erreichen, muss die Standardanimationsabbildung überschrieben werden. Dies wird mit den animierten visuellen Mustern erreicht. In Quelltext 3.6 ist dargestellt, wie der Sprachkonstruktknoten pacmView_Pacman von AVPs erbt, um so die intendierte Animation zu erreichen. Wird eine Änderungsoperation an einem Sprachkonstruktknoten ausgelöst, werden vom Animationsframework die an den Knoten gebundenen AVPs und nicht die Standardanimationsaktionen benutzt.

```
SYMBOL pacmView_Pacman INHERITS AVPOnMoveRotate, AVPOnMoveMove
COMPUTE
  SYNT.compactObject_drawn = 1 <- THIS.canDraw;
  SYNT.onMoveRotateAngle = THIS.pers_angle;
  SYNT.onMoveRotateClockwise = THIS.pers_clockwise;
  SYNT.onMoveRotateDuration = 600;
  SYNT.onMoveMoveRaiseDisplayOrder = 1;
  SYNT.onMoveMoveAnimationTime = 2;
  SYNT.onMoveMoveDuration = 900;
END;
```

Quelltext 3.6: Animation des Pac-Man.

Zur Animation des Pac-Man wird das AVP AVPOnMoveRotate, welches auf die Änderungsoperation MOVE reagiert, verwendet, um zu erreichen, dass sich der Pac-Man innerhalb seiner Ausgangskachel in die entsprechende Richtung dreht. Dafür

werden die Kontrollattribute `onMoveRotateAngle` und `onMoveRotateClockwise` überschrieben und ihnen der Winkel und die Angabe, ob die Bewegung im Uhrzeigersinn erfolgen soll, zugewiesen. Diese beiden Informationen, die in Attributen der Klasse `Pacman` gespeichert sind, wurden in dem Simulationsereignis `computeRotation` berechnet. Außerdem wird das AVP `AVPOnMoveMove` verwendet, um die Bewegung von der Ausgangs- zur Zielkachel zu erreichen. Um zu gewährleisten, dass zunächst die Drehung innerhalb der Ausgangskachel und dann die Bewegung zur Zielkachel durchgeführt wird, muss dem Kontrollattribut `onMoveMoveAnimationTime` der Wert 2 zugewiesen werden, um sicherzustellen, dass die Bewegung nach der Rotation durchgeführt wird.

```
SYMBOL pacmView_Pacman INHERITS AVPCreateDynamicObject,
  AVPMoveDynamicObject, AVPOnRemoveShrink
COMPUTE
  SYNT.createDynamicObjectModificationAction = REMOVE;
  SYNT.createDynamicObjectDrawing = ADDROF(SkullDrawing);
  SYNT.createDynamicObjectPosition = POSITION(
    SELECT(THIS.oPosition, sub(VLPoint(0,10))));
  SYNT.moveDynamicObjectDuration = 8000;
  SYNT.moveDynamicObjectStartPosition = POSITION(
    SELECT(THIS.oPosition, sub(VLPoint(0,10))));
  SYNT.moveDynamicObjectEndPosition = POSITION(
    SELECT(THIS.oPosition, sub(VLPoint(0,60))));
  SYNT.onRemoveShrinkAnimationTime = 10;
END;
```

Quelltext 3.7: Animation eines dynamischen Objektes.

In Quelltext 3.7 sind Attributberechnungen zu sehen, die notwendig sind, um die Animation eines dynamischen Objektes zu realisieren. Dies ist erforderlich, um den Totenkopf darzustellen, wenn Pac-Man gegessen wird. Das Besondere an dynamischen Objekten ist, dass sie keine Repräsentation im semantischen Modell haben. Um diese Objekte zu realisieren, wird das AVP `AVPCreateDynamicObject` verwendet, welches durch Überschreiben des Kontrollattributs `createDynamicObjectModificationAction` auf die Änderungsoperation `REMOVE` reagiert. Das Bild des Totenkopfs wird oberhalb des Pac-Man dargestellt und bewegt sich durch Verwendung des AVPs `AVPMoveDynamicObject` nach oben, bis es nach kurzer Zeit verschwindet.

4. Aspekte der Implementierung

In diesem Kapitel sollen interessante Aspekte der Implementierung beschrieben werden, die über die grundlegenden Implementierungskonzepte aus Kapitel 3 hinaus gehen. Zunächst werde ich die *Initialisierung* von Sprachobjekten beschreiben, die dafür sorgt, dass neu eingefügten Objekten bestimmte Werte zugewiesen werden. Danach gehe gehe ich auf *Konsistenzprüfungen* ein, die sicherstellen, dass Instanzen der visuellen Sprache gewissen Anforderungen genügen. Im Anschluss daran beschreibe ich die *Synchronisation*, die u.a. dafür sorgt, dass die Referenzen jeder Kachel zu ihrer Spalte richtig gesetzt werden. Dann gehe ich auf die *Strategien* ein, nach denen die Geister den Pac-Man jagen. Dazu zählt eine zufällige Strategie, eine Strategie der schrittweisen Annäherung, bei der der euklidische Abstand zwischen Geist und Pac-Man betrachtet wird und die etwas kompliziertere Strategie des Bergsteigens. Anschließend beschreibe ich das Paradigma *aspektorientierte Programmierung* sowie deren konkrete Anwendung im Kontext des in dieser Arbeit entwickelten Editors.

4.1. Initialisierung

Sehr oft wird vom Entwickler gewünscht, dass Sprachelementen bei ihrer Instanziierung bestimmte Werte zugewiesen werden können. Dies lässt sich in einfachen Fällen, wie der Zuweisung eines Wertes an ein VAL-Attribut, über die bereits im Grundlagenkapitel beschriebene INIT-Klausel in DSSL lösen. Für komplexere Initialisierungen kann eine Tcl-Funktion folgender Signatur implementiert werden:

```
proc edithook::create_Objekt { obj }
```

Bei dieser Funktion handelt es sich um eine *Callback-Funktion*, die vom System automatisch aufgerufen wird, sobald das neue Programmobjekt in den Baum eingehängt wurde. Dies hat den großen Vorteil, dass mittels Pfadausdrücken ein Durchwandern des Baumes möglich ist und so auch kontextsensitive Initialisierungen durchgeführt werden können.

In Quelltext 4.1 ist die Initialisierungsfunktion für ein Geist-Objekt dargestellt. Darin wird der Variable `random` eine Zufallszahl zwischen eins und drei zugewiesen. Mittels der Hilfsfunktion `setValue` wird dieser zufällige Wert dem Attribut

`strategy` des Geist-Objektes zugewiesen. Durch diese Funktion wird der bereits in Abschnitt 3.1.1 beschriebene Effekt erzielt, dass jedem neu eingefügten Geist eine zufällige Strategie zugeordnet wird.

```
proc edithook::create_Ghost {obj} {
  set random [expr int(rand()*3)+1]
  edit::setValue [c::get $obj.strategy] "$random"
}
```

Quelltext 4.1: Initialisierungsfunktion für ein Geist-Objekt.

Neben der Funktion zur Initialisierung von Geist-Objekten wurden noch weitere Funktionen implementiert. Die Funktion `create_Root` sorgt dafür, dass bei der Instanziierung des Root-Objektes automatisch die untergeordneten Sprachkonstrukte Score, Lives und Matrix eingefügt werden. Mittels der Funktion `create_Matrix` wird eine initiale 10×10 Matrix erstellt, sobald das Matrix-Objekt instanziiert wird. Da das Root-Programmobjekt in den Baum eingehängt wird, sobald der Benutzer im Editor ein neues Dokument anlegt, wird an dieser Stelle die Funktion `create_Root` aufgerufen. Da in dieser ein neues Matrix-Objekt eingefügt wird, wird danach sofort die Funktion `create_Matrix` aufgerufen, die das initiale Spielfeld erstellt. Dieser Vorgang sorgt dafür, dass der Benutzer ein leeres Spielfeld vorfindet und nicht von Grund auf das Spielfeld durch Einfügen von Zeilen und Spalten aufbauen muss. Der Benutzer kann selbstverständlich das initiale Spielfeld durch Löschen oder Hinzufügen von Zeilen oder Spalten an seine Anforderungen anpassen.

Um jede einzelne Kachel des Spielfeldes direkt referenzieren zu können, erwies es sich im Laufe der Implementierung als sinnvoll, jeder Kachel ein Attribut für seine kartesische Koordinate zuzuweisen. Um dies zu realisieren, wurde zunächst in der abstrakten Struktur der Klasse `Tile` das Attribut `position` hinzugefügt. Die Initialisierung jeder einzelnen Kachel wurde in einer Tcl-Funktion `create_Tile` realisiert. Mittels Pfadausdrücken wurde zwei Variablen jeweils eine Liste sämtlicher Zeilen und Spalten zugewiesen und zwei weiteren Variablen die ID der aktuellen Zeile bzw. Spalte, mit der die aktuelle Kachel assoziiert ist. Mithilfe einer Iteration über sämtliche Zeilen und einem Vergleich der IDs zwischen aktuell betrachteter Zeile und der Zeile mit der die Kachel assoziiert ist, konnte mithilfe eines Zählers herausgefunden werden, welche Position die Zeile der Kachel im gesamten Spielfeld einnimmt. Die Position der Spalte der Kachel konnte nach dem selben Prinzip ermittelt werden. Durch diese beiden ermittelten Werte konnte nun das Positionsattribut der Kachel gesetzt werden.

Die `create_Tile` Funktion wurde unmittelbar aufgerufen, nachdem das Matrix-Programmobjekt initialisiert wurde und sorgte dafür, dass das Positionsattribut

jeder Kachel auf dem initialen Spielfeld korrekt gesetzt wurde. Zu großen Problemen kam es beim Löschen oder Hinzufügen von neuen Zeilen oder Spalten. Beim Löschen einer Zeile oder Spalte an Position i entstand in der Nummerierung einfach eine Lücke, sodass auf Zeile oder Spalte $i-1$ Zeile oder Spalte $i+1$ folgte. Beim Einfügen einer Zeile oder Spalte an Position i wurde zwar die Initialisierungsfunktion aufgerufen, die den in der Zeile oder Spalte enthaltenen Kacheln auch die korrekte Position zuwies, allerdings fand dieser Aufruf nur für die neu eingefügte Zeile oder Spalte und nicht für die hinter ihr liegenden statt. Dies hatte zur Folge, dass eine Zeilen- bzw. Spaltennummer doppelt vergeben wurde; es entstand eine Folge der Form: $\ldots i-1, i, i, i+1, \ldots$. Da dies die Funktionstüchtigkeit des Pac-Man-Spieles gefährdete, wurde DEViL um die Möglichkeit erweitert, primitive Attribute in das Simulationsmodell auszulagern. Damit einhergehend wurde es ermöglicht, für das Simulationsmodell Initialisierungsfunktionen zu implementieren. Die Signatur einer solchen Funktion lautet:

```
proc edithook::create_SMObjekt { obj }
```

Diese Initialisierungsfunktionen sind mit denen für Programmobjekte aus dem Modell der abstrakten Struktur absolut vergleichbar, arbeiten jedoch im Unterschied nur mit den Objekten des Simulationsmodells, welches durch das Präfix `SM` gekennzeichnet wird. So konnte eine Funktion `create_SMTile` implementiert werden, die genau so arbeitet wie die oben beschriebene Funktion `create_Tile`, allerdings die genannten negativen Auswirkungen vermeidet. Die Initialisierung findet hier erst statt, wenn die Simulation gestartet wird. Es wird das Spielfeld nach der Bearbeitung durch den Benutzer betrachtet und so haben das Löschen oder Hinzufügen von Zeilen oder Spalten keine negativen Auswirkungen auf die Initialisierung des Positionsattributs.

4.2. Konsistenzbedingungen

Manchmal ist es notwendig, dass Instanzen der visuellen Sprache, die mit einem generierten Editor erstellt werden, gewisse Konsistenzbedingungen erfüllen. In einfachen Fällen kann dies bereits bei der Spezifikation der abstrakten Struktur mittels der CHECK-Klausel realisiert werden. Dies ist z.B. der Fall, wenn der Wert eines VAL-Attributs in einem bestimmten Bereich liegen soll oder wenn ein SUB-Attribut auf einen bestimmten Typ hin geprüft werden soll. Für diese Zwecke bietet DEViL eine Reihe vordefinierter Funktionen an, die recht einfach angewandt werden können.

Um kompliziertere und evtl. kontextabhängige Prüfungen zu implementieren, bietet DEViL die Möglichkeit, benutzerdefinierte Prüffunktionen in Form von

Tcl-Funktionen anzugeben. Die Prüffunktionen werden durch einen Aufruf von `checkutil::addCheck` registriert. Die Signatur eines solchen Aufrufs sieht wie folgt aus:

checkutil::addCheck *Objekt.attribut* { *Rumpf* }

Der erste Parameter `Objekt.attribut` dieses Aufrufs beschreibt den zu prüfenden Attributknoten. Der zweite Parameter `Rumpf` ist eine Funktion, die die Prüfung beschreibt und entweder eine Fehlermeldung oder eine leere Zeichenkette zurück gibt. Innerhalb dieser Funktion kann über die Variable `obj` auf das zu prüfende Objekt zugegriffen werden. Mit der Variable `value` kann bei VAL- oder REF-Attributen auf den Wert des Objektes zugegriffen werden.

```
checkutil::addCheck Matrix.rows  {
  set pacmans [c::getList
    {$obj.CHILDREN.tiles.CHILDREN.item.CHILDREN[Pacman]}]
  if { [llength $pacmans] > 1 } {
    return "There is only one Pac-Man allowed."
  }
  return ""
}
```

Quelltext 4.2: Prüffunktion, die auf mehr als einen Pac-Man aufmerksam macht.

Im Rahmen dieser Arbeit wurden mehrere Prüffunktionen implementiert, die z.B. prüfen, ob sich mehr als ein Pac-Man auf dem Spielfeld befindet. Die konkrete Implementierung für dieses Beispiel ist in Quelltext 4.2 zu sehen. Der erste Parameter des Aufrufs von `checkutil::addCheck` bekommt den Attributknoten `Matrix.rows` des abstrakten Strukturbaumes übergeben. Innerhalb des Rumpfes ist die Prüfung implementiert. Dort wird der Variable `pacmans` mittels eines Pfadausdrucks die Liste aller auf dem Spielfeld vorhandenen Pac-Mans zugewiesen. Ergibt die bedingte Abfrage, dass sich mehr als ein Pac-Man auf dem Spielfeld befindet, gibt die Funktion eine Zeichenkette zurück, die auf diese fehlerhafte Situation hinweist. Andernfalls gibt die Funktion eine leere Zeichenkette zurück. Neben dieser Prüffunktion wurden auch noch weitere Funktionen implementiert, die sicherstellen, dass sich auf dem Spielfeld jeweils mindestens ein Pac-Man, ein Geist und eine Kraftpille befindet. Zusammen mit der oben genauer betrachteten Prüffunktion bedeutet dies, dass sich immer nur genau ein Pac-Man auf dem Spielfeld befinden darf.

Sollte der Benutzer im Editor ein nicht korrektes Spielfeld erstellt haben, welches den oben beschriebenen Anforderungen nicht genügt, öffnet sich beim Starten der Simulation ein Fenster, welches auf den misslichen Zustand hinweist. Das Starten der Simulation wird erst möglich, wenn alle Anforderungen erfüllt sind.

4.3. Synchronisation

Es kommt vor, dass bestimmte Konsistenzbedingungen automatisch hergestellt werden sollen. So ist es oft erforderlich, in Listen oder Matrizen automatisch neue Elemente einzufügen, um z.B. eine bestimmte Mindestanzahl zu gewährleisten. Um diese Konsistenzbedingungen automatisch herzustellen, können Tcl-Funktionen folgender Signatur implementiert werden, wobei `KONTEXT` dort für die Klasse eines Objekt- oder Attributknotens steht.

```
proc synchook::KONTEXT { obj }
```

Diese Funktionen werden automatisch aufgerufen, sobald vom Benutzer eine Editieroperation durchgeführt wird. Wurde eine Editieroperation durchgeführt, so stellt sich der genaue Ablauf wie folgt dar: Zunächst werden die neu erzeugten Programmobjekte initialisiert. Danach findet eine iterative Synchronisation der Programmstrukur statt, bis ein stabiler Zustand erreicht ist. Abschließend werden die Konsistenzbedingungen überprüft.

```
proc synchook::Row {obj} {
  set defList [c::get {$obj.INCLUDING Matrix.columns}]
  set repList [c::get {$obj.tiles}]

  syncutil::createMissing $defList $repList columnRef
      createMissingCell
  syncutil::deleteSuperfluous $defList $repList columnRef
  syncutil::order $defList $repList columnRef
}
```

Quelltext 4.3: Synchronisation zwischen Kacheln und Spalten-Referenz.

Im Rahmen dieser Arbeit wurde die Funktion `synchook::Matrix` implementiert, die sicherstellt, dass jeweils mindestens ein Attributknoten `rows` und `columns` der Klasse `Matrix` existiert. Die Funktion `synchook::Row`, deren Implementierung in Quelltext 4.3 zu sehen ist, weist jeder Kachel eine Referenz zu einer Spalte zu. Zunächst wird der Variable `defList` der Attributknoten `columns` und der Variable `repList` der Attributknoten `tiles` zugewiesen. Die Hilfsfunktion `createMissing` aus dem `syncutil`-Modul erzeugt eine fehlende Repräsentation von Kacheln, die Objekte in der Spalte repräsentieren. `columnRef` ist der Name des Kachel-Attributs, das das repräsentierte Spalten-Objekt referenziert. Durch die Funktion `deleteSuperluous` werden Kachel-Objekte gelöscht, die kein Spalten-Objekt referenzieren. Dies wird benötigt, wenn der Benutzer eine Spalte des Spielfeldes löscht. Ändert der Benutzer die Reihenfolge der Spalten, kommt die Hilfsfunktion `order` zur Anwendung, die die Reihenfolge der Kachel-Elemente ändert, sodass diese der Reihenfolge der repräsentierten Spalten-Elemente entspricht.

4.4. Strategien

In diesem Abschnitt sollen die Strategien, die in dem Spiel von den drei verschiedenen Geistern verwendet werden, beschrieben werden. Jede einzelne Strategie soll zunächst informell erklärt werden, bevor danach etwas genauer auf Details der Implementierung eingegangen wird. Die Ideen für die drei Strategien stammen von Alexander Repenning [Rep06].

Da besonders für die Implementierung dieser Strategien die C-Funktionen benutzt werden, die an vorherigen Stellen schon kurz erwähnt wurden, möchte ich jetzt die Bedeutung dieser Funktionen im Kontext der gesamten Implementierung beleuchten.

Das Spielfeld ist als Matrix aufgebaut, welche DEViL-intern als Liste von Listen gespeichert wird. Innerhalb der Simulation wird immer auf einem Objekt (z.B. Pac-Man- oder Geist-Objekt) aus dem erweiterten Simulationsmodell gearbeitet, von dem nur die Kachel bekannt ist, auf dem es sich selber befindet. Und genau diese Kachel ist ein Element in einer Liste von Listen. Allerdings ist es notwendig, von dieser Kachel aus zu anderen Kacheln zu navigieren. Genau für diesen Zweck wurden in C Hilfsfunktionen implementiert, die es möglich machen, von der Datenstruktur der „Listen von Listen" zu abstrahieren, um so auf einem höheren Niveau arbeiten zu können. Zu diesem Zweck ist das Attribut `position` jeder Kachel besonders von Vorteil, da dieses der Hilfsfunktion übergeben werden kann und so einen Array-artigen Zugriff erlaubt.

So wurde eine C-Funktion `getTiles` implementiert, die genau das eben beschriebene realisiert. Sie bekommt die Position der gesuchten Kachel übergeben, findet diese mittels zweier weiterer Hilfsfunktionen in der Liste von Listen und kann diese Kachel dann als Rückgabewert zurückgeben. Die zwei weiteren Hilfsfunktionen sind zum einen die Funktion `getTilesFromTiles`, die innerhalb einer der Spalten in einer vorgegebenen Zeile arbeitet und zum anderen die Funktion `getRow`, die eine Zeile an vorgegebener Position zurückliefert.

Bei der Implementierung der Strategien, die auch mit C-Funktionen realisiert wurden, kann nun durch die Funktion `getTiles` recht elegant auf andere Kacheln zugegriffen werden. Auch die in Abschnitt 3.2.2 erwähnten und innerhalb der Simulationschleife verwendeten Funktionen wurden in C implementiert und nutzen den Vorteil des einfachen Zugriffs auf die anderen Kacheln.

4.4.1. Zufall

Der Geist der nach der zufälligen Strategie agiert, bewegt sich zufällig über das Spielfeld. Vor jedem seiner Schritte werden die möglichen Kacheln evaluiert, auf

die er sich bewegen kann. Aus diesen möglichen Kacheln wird dann zufällig eine ausgewählt.

Bei diesen Kacheln handelt es sich immer um die Kacheln unmittelbar rechts, links, oberhalb und unterhalb der aktuellen Kachel. Diese Art der Nachbarschaft wird auch als *von-Neumann-Nachbarschaft* [PW08] bezeichnet. Eine Kachel hat weniger als vier Nachbarn, wenn sie direkt am Rand des Spielfeldes liegt. In diesem Fall ist die Menge der Kacheln, auf die er sich bewegen kann, natürlich geringer und es wird aus dieser kleineren Menge zufällig eine Kachel ausgewählt. Gleiches gilt für den Fall, wenn in der Nachbarschaft der Kachel ein Wandelement ist.

Um dies zu realisieren, wurde die C-Funktion `getRandomNeighbour` implementiert, die alle freien Nachbarkacheln betrachtet und aus diesen eine zufällig auswählt.

4.4.2. Schrittweise Annäherung

Ziel bei dieser Strategie ist es, dass der Geist dem Pac-Man schrittweise immer näher rückt. Dafür werden vor jedem Schritt die freien Kacheln n_i in der von-Neumann-Nachbarschaft betrachtet und für jede die euklidische Distanz zwischen sich selber und der Kachel p, auf der sich Pac-Man befindet, berechnet. Außerdem wird die Distanz zwischen der aktuellen Kachel des Geistes und der Kachel des Pac-Man berechnet. Die euklidische Distanz berechnet sich wie folgt:

$$d(n_i, p) = \sqrt{(n_{ix} - p_x)^2 + (n_{iy} - p_y)^2}$$

Der Geist bewegt sich auf diejenige Nachbarkachel mir geringster euklischer Distanz, sofern dies die Distanz zu Pac-Man verringert. Für diese Strategie wurde die Funktion `getNearestNeighbourToPacman` implementiert.

4.4.3. Bergsteigen

Ausgangspunkt für diese Strategie ist die Strategie der schrittweisen Annäherung, die oft sehr ineffektiv sein kann. Angenommen, der Geist und Pac-Man stehen sich, nur durch eine Wand voneinander getrennt, direkt gegenüber. Dann würde die kürzeste Distanz vom Geist zu Pac-Man durch die Wand verlaufen. Die einzig möglichen Bewegungen, nach rechts oder links, würden den Geist nur weiter von Pac-Man entfernen.

Durch die Strategie des Bergsteigens, die auch in einem Pac-Man-Spiel [Ageb] von Agentsheets implementiert wurde, kann die eben beschriebene Hilflosigkeit des Geistes verhindert werden. Dies geschieht durch das Konzept der *Antiobjects*, welches darauf beruht, dass die gesamte Berechnungsarbeit nicht auf dem Geist lastet, sondern zu Teilen auf die Kacheln übertragen wird. Dafür wird jede Kachel

als ein solches Antiobject aufgefasst und enthält Informationen, die dem Geist bei der Suche nach Pac-Man helfen. Ausgangspunkt hierfür ist die sogenannte *Diffusion*. Jede Kachel bekommt ein Attribut `diffVal`, welches einen beliebigen aber konstanten Diffusionswert beschreibt. Dieser wird dafür benutzt, die Witterung nach Pac-Man aufzunehmen. Der Wert ist ein Indikator dafür, wie nah der Geist dem Pac-Man gekommen ist. Den höchsten Wert bekommt diejenige Kachel zugewiesen, auf der sich Pac-Man befindet. Ausgehend von dieser Kachel verteilt sich der Indikator auf alle begehbaren Kacheln des Spielfeldes. Diese Technik wird als Diffusion bezeichnet.

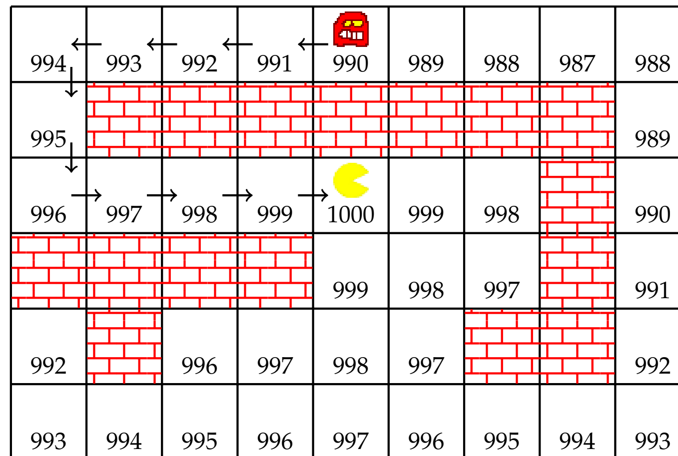

Abbildung 4.1.: Veranschaulichung der Diffusion.

In Abbildung 4.1 ist ein kleines Beispiel skizziert, welches die Verteilung des Diffusionswerts über das Spielfeld zeigt und den Weg weist, den der Geist gehen wird, um den Pac-Man zu jagen. Betrachtet man diejenigen Kacheln, die kein Wandelement enthalten als Knoten und die von-Neumann-Nachbarschaften zwischen diesen Kacheln als Kanten eines Graphen, so ist es naheliegend, die Diffusionswerte durch eine *Breitensuche* zu berechnen, die auf der Kachel, auf der sich Pac-Man befindet, startet.

Diese Breitensuche wurde ebenfalls in einer C-Funktion implementiert. Da sich die Diffusionswerte nach jeder Bewegung des Pac-Man ändern, wird diese Funktion während jedes Durchlaufs der Simulationsschleife von neuem ausgeführt. Um den Geist auf die Nachbarkachel mit maximalem Diffusionswert zu navigieren, wurde die Funktion `getMaxDiffValNeighbour` implementiert, die aus allen Nachbarkacheln diejenige mit dem maximalen Diffusionswert auswählt.

4.5. Aspektorientierte Programmierung

Die aspektorientierte Programmierung[1] ist ein Programmierparadigma, welches anstrebt, sogenannte *Cross-Cutting Concerns* zu modularisieren. Im nachfolgenden Abschnitt werde ich zunächst das Paradigma genauer erklären und an einem einfachen Beispiel seine Anwendung zeigen. Dabei handelt es sich um ein Beispiel der aspektorientierten Programmiersprache *Aspect*C++, da sämtlicher Quelltext, der mit DEViL spezifiziert wird, in C++-Code übersetzt wird und somit aspektorientierter Code auf C++-Ebene erstellt werden muss. Danach werde ich beschreiben, wie die aspektorientierte Programmierung bei der Entwicklung des Pac-Man-Editors zum Einsatz kam und welche Vorteile sich daraus ergaben.

4.5.1. Beschreibung des Paradigmas

Komplexe Softwareprojekte sind nur durch Modularisierung beherrschbar. Dies erhöht die Übersichtlichkeit und erleichtert die Wartung und Weiterentwicklung. Außerdem lässt sich durch Modularisierung Software effizienter wiederverwenden und besser konfigurieren. So lassen sich in objektorientierten Programmiersprachen konzeptionell zusammengehörige Variablen und Funktionen in Klassen mit fest definierten Schnittstellen kapseln. Allerdings gibt es Fälle, in denen herkömmliche Sprachen keine Modularisierungsmöglichkeiten anbieten.

Ein solcher Fall ist z.B. die Ablaufverfolgung von Programmen, bei denen jeder Ein- und Austritt in bzw. aus einer Funktion bzw. Methode mitprotokolliert wird. Um dies zu realisieren, muss am Beginn und am Ende jeder Funktion ein Aufruf einer Funktion aus einer Protokollklasse stehen. Diese Aufrufpunkte sind also über den gesamten und beliebig komplexen Quelltext verstreut. Man spricht in diesem Fall von einem Cross-Cutting Concern, weil die Aufrufe der Protokollklasse die Belange jeder anderen Funktion quer schneiden.

Die aspektorientierte Programmierung hat zum Ziel, diese Cross-Cutting Concerns zu modularisieren. So lässt sich redundanter Quellcode vermeiden, die Wartbarkeit und Konfigurierbarkeit erhöhen und dafür sorgen, dass der Quelltext das dahinter stehende Design besser widerspiegelt. Um dies zu erreichen, werden Aspekte beschrieben. Diese Aspekte sind Module zur Implementierung von Cross-Cutting Concerns und stellen eine nicht-modulare Einheit für nichtmodularen Code dar. Durch die Definition von Aspekten wird der Aspektcode, der einen Cross-Cutting Concern beschreibt, von dem eigentlichen Komponentencode separiert. Spätestens zum Ausführungszeitpunkt werden Komponentencode und Aspektcode wieder miteinander kombiniert. Dieser Vorgang wird als *Weben* bezeichnet.

[1] Vgl. [SG02], [SLU05], [SU06] und [SL07].

Aspekte sind syntaktisch mit Klassen zu vergleichen, da sie Attribute und Methoden definieren können und außerdem von Klassen oder abstrakten Aspekten erben können. Für jeden Aspekt wird mindestens ein Objekt instanziiert, um seinen Zustand zu speichern. Die Implementierung der Cross-Cutting Concerns geschieht mittels sogenannter *Advices*. Innerhalb eines Advices kann Quelltext spezifiziert werden, der ausgeführt wird, wenn sogenannte *Zielpunkte* erreicht werden. Diese Zielpunkte sind Einfügestellen im Komponentencode. In dem Beispiel der Ablaufverfolgung befindet sich am Beginn und am Ende jeder Funktion eine solche Einfügestelle. Allgemein gibt es zwei verschiedene Arten von Zielpunkten: Bei der Ausführung einer Funktion und bei dem Aufruf einer Funktion. Zielpunkte werden in einem Advice durch die Angabe von *Verbindungspunkt-Ausdrücken*, die *Verbindungspunkte* beschreiben, angegeben. Verbindungspunkte sind eine Menge von Zielpunkten und werden anhand der Programmstruktur lokalisiert. Verbindungspunkt-Ausdrücke werden durch *Matching-Ausdrücke* zusammengesetzt, die eine Zeichenkette darstellen, deren Muster eine Menge von Zielpunkten beschreibt.

Jedes Advice kann durch eine funktionale Eigenschaft genauer spezifiziert werden. So kann durch das Schlüsselwort `before()` angegeben werden, dass der im Advice definierte Code vor den, durch Verbindungspunkte angegebenen, Zielpunkten innerhalb des Komponentencodes ausgeführt wird. Durch Angabe von `after()` wird der definierte Code nach den Zielpunkten ausgeführt. Das Schlüsselwort `around()` gibt an, dass der Code an Stelle der Zielpunkte ausgeführt wird.

```
aspect Tracing{
  advice execution ("% ...::%(...)") : before () {
    std::printf ("in %s\n", JoinPoint::signature ());
  }
};
```

Quelltext 4.4: Ablaufverfolgungsaspekt in AspectC++.

In Quelltext 4.4 ist ein Beispiel für einen Ablaufverfolgungsaspekt in AspectC++ zu sehen. Durch das Schlüsselwort `execution` wird angegeben, dass sich der Zielpunkt bei der Ausführung einer Funktion befindet. Das Muster des Verbindungspunkt-Ausdrucks `excecution ("% ...::%(...)")` passt zu sämtlichen Funktionen in jedem Namensraum und jeder Klasse. Durch die Angabe dieses Beispiels wird jedes Eintreten in jede Funktion durch eine `printf`-Anweisung mitprotokolliert.

4.5.2. Einsatz bei der Implementierung

Es kann notwendig sein, das Modell bzw. das Simulationsmodell auf bestimmte Eigenschaften hin zu untersuchen. Da diese Eigenschaften im gesamten Modell verteilt sein können, lässt sich diese Überprüfung der Eigenschaften nicht einfach in der Spezifikationssprache DSIM implementieren. Ein solcher Fall ergab sich bei der Implementierung des Editors zweimal. Zum einen soll in einem Dialog angezeigt werden, dass das Spiel gewonnen wurde, wenn alle Kraftpillen gegessen wurden. Zum anderen soll bei einem verlorenen Spiel in Abhängigkeit von den noch vorhandenen Leben des Pac-Man ein spezieller Dialog angezeigt werden.

```
aspect Counter {
  advice execution ("% Simulator::gameLost_Event(...)") &&
        that(simulator) : after(Simulator & simulator){
    //bestimme Attribut "lives" und dekrementiere es
    if(lives == 0){
      vlStopSimulation();
      vlShowInformationDialog
        ("Game definitely lost!", "Information", "lost");
    }
    else{
      VLString question = vlShowConfirmDialog
        ("Live lost! New Game?", "Ok", "Cancel", "lost");
      if(strcmp(question.getCharPtr(), "Ok") == 0){
        //Aufruf des addNewPacman-Events
      }
      else{
        vlStopSimulation();
      }
    }
  }

  advice execution ("% Simulator::gameWon_Event(...)") &&
        that(simulator) : after(Simulator & simulator){
    if(simulator.getPowerPills()->size() < 1){
      vlStopSimulation();
      vlShowInformationDialog
        ("Game won!", "Information", "won");
    }
  }
};
```

Quelltext 4.5: Aspekt zum Zählen von Leben und Kraftpillen.

In Quelltext 4.5, in dem an zwei Stellen von Implementierungsdetails, die an dieser Stelle zu weit führen würden, abstrahiert wurde, ist ein Aspekt zum Zählen von Leben und Kraftpillen zu sehen. Dafür werden zwei Advices definiert, die an

den Funktionen `Simulator::gameLost_Event` bzw. `Simulator::gameWon_Event` anknüpfen und am Ende jeder dieser Funktionen den spezifizierten Code ausführen. Bei diesen beiden Funktionen handelt es sich um die C++-Funktionen, die durch den Übersetzungsprozess von DEViL aus den in DSIM spezifizierten Ereignissen `gameLost` und `gameWon` generiert wurden. Diese beiden Ereignisse wurden in der Simulationsdatei ohne Inhalt angelegt, um mittels aspektorientierter Programmierung ihre Funktionalität zugewiesen zu bekommen. Das Ereignis `gameLost` wird immer am Ende des `eatPacman`-Ereignisses aufgerufen und der Aufruf des Ereignisses `gameWon` geschieht immer am Ende des `eatPowerpill`-Ereignisses.

Der mit AspectC++ spezifizierte Code sorgt dafür, dass ein Dialog angezeigt wird, der mitteilt, dass das Spiel endgültig verloren ist, wenn der Pac-Man keine Leben mehr hat. Hat Pac-Man noch mindestens ein Leben, wird auch ein Dialog angezeigt, in dem mitgeteilt wird, dass ein Leben verloren wurde. Allerdings bietet dieser Dialog die Möglichkeit ein neues Spiel zu starten. Wird diese Alternative gewählt, wird das `newPacman`-Ereignis aufgerufen, welches einen neuen Pac-Man auf einer zufällig bestimmten leeren Kachel erzeugt. Möchte der Benutzer hingegen nicht weiterspielen, wird die Simulation gestoppt.

Das andere im Quelltext dargestellte Advice sorgt dafür, dass ein Dialog anzeigt, dass das Spiel gewonnen wurde, wenn alle Kraftpillen gegessen wurden.

5. Evaluation

In diesem Kapitel soll der Implementierungsprozess, den ich zur Erstellung des regelbasierten Spieles durchlaufen habe, evaluiert werden. Im ersten Abschnitt betrachte ich die Komplexität der Implementierung, indem ich den Aufwand für die einzelnen Teile der Spezifikation gegenüberstelle. Außerdem stelle ich eine Performanzuntersuchung für das matrixartige Spielfeld vor. Anschließend stelle ich eine Möglichkeit vor, wie die Sprache DSIM um eine regelbasierte Syntax erweitert werden kann. Dies bringt gleich zwei Vorteile mit sich: Zum einen vermeidet die regelbasierte Syntax, dass der Programmierer einen Großteil der C-Funktionen selber schreiben muss und zum anderen wird der DSIM-Quelltext durch den regelbasierten Ansatz sehr viel übersichtlicher und somit besser nachvollziehbar.

5.1. Komplexitäts- und Performanzuntersuchungen

Einführend in dieses Evaluationskapitel habe ich mich von zwei Evaluationsmethoden, die in [SCK07] und [Sch06] angewendet werden, inspirieren lassen. Zunächst habe ich untersucht, wie komplex die Implementierung der einzelnen Teile der Editor-Spezifikation ist. Dafür habe ich das Maß der Anzahl der Quelltextzeilen (LOC[1]) verwendet. Dabei werden allerdings nur die Spezifikationszeilen mitgezählt, die keine Leerzeilen oder Kommentarzeilen sind. Der Vorteil der LOC ist, dass mit ihnen gut die Lösung einer Aufgabe verglichen werden kann, auch wenn die Komplexität verschiedener Zeilen stark variieren kann.

In Tabelle 5.1 ist dargestellt, wie viele LOC die einzelnen Teile der Editor-Spezifikation benötigen. Außerdem ist ihr prozentualer Anteil an der Gesamtspezifikation aufgelistet. Mit 43,1% beansprucht die Implementierung der C-Funktionen den größten Teil. Dies zeigt sehr anschaulich die schon in Abschnitt 4.4 erwähnte herausragende Bedeutung der C-Funktionen. Die Spezifikation der Simulation nimmt mit 23,5% den zweitgrößten Anteil an der gesamten Spezifikation ein. Da die C-Funktionen ausschließlich im Kontext der Simulation genutzt werden, lässt sich sagen, dass der Spezifikationsaufwand für die Simulation insgesamt zwei Drittel der Gesamtspezifikation ausmacht. Alle anderen Teile der Spezifikation machen zusammen nur ein Drittel des Spezifikationsaufwands aus.

[1] *lines of code*

In der Spalte *generierte LOC* der Tabelle sind die Anzahl der C++-Quelltextzeilen aufgeführt, die durch den Übersetzungsprozess von DEViL entstanden sind. Der große Unterschied zwischen der Anzahl der selber geschriebenen Quelltextzeilen und den daraus generierten Zeilen C++-Code verdeutlicht sehr anschaulich den relativ geringen Aufwand, mit dem sich mit DEViL ein umfangreicher Editor spezifizieren lässt.

Im Abschnitt über die regelbasierte Syntax werde ich Vorschläge machen, wie sich die Aufgaben, die bisher mithilfe der C-Funktionen erledigt wurden, fest in die Syntax von DSIM integrieren lassen. Dadurch wird es in Zukunft möglich sein, bei Editoren für kachel- und regelbasierte Spiele weit weniger C-Code selber schreiben zu müssen.

<table>
<tr><th></th><th>LOC</th><th>Anteil</th><th>generierte LOC</th></tr>
<tr><td>abstrakte Struktur</td><td>40</td><td>4,3%</td><td rowspan="8">278.119</td></tr>
<tr><td>Sicht</td><td>9</td><td>1,0%</td></tr>
<tr><td>Attributberechnungen (Sicht)</td><td>94</td><td>10,1%</td></tr>
<tr><td>Simulation</td><td>218</td><td>23,5%</td></tr>
<tr><td>Attributberechnungen (Animation)</td><td>33</td><td>3,6%</td></tr>
<tr><td>Tcl-Funktionen</td><td>94</td><td>10,1%</td></tr>
<tr><td>C-Funktionen</td><td>400</td><td>43,1%</td></tr>
<tr><td>AspectC++</td><td>40</td><td>4,3%</td></tr>
<tr><td></td><td>928</td><td>100,0%</td><td>278.119</td></tr>
</table>

Tabelle 5.1.: Verteilung des Spezifikationsaufwands.

Nachdem ich den Aufwand für die Spezifikation eines Editors aus Entwicklersicht evaluiert habe, möchte ich nun eine Performanzuntersuchung vorstellen, die mit dem generierten Editor durchgeführt wurde und für den Benutzer von großer Bedeutung ist.

Beim Experimentieren mit dem fertigen Editor und verschieden großen Spielfeldern stellte sich schnell heraus, dass die Reaktionszeit des Editors nach einer Benutzeraktion stark von der Größe des Spielfeldes abhängig ist. Dies veranlasste mich, eine Performanzuntersuchung der Sicht-Aktualisierung nach einer Benutzeraktion durchzuführen. Ein Maß, welches bei dieser Untersuchung zur Anwendung kam, ist das der Anzahl der auf dem Spielfeld vorhandenen Kacheln. Allerdings bezog ich zusätzlich, wie in [Sch06], noch das Maß der *Sprachkonstrukt-Knoten* (SK-Knoten) mit ein. Die SK-Knoten sind all diejenigen Knoten, die sich im Strukturbaum befinden und Instanzen von Klassen der abstrakten Struktur darstellen. Das Maß der SK-Knoten ist als etwas genauer anzusehen, weil es sämtliche im Baum befindliche Knoten und nicht nur die Kachel-Knoten berücksichtigt. Auch

wenn diese (bei hinreichend großen Spielfeldern) die absolute Majorität der Gesamtknoten darstellen.

Bei dieser Untersuchung ging ich so vor, dass ich in der Tcl-Funktion `create_Matrix` die Größe des initial zu erstellenden Spielfeldes angab. Daraus ergab sich automatisch die Anzahl der Kacheln. Die Anzahl der SK-Knoten kann man sich in jedem mit DEViL generierten Editor anzeigen lassen. Zur Ermittlung, wie lange die Sicht-Aktualisierung dauert, habe ich für jede Spielfeldgröße acht Sprachkonstrukte aus der Toolbar-Knopf-Leiste in das Spielfeld eingefügt. Auf der Konsole wurde dann protokolliert, wie lange eine Sicht-Aktualisierung gedauert hat. Aus diesen acht ermittelten Werten habe ich dann das arithmetische Mittel gebildet und gelangte zu den Ergebnis, welches in Tabelle 5.2 zu sehen ist.

Die Messungen wurden auf einem Intel Centrino Mobile Prozessor mit 1,86 GHz Taktfrequenz unter Ubuntu 8.10 durchgeführt. Dabei untersuchte ich sieben verschiedene Spielfeldgrößen, die in Tabelle 5.2 neben der Anzahl der Kacheln und der Anzahl der SK-Knoten aufgelistet sind. Die Zeit in Millisekunden für eine Sicht-Aktualisierung ist in der Spalte *Akt.Zeit* abzulesen. Der Spalte *Akt.Zeit / Kacheln* ist das Verhältnis zwischen der Aktualisierungszeit und der Anzahl der Kacheln zu entnehmen. Entsprechend ist in der Spalte *Akt.Zeit / SK-Knoten* das etwas genauere Verhältnis zwischen Aktualisierungszeit und der Anzahl der SK-Knoten aufgelistet.

Matrix	**Kacheln**	**SK-Knoten**	**Akt.Zeit**	**Akt.Zeit / Kacheln**	**Akt.Zeit / SK-Knoten**
5×5	25	39	191	7,6	4,9
10×10	100	124	522	5,2	4,2
15×15	225	259	1071	4,8	4,1
20×20	400	444	2185	5,5	4,9
25×25	625	679	4853	7,8	7,1
30×30	900	964	11036	12,3	11,4
35×35	1225	1299	23054	18,1	17,7

Tabelle 5.2.: Dauer der Sicht-Aktualisierung nach Programmänderungen.

Wie man sieht, wachsen diese beiden Verhältnisse mit steigender Anzahl der SK-Knoten bzw. Kacheln exponentiell. Dies führt zu dem Ergebnis, dass Spielfelder beliebiger Größe nicht praktikabel sind. Allerdings ist auch zu beachten, dass die Aktualisierungszeiten auf einem Rechner mit schnellerem Prozessor weniger langsam gewesen wären. Trotzdem lässt sich feststellen, dass ein Spiel ab einer Spielfeldgröße von 20×20 nicht mehr sinnvoll erscheint.

5.2. Regelbasierte Syntax

In diesem Abschnitt möchte ich Vorschläge machen, wie sich DSIM um eine regelbasierte Syntax erweitern lässt. Dies hat den Vorteil, dass der Code sehr viel übersichtlicher wird und dass sich die regelbasierte semantische Bedeutung des Quelltexts schon in der Syntax widerspiegelt. Außerdem kann durch diesen regelbasierten Ansatz der Anteil der in C implementierten Hilfsfunktionen erheblich reduziert werden. Dies wird möglich, indem zukünftig aus den neuen Syntaxkonstrukten in DSIM Aufrufe von C-Funktionen generiert werden. Diese C-Funktionen sind Verallgemeinerungen der Funktionen, die ich für den Pac-Man-Editor implementiert habe.

Die Entwicklung hin zu der regelbasierten Syntax möchte ich im Folgenden schrittweise zeigen. Zunächst nehme ich eine Klassifikation der für den Pac-Man-Editor implementierten C-Funktionen vor, um daraus verallgemeinerte Funktionen abzuleiten. Danach mache ich einen Vorschlag, wie ein *Mapping* zwischen Bezeichnern aus einer neuen Anwendung und Bezeichnern der allgemeinen C-Funktionen realisiert werden kann. Abschließend folgen die eigentlichen Vorschläge für eine regelbasierte Syntax und ihre Transformationsvorschriften in Aufrufe der verallgemeinerten C-Funktionen.

5.2.1. Klassifikation und Verallgemeinerung der C-Funktionen

Es gibt grundsätzlich zwei Klassen von C-Funktionen, die ich für den Pac-Man-Editor zur Steuerung der Spielfiguren implementiert habe. Dies sind zum einen die Funktionen, die zur Steuerung des Pac-Man verwendet wurden und eine konkrete Richtungsangabe übergeben bekommen. Zum anderen gibt es die Funktionen, die für die Steuerung der Geister sorgen und nicht nur eine spezielle Nachbarkachel in einer bestimmte Richtung betrachten, sondern innerhalb der gesamten von-Neumann-Nachbarschaft nach der Ergebniskachel suchen.

Zunächst möchte ich die Funktionen zur Steuerung des Pac-Man etwas genauer betrachten, um aufzuzeigen, ob eine Verallgemeinerung möglich ist.

`getNeighbour` Bekommt die Ausgangskachel und eine Richtungsangabe übergeben und liefert die Nachbarkachel in angegebener Richtung zurück, falls diese existiert und zusätzlich noch von keinem anderen Element besetzt ist.

`neighbourHasPowerpill` Arbeitet strukturell so wie die `getNeighbour`-Funktion, gibt jedoch nur dann die Nachbarkachel in angegebener Richtung zurück, wenn diese eine Kraftpille enthält.

neighbourHasGhost Arbeitet so wie die `neighbourHasPowerpill`-Funktion, gibt allerdings die Nachbarkachel nur zurück, wenn das Attribut `eatable` des Geistes, der sich auf der Nachbarkachel befindet, *true* ist.

Die Funktion `getNeighbour` ist schon sehr allgemein und kann so direkt als generalisierte Funktion übernommen werden. Die Funktion `neighbourHasPowerpill` wird derart verallgemeinert, dass die Nachbarkachel nach einem beliebigen Element durchsucht wird. Dies lässt sich lösen, indem der allgemeinen Funktion zusätzlich noch ein Parameter für das Element, nach dem gesucht werden soll, übergeben wird. Eine Generalisierung der `neighbourHasGhost`-Funktion sähe so aus, dass diese Funktion prüft, ob auf der Nachbarkachel das angegebene Element vorhanden ist und zusätzlich überprüft, ob ein beliebiges Attribut dieses Elements einen bestimmten Wert besitzt. Diese Überprüfung des Attributs ist allerdings nicht so einfach, da es einen beliebigen Typ haben kann. Dieser Typ muss allerdings für die Implementierung konkret bekannt sein, weil damit eine Typumwandlung durchgeführt werden muss. Aus diesem Grund wird auf eine Verallgemeinerung dieser Funktion zunächst verzichtet.

Jetzt werde ich kurz die Funktionen beschreiben, die zur Steuerung der Geister implementiert wurden und sie danach ebenfalls auf Generalisierungen hin untersuchen.

hasPacmanInNeighbourhood Bekommt die Ausgangskachel übergeben und liefert die Nachbarkachel zurück, auf der sich Pac-Man befindet. Gibt es keine solche Nachbarkachel wird `NULL` zurückgegeben.

getRandomNeighbour Liefert aus der von-Neumann-Nachbarschaft eine zufällige Kachel zurück, die noch von keinem anderen Element besetzt ist.

getNearestNeighbourToPacman Liefert eine Nachbarkachel zurück, wenn von dieser aus die Distanz zwischen dem Geist und Pac-Man verkürzt wird.

getMaxDiffValNeighbour Liefert diejenige Nachbarkachel zurück, die den größten Diffusionswert besitzt.

Die Funktion `hasPacmanInNeighbourhood` lässt sich verallgemeinern, indem der Funktion das Element übergeben wird, nach dem die Nachbarschaft durchsucht werden soll. So ist es möglich, die Nachbarschaft nach beliebigen Elementen zu durchsuchen. Die Funktion `getRandomNeighbour` ist schon allgemein genug und wird den verallgemeinerten Funktionen hinzugefügt, weil es für zukünftige kachelbasierte Spiele durchaus vorstellbar ist, dass eine zufällige Nachbarkachel gesucht wird. Es kann zukünftig ebenfalls interessant sein, diejenige Nachbarkachel zu erhalten, die die Distanz zu einem bestimmten Element verkürzt. So

wird die Funktion `getNearestNeighbourToPacman` derart verallgemeinert, dass die Distanz zu einem beliebigen Element betrachtet wird. Die Generalisierung der Funktion `getMaxDiffValNeighbour` ist aus ähnlichen Gründen wie bei der `neighbourHasGhost`-Funktion nicht so einfach und wurde darum erst einmal nicht vorgenommen.

Bisher wurde immer nur die von-Neumann-Nachbarschaft einer Kachel betrachtet. Für andere Anwendungen ist es durchaus vorstellbar, dass die *Moore-Nachbarschaft* [PW08] betrachtet werden soll. Diese ist eine Erweiterung der von-Neumann-Nachbarschaft, bei der zusätzlich zu den von-Neumann-Nachbarn auch die vier ihr diagonal benachbarten Kacheln betrachtet werden. Ich habe alle verallgemeinerten Funktionen, die die Nachbarschaft nach einem bestimmten Kriterium durchsuchen, in zweifacher Ausprägung implementiert. Einmal wird die von-Neumann-Nachbarschaft und einmal wird die Moore-Nachbarschaft betrachtet.

5.2.2. Mapping von Bezeichnern

Die verallgemeinerten Funktionen verwenden die Bezeichner für Klassen und Attribute aus der abstrakten Struktur, wie sie dem Klassendiagramm auf Seite 25 zu entnehmen sind. Da die grundlegenden Klassen, wie z.B. die Klassen für die Zeile oder Spalte, die für die abstrakte Struktur eines Editors mit kachelbasiertem Spielfeld notwendig sind, in jeder Implementierung unterschiedlich bezeichnet werden können, muss es ein Mapping der Bezeichner, die in den C-Hilfsfunktionen und in der individuellen Implementierung verwendet werden, geben. Ein solches Mapping ist natürlich nur notwendig, wenn sich die Bezeichner in einer neuen Anwendung von denen, die ich in den C-Funktionen verwendet habe, unterscheiden.

```
MAPPING{
  MyRow(myTiles) ::= Row(tiles);
  MyColumn ::= Column;
  MyTiles(i, p) ::= Tiles(item, position);
}
```

Quelltext 5.1: Mapping von Klassen- und Attributbezeichnern.

Das Mapping wird im Konfigurationsblock angegeben und sollte eine Syntax verwenden, wie sie in Quelltext 5.1 angegeben ist. Durch das Schlüsselwort `MAPPING` wird eine Folge von Ersetzungsregeln eingeleitet, die syntaktisch der aus der Definition von kontextfreien Grammatiken bekannten *Backus Naur Form* ähneln. Auf der linken Seite steht der Bezeichner einer Klasse gefolgt von in Klammern notierten Bezeichnern der Attribute der Klasse, wie sie in den der neuen Implementierung verwendet werden. Auf der rechten Seite müssen die entsprechen-

den Bezeichner der äquivalenten Klassen folgen, wie sie in den C-Hilfsfunktionen verwendet wurden. Die Ersetzungsregel `MyColumn ::= Column` hat die Bedeutung „`MyColumn` lässt sich durch `Column` ersetzen“. Genau diese Ersetzungen aller Bezeichner in der neuen Implementierung muss DEViL bei der Generierung zu ausführbarem C++-Code durchführen, um so die Einheitlichkeit zu den Bezeichnern in den C-Funktionen herzustellen.

5.2.3. Syntaxvorschläge

In diesem Abschnitt möchte ich Vorschläge machen, wie sich in DSIM eine regelbasierte Syntax integrieren lässt, die intern durch Aufrufe der eingangs beschriebenen verallgemeinerten C-Funktionen repräsentiert wird. So soll es ermöglicht werden, dass das Erfragen von bestimmten Eigenschaften in einer Nachbarkachel direkt durch eine Abfrage folgender Form erfolgen kann:

```
IF(THIS.tile[NW] == Item) {...}
```

`THIS` und `Item` sind in diesem Fall Platzhalter und werden in einer konkreten Implementierung durch das aktuell betrachtete Objekt bzw. durch das abzufragende Element ersetzt. Die Abfrage wird wahr, wenn sich auf der Kachel nordwestlich des aktuell betrachteten Objektes das entsprechende Element befindet. Innerhalb der eckigen Klammern wird eine Richtung, entsprechend den Himmelsrichtungen, angegeben. So kann mit `NW` auf die diagonale Moore-Nachbarkachel links oben zugegriffen werden. Die Angabe in Form von Himmelsrichtungen wurde gewählt, weil dies für den Programmierer intuitiver ist, als die Richtungen durch Zahlen zu repräsentieren. Soll z.B. überprüft werden, ob eine Nachbarkachel leer ist, kann dies über das Schlüsselwort `EMPTY` anstatt des gesuchten Elements überprüft werden. Eine genauere Auflistung von allen neuen Abfragen und deren Transformation in C++-Code ist im Anhang A.1 nachzulesen.

Um die Abfragen in einer regelbasierten Syntax zu verwenden, werden sie in einen Regelblock, der eine Regel beschreibt, eingebettet. Der Regelblock wird durch das Schlüsselwort `RULE` eingeleitet und kann von einer in eckigen Klammern gefassten Ganzzahl gefolgt werden, die die bedingte Ausführungsreihenfolge angibt. Danach folgt ein optionaler Name für die Regel und innerhalb von geschweiften Klammern die eigentliche Definition der Regel. Die Menge der Grammatikproduktionen für die Regel-Anweisung sind in A.2.1 nachzulesen.

Die Abfragen, die die Regel definieren, werden immer auf C-Funktionen zurückgeführt, die entweder `NULL` oder eine Kachel, wenn die Bedingung erfüllt ist, zurückliefern. Ist innerhalb einer Regel die Bedingung erfüllt, wird ein Ereignis ausgeführt. Manchmal ist es erwünscht, dass aus einer Folge von Regeln immer nur ein einziges Ereignis ausgeführt wird, egal ob die Bedingungen in weiteren

Regeln erfüllt sind, und somit eigentlich mehrere Ereignisse ausgeführt werden müssten. Um dies sicherzustellen, kann für Regeln die bedingte Ausführungsreihenfolge festgelegt werden. Dies bedeutet, dass die Bedingung einer Regel mit kleinstem Ausführungsindex als erstes überprüft wird. Ist diese Bedingung wahr, wird das für diesen Fall angegebene Ereignis ausgeführt und alle weiteren Regeln ignoriert. Andernfall wird die Bedingung der Regel mit zweitkleinstem Ausführungsindex ausgewertet und das entsprechende Ereignis ausgeführt, wenn die Bedingung wahr ist. Alle weiteren Regeln werden ignoriert. Dieses Prinzip setzt sich für alle hintereinander definierten Regeln fort und sorgt dafür, dass immer nur ein Ereignis ausgeführt wird. In A.2.2 werden zwei Beispiele für Regeln, einmal mit bedingter Ausführungsreihenfolge und einmal ohne, und deren Übersetzung in C++-Code gezeigt.

Zwei Beispiele für die regelbasierte Syntax sind in A.3 zu sehen. Dort wird ein Teil der Implementierung zur Steuerung der Geister dargestellt, wie er für den entwickelten Editor verwendet wurde. Danach wird dieser Code äquivalent durch die regelbasierte Syntax wiedergegeben und es zeigt sich, dass der Quelltext dadurch sehr viel übersichtlicher geworden ist. Abschließend wird noch gezeigt, wie das Ereignis `coordinatePacman` aus Quelltext 3.4 unter Verwendung der regelbasierten Syntax aussehen würde.

6. Resümee und Ausblick

In dieser Arbeit wurde eine visuelle Sprache zur Erstellung von Spielwelten für das regelbasierte Spiel Pac-Man erstellt. Dies umfasst die flexible Möglichkeit für den Benutzer, zunächst die Größe des Spielfeldes individuell festzulegen und danach die für das Spiel benötigten Figuren, die Instanzen von vier verschiedenen Klassen sind, an einer beliebigen Position zu platzieren. Zur Wahrung bestimmter Konsistenzanforderungen an die einzelnen Objekte wurden in Tcl Funktionen implementiert, die sicherstellen, dass sich immer nur eine bestimmte Anzahl von Figuren eines Objekttyps auf dem Spielfeld befinden.

Für derart erstellte Spielwelten wurde eine Simulation mit anschließender Animation implementiert, die das Spielen des Spieles ermöglicht und Benutzerstimuli von der Tastatur in Bewegungen des Pac-Man umsetzt. Neben diesen modalen Ereignissen von außen gibt es in Form der Strategien, nach denen sich die Geister bewegen, auch kontinuierliche Ereignisse, die in jedem Simulationsschritt ausgeführt werden. Bei der Realisierung der Simulation war es besonders wichtig, eine Methode zu entwerfen mit der auf entfernt liegende Kacheln zugegriffen werden kann. Dies wurde realisiert, indem im Simulationsmodell der Kachel-Klasse ein Attribut für die Position hinzugefügt wurde. Um dieses Attribut aus dem erweiterten Simulationsmodell initialisieren zu können, wurde in Tcl eine entsprechende Funktion implementiert. Mithilfe des Positionsattributs und geeigneten C-Funktionen, die von der Matrix-Datenstruktur „Liste von Listen" abstrahieren, wurde es möglich, auf andere Kacheln recht einfach zuzugreifen und dadurch die Interaktion zwischen mehreren Spielfiguren zu ermöglichen. Zur Umsetzung einer geeigneten Animation wurde die Standardanimationsabbildung überschrieben und für diesen Zweck auf verschiedene animierte visuelle Muster zurückgegriffen.

Bei der Evaluation wurde sowohl die Komplexität der Implementierung als auch die Performanz des fertigen Editors untersucht. Anhand des durchlaufenen Implementierungsprozesses konnte mit einer Komplexitätsuntersuchung gezeigt werden, welcher Teil der Implementierung besonders komplex ist. Dabei zeigte sich, dass besonders die Simulation und dort speziell die dafür verwendeten C-Funktionen am komplexesten sind. Die Performanzuntersuchung deckte auf, dass die Reaktionszeit des Editors nach Programmänderungen mit steigender Spielfeldgröße stark zunahm, woraus geschlossen wurde, dass Spiele nur bis zu einer bestimmten Größe des Spielfeldes sinnvoll sind.

Ein wichtiges Ergebnis dieser Arbeit ist die Vorstellung der regelbasierten Syntax für DSIM, die die Möglichkeit eröffnen würde, bei der Implementierung von Regeln gleich eine regelbasierte Syntax zu verwenden. Durch deren Etablierung in DSIM würde in Zukunft der Anteil selbst geschriebener C-Funktionen erheblich minimiert werden. Außerdem hätte dies für den Entwickler den Vorteil, die Regeln, die er programmieren möchte und in denen er auch denkt, gleich in einer angepassten Syntax wiederzufinden. Dies würde den positiven Effekt haben, dass die Programmierung etwas schneller abläuft und der entstandene Quellcode besser zu lesen ist, was der Wartbarkeit und Weiterentwicklung unmittelbar zugute käme.

Neben der Etablierung der regelbasierten Syntax in DSIM könnte eine mögliche Erweiterung des Editors darin bestehen, dem Benutzer die Möglichkeit zu geben, die Regeln des Spieles selber zu definieren. Dies könnte optimalerweise in grafischer und nicht textueller Form stattfinden. Eine Regel kann so durch eine Vor- und eine Nachbedingung beschrieben werden. In der Vorbedingung können z.B. zwei Kacheln dargestellt sein – auf der einen befindet sich Pac-Man, auf der anderen eine Kraftpille. In der Nachbedingung werden diese Kacheln wieder dargestellt, jedoch besetzt der Pac-Man hier die Kachel, wo sich vorher die Kraftpille befand. Eine solche visuelle Darstellung einer Regel, die beschreibt, dass Pac-Man Kraftpillen essen darf, muss interpretiert werden und in DSIM-Quelltext übersetzt werden. Ein noch allgemeinerer Ansatz wäre die Entwicklung eines Editors, mit dem sich Editoren für beliebige regelbasierte Spiele entwickeln lassen.

A. Konkretisierte Syntaxvorschläge

In diesem Anhang möchte ich die in Abschnitt 5.2.3 bereits erklärten Syntaxvorschläge für DSIM genauer beschreiben und vorstellen, wie daraus die verallgemeinerten C-Funktionen aufgerufen werden. Danach werde ich den Aufbau von Regeldefinitionen beschreiben. Abschließend werde ich zwei Beispiele geben, welche die bisherige Implementierung der Implementierung mit regelbasierter Syntax gegenüberstellen.

A.1. Syntax der Abfragen und Aufruf der C-Funktionen

Im Folgenden stelle ich die fünf neuen Syntaxkonstrukte von DSIM und deren Übersetzung in ausführbaren C++-Code vor. Dabei orientiere ich mich noch an dem Beispiel Pac-Man, um konkrete Abfragen angeben zu können.

1. `IF(ghost.tile[E] == EMPTY) {...}`

   Innerhalb der eckigen Klammern wird eine Richtungsangabe in Form einer Himmelsrichtung erwartet. Über das Schlüsselwort `EMPTY` wird überprüft, ob die Kachel in angegebener Richtung leer ist. Im folgenden Kasten ist C++-Code dargestellt, in den die obige Abfrage transformiert werden muss. Der Funktion `getNeighbour` wird die Kachel und die in eine Ganzzahl umgewandelte Richtungsangabe übergeben. Die Funktion liefert die Nachbarkachel in angegebener Richtung zurück, falls diese leer ist.

   ```
   SMTile *t = getNeighbour(ghost->getValue_tile(), 3);
   if(t != NULL) {...}
   ```

2. `IF(pacman.tile[S] == Ghost) {...}`

   Prinzipiell mit der Abfrage aus 1. zu vergleichen, allerdings wird diesmal überprüft, ob sich auf der Nachbarkachel ein Geist befindet. Der C-Funktion wird zusätzlich eine entsprechende Zeichenkette übergeben.

   ```
   SMTile *t = getNeighbour(pacman->getValue_tile(), 5,
       "SMGhost");
   if(t != NULL) {...}
   ```

3. `IF(NB[Neumann](ghost.tile, Pacman)) {...}`

   Durch das Schlüsselwort `NB` soll ausgedrückt werden, dass die gesamte Nachbarschaft nach der Ergebniskachel durchsucht wird. Innerhalb der eckigen Klammern wird angegeben, ob in der von-Neumann- oder der Moore-Nachbarschaft gesucht werden soll. Entsprechend dieser Angabe wird jeweils eine andere C-Funktion aufgerufen, die die Nachbarschaft nach einem bestimmten Element durchsucht.

```
SMTile *t = getNeighbourFromNeumannNeighbourhood(
    ghost->getValue_tile(), "SMPacman");
if(t != NULL) {...}
```

4. `IF(NB[Neumann, RAND](ghost.tile)) {...}`

   Hier soll eine zufällige Nachbarkachel aus der von-Neumann-Nachbarschaft der Kachel `ghost.tile` ermittelt werden.

```
SMTile *t = getRandomNeumannNeighbour(
    ghost->getValue_tile());
if(t != NULL) {...}
```

5. `IF(NB[Neumann, NEXT](ghost.tile, #[0]Pacman.tile)) {...}`

   In diesem Fall wird diejenige Nachbarkachel zurückgegeben, die den Abstand zu der Kachel auf der sich Pac-Man befindet verringert.

```
SMTile *t = getNearestNeumannNeighbour(
    ghost->getValue_tile(),
    (*(getPacmans()))[0]->getValue_tile());
if(t != NULL) {...}
```

A.2. Regelbasierte Syntax

Zunächst werde ich die Grammatikproduktionen für die Regel-Anweisung darstellen, um so den Aufbau von Regeldefinitionen zu verdeutlichen. Danach gebe ich jeweils ein Beispiel für die Definition von Regeln und deren Transformation in C++-Code – einmal mit bedingter Ausführungsreihenfolge und einmal ohne.

A.2.1. Menge der `RuleStmt`-Produktionen

```
RuleStmt: 'RULE' '[' Number ']' Name '{' RuleDef '}'
RuleStmt: 'RULE' '[' Number ']' '{' RuleDef '}'
RuleStmt: 'RULE' Name '{' RuleDef '}'
RuleStmt: 'RULE' '{' RuleDef '}'
```

A.2.2. Beispiele für Regeln in DSIM

Mit bedingter Ausführungsreihenfolge

DSIM:

```
RULE[1] rule1{
  IF(item.tile[W] == EMPTY){...}
}
RULE[2] rule2{
  IF(item.tile[NW] == OtherTile){...}
}
```

C++:

```
SMTile *t = getNeighbour(item->getValue_tile(), 7);
if(t != NULL){...}
else{
  SMTile *t = getNeighbour(item->getValue_tile(), 8,
    "SMOtherTile");
  if(t != NULL){...}
}
```

Ohne bedingte Ausführungsreihenfolge

DSIM:

```
RULE rule1{
  IF(item.tile[W] == EMPTY){...}
}
RULE rule2{
  IF(item.tile[NW] == OtherTile){...}
}
```

C++:

```
SMTile *t = getNeighbour(item->getValue_tile(), 7);
if(t != NULL){...}
SMTile *t = getNeighbour(item->getValue_tile(), 8,
    "SMOtherTile");
if(t != NULL){...}
```

A.3. Zwei Beispiele

A.3.1. Geister-Strategien

An dieser Stelle greife ich die Implementierung einer Strategie eines Geistes auf und zeige zunächst, wie dies herkömmlich in DSIM implementiert wird. Der zweite Ausschnitt zeigt eine äquivalente Implementierung unter Verwendung der entwickelten regelbasierten Syntax.

```
IF(ghost->strategy == 1){
  Tile to = hasPacmanInNeighbourhood(ghost.tile);
  IF(NOTNULL(to) AND (ghost->eatable == 0)){
    FIRE eatPacman(ghost.tile, to) @ TIME_DIRECT;
  }{
    IF(NOTNULL(#[0]Pacman)){
      Tile to = getRandomNeighbour(ghost.tile);
      IF(NOTNULL(to)){
        FIRE goGhost(ghost.tile, to) @ TIME_DIRECT;
      }
    }
  }
}
```

```
IF(ghost->strategy == 1){
   RULE[1] eatPacman{
    IF(NB[Neumann](ghost.Tile, Pacman) AND
        (ghost->eatable == 0)){
      FIRE eatPacman(ghost.tile, t) @ TIME_DIRECT;
    }
  }
  RULE[2] random{
    IF(NB[Neumann, RAND](ghost.tile)){
      FIRE goGhost(ghost.tile, t) @ TIME_DIRECT;
    }
  }
}
```

A.3.2. coordinatePacman-Ereignis

Hier möchte ich kurz zeigen, wie das coordinatePacman-Ereignis aussieht, wenn die regelbasierte Syntax verwendet wird. Dabei ist zu beachten, dass die Regel eatGhost auf herkömmlichem Weg implementiert wurde, da es keine verallgemeinerte Funktion für die neighbourHasGhost-Funktion gibt.

```
coordinatePacman(Pacman pacman, int direction){
  RULE[1] eatPowerpill{
    IF(pacman.tile[direction] == Powerpill){
      FIRE eatPowerpill(pacman.tile, t, pacman, direction)
          @ TIME_DIRECT;
    }
  }
  RULE[2] eatGhost{
    Tile go = neighbourHasGhost(pacman.tile, direction);
    IF(NOTNULL(go)){
      FIRE eatGhost(pacman.tile, go, pacman, direction)
          @ TIME_DIRECT;
    }
  }
  RULE[3] go{
    IF(pacman.tile[direction] == EMPTY){
      FIRE goPacman(pacman.tile, t, pacman, direction)
          @ TIME_DIRECT;
    }
  }
}
```

B. Anmerkungen zur Spezifikation

Dieser Arbeit liegt eine CD mit folgendem Inhalt bei:

- Im Ordner `pacman` befindet sich die in dieser Arbeit erstellte Spezifikation. Diese Spezifikation setzt DEViL in der Version 1.4.1 voraus.
- Im Ordner `tileFunctions` befinden sich die verallgemeinerten C-Funktionen.
- Diese Ausarbeitung im PDF-Format.

Abbildungsverzeichnis

2.1. LEGOsheets-Arbeitsfläche und Regel-Editor. 5
2.2. Simulationsfolge aus dem Spiel des Lebens. 7
2.3. Der Spieler mit den gelben Spielsteinen hat gewonnen. 8
2.4. Bildschirmfoto des originalen Pac-Man-Spieles. 9
2.5. Multi-Fenster Umgebung des Editors für generische Zeichnungen. 10
2.6. Generische Zeichnung eines If-Konstrukts in Nassi-Shneiderman-Diagrammen. 15
2.7. Das Konzept des Simulators. 18

3.1. Editor für Pac-Man-Spielwelten. 23
3.2. Editieren der Geister-Attribute. 24
3.3. Editieren der Kraftpillen-Attribute. 24
3.4. Klassendiagramm des Modells der abstrakten Struktur. 25
3.5. Generische Zeichnung für die Sicht pacmView. 26
3.6. Pac-Man hat das Spiel verloren. 29

4.1. Veranschaulichung der Diffusion. 43

Quelltextverzeichnis

2.1. Spezifikation der abstrakten Struktur. 11
2.2. Definition der Sicht. 13
2.3. Anwendung visueller Muster durch eine attributierte Grammatik. . 14
2.4. Zuordnung einer generischen Zeichnung. 15
2.5. IPTG-Textmuster zur Code-Generierung. 16
2.6. Spezifikation der Code-Generierung. 16
2.7. Simulationsmodell für Petri-Netze. 19
2.8. Simulationsschleife. 19
2.9. Ereignisblock. 20

3.1. Kontextabhängige Zuordnung generischer Zeichnungen. 27
3.2. Konfigurationsblock. 30
3.3. Ausschnitt aus dem Simulationsmodell. 30
3.4. Ausschnitt aus dem Ereignisblock. 32
3.5. Ausschnitt aus der Simulationsschleife. 33
3.6. Animation des Pac-Man. 34
3.7. Animation eines dynamischen Objektes. 35

4.1. Initialisierungsfunktion für ein Geist-Objekt. 37
4.2. Prüffunktion, die auf mehr als einen Pac-Man aufmerksam macht. . 39
4.3. Synchronisation zwischen Kacheln und Spalten-Referenz. 40
4.4. Ablaufverfolgungsaspekt in AspectC++. 45
4.5. Aspekt zum Zählen von Leben und Kraftpillen. 46

5.1. Mapping von Klassen- und Attributbezeichnern. 53

Tabellenverzeichnis

5.1. Verteilung des Spezifikationsaufwands. 49
5.2. Dauer der Sicht-Aktualisierung nach Programmänderungen. 50

Literaturverzeichnis

[Agea] AgentSheets, Inc. http://www.agentsheets.com. [Online; Stand 20. Mai 2009].

[Ageb] AgentSheets, Inc. Ultimate Pacman. http://www.agentsheets.com/Applets/ultimate-pacman/readme.html. [Online; Stand 20. Mai 2009].

[All88] Victor Allis. A Knowledge-based Approach of Connect-Four. The Game is Solved: White Wins. Masterarbeit, Vrije Universiteit, Amsterdam, Oktober 1988.

[CK09] Bastian Cramer und Uwe Kastens. Animation automatically generated from simulation specifications. In *Proceedings of VL/HCC'09, Corvallis, Oregon, USA*. IEEE Computer Society Press, September 2009.

[Cra08] Bastian Cramer. Ein Konzept zur Simulation und Animation visueller Sprachen. Technical Report tr-ri-08-293, Universität Paderborn, Fachbereich Mathematik-Informatik, Juli 2008.

[FH] James Freeman-Hargis. Introduction to Rule-Based Systems. http://ai-depot.com/Tutorial/RuleBased.html. [Online; Stand 25. Mai 2009].

[Fir] Firelight Technologies. FMOD – music and sound effects systems. http://www.fmod.org. [Online; Stand 21. August 2009].

[Gam] The Game Programming Wiki. Rule-Based-AI. http://gpwiki.org/index.php/Rule-Based-AI. [Online; Stand 25. Mai 2009].

[GIL+95] Jim Gindling, Andri Ioannidou, Jennifer Lou, Olav Lokkebo und Alexander Repenning. LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick. In *Proceeding of Visual Languages, Darmstadt, Deutschland*, Seiten 172–179. IEEE Computer Society Press, 1995.

[GR03] Marcus Gallagher und Amanda Ryan. Learning to Play Pac-Man: An Evolutionary, Rule-based Approach. In R. Sarkar et al. (Redaktion), *Proc. Congress on Evolutionary Computation (CEC)*, Seiten 2462–2469, 2003.

[Kas] AG Kastens. *IPTG Documentation*. Fakultät für Elektrotechnik, Informatik und Mathematik – Universität Paderborn. http://devil.cs.upb.de/documentation/iptg/iptg.txt. [Online; Stand 20. Juni 2009].

[Kas09] Uwe Kastens. Vorlesungsfolien: Objektorientierte Programmierung. http://ag-kastens.upb.de/lehre/material/oop/folien/comment-alle.2.pdf, 2009. [Online; Stand 23. Juli 2009].

[Köl00] Jürgen Köller. Game of Life. http://www.mathematische-basteleien.de/gameoflife.htm, 2000. [Online; Stand 11. Juli 2009].

[KPJ98] Uwe Kastens, Peter Pfahler und Matthias Jung. The Eli system. In Kai Koskimies (Herausgeber), *Proceedings of 7th International Conference on Compiler Construction CC'98*, Nummer 1383 in Lecture Notes in Computer Science, Seiten 294–297. Springer Verlag, März 1998.

[Luc05] Simon M. Lucas. Evolving a Neural Network Location Evaluator to Play Ms. Pac-Man. In *IEEE Symposium on Computational Intelligence and Games*, Seiten 203–210, 2005.

[PW08] Lutz Priese und Harro Wimmel. *Petri-Netze*. Springer-Verlag, 2008.

[RC93] Alexander Repenning und Wayne Citrin. Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction. In *IEEE Workshop on Visual Languages, Bergen, Norwegen*, Seiten 77–82. IEEE Computer Society Press, 1993.

[RCT95] Richard Robinson, Devin Cook und Steven Tanimoto. Programming Agents with Visual Rules. In *Proceeding of Visual Languages, Darmstadt, Deutschland*, Seiten 13–20. IEEE Computer Society Press, 1995.

[Rep95] Alexander Repenning. Bending the Rules: Steps Toward Semantically Enriched Graphical Rewrite Rules. In *Proceeding of Visual Languages, Darmstadt, Deutschland*, Seiten 226–233. IEEE Computer Society Press, 1995.

[Rep06] Alexander Repenning. Collaborative Diffusion: Programming Antiobjects. In *OOPSLA 2006, ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, (Portland, Oregon, 2006)*. IEEE Press, 2006.

[SC09a] Carsten Schmidt und Bastian Cramer. *DEViL Benutzerhandbuch*. Fakultät für Elektrotechnik, Informatik und Mathematik – Universität Paderborn, 2009. http://devil.cs.upb.de/documentation/manual-html.gen/main.html. [Online; Stand 17. Juni 2009].

[SC09b] Carsten Schmidt und Bastian Cramer. *Visuelle Muster in DEViL*. Fakultät für Elektrotechnik, Informatik und Mathematik – Universität Paderborn, 2009. http://devil.cs.upb.de/documentation/visualPatterns-html.gen/main.html. [Online; Stand 17. Juni 2009].

[Sch06] Carsten Schmidt. *Generierung von Struktureditoren für anspruchsvolle visuelle Sprachen*. Dissertation, Universität Paderborn, Januar 2006. http://ubdata.uni-paderborn.de/ediss/17/2006/schmidt/disserta.pdf. [Online; Stand 23. August 2009].

[SCK07] Carsten Schmidt, Bastian Cramer und Uwe Kastens. Usability Evaluation of a System for Implementation of Visual Languages. In *Symposium on Visual Languages and Human-Centric Computing*, Seiten 231–238, Coeur d'Alène, Idaho, USA, September 2007. IEEE Computer Society Press.

[SG02] Olaf Spinczyk und Andreas Gal. Aspektorientierung und Betriebssysteme – Tutorium. http://www.aspectc.org/download/gi-tutorium-2002.pdf, 2002. [Online; Stand 20. August 2009].

[SL07] Olaf Spinczyk und Daniel Lohmann. Aspect-Oriented Programming with C++ and AspectC++. http://www.aspectc.org/fileadmin/publications/aosd-2007-tut-2x2.pdf, 2007. [Online; Stand 18. August 2009].

[SLU05] Olaf Spinczyk, Daniel Lohmann und Matthias Urban. AspectC++: an AOP Extension for C++. *Software Developer's Journal*, Seiten 68–74, Mai 2005.

[SU06] Olaf Spinczyk und Matthias Urban. *AspectC++ Language Reference*. pure-systems GmbH, 2006. http://www.aspectc.org/fileadmin/documentation/ac-languageref.pdf. [Online; Stand 18. August 2009].