Ein Konzept zur Simulation und Animation visueller Sprachen

Bastian Cramer
University of Paderborn
Department of Computer Science
Fürstenallee 11, 33102 Paderborn, Germany
bcramer@uni-paderborn.de

Abstract

Die Modellierung von Software in einer bestimmten Domäne und durch eine visuelle Repräsentation wird immer wichtiger. Insbesondere die Simulation und Animation dieser Domäne kann neue Erkenntnisse bringen. In diesem Papier stellen wir das Generatorsystem für visuelle Sprachen DEViL vor und zeigen ein Konzept auf, wie DEViL erweitert werden kann um eine Simulation und Animation einer visuellen Sprache zu erreichen. Der Fokus liegt dabei auf einer universellen Simulation und einer Abbildung auf ein Animationslayout. Dazu werden maßgeschneiderte Spezialsprachen benutzt, die eine einfache Spezifikation erlauben.

1. Einführung

In der Softwareentwicklung werden zur Spezifikation häufig visuelle Sprachen eingesetzt, bekanntes Beispiel ist hierfür UML. Der Vorteil visueller Sprachen gegenüber einer herkömmlichen Entwicklung ist, dass sie sich Metaphern einer speziellen Domäne bedienen können und durch graphische Editoren eine Spezifikation auf einem hohen Abstraktionsniveau ermöglichen. Dies erlaubt es gerade auch Nicht-Experten Software für spezielle Anwendungsgebiete zu entwerfen.

Zur Erstellung solcher graphischer Editoren bieten sich Generatorsysteme, wie zum Beispiel DEVil (Development Environment for Visual Languages) an, das in der Fachgruppe "Programmiersprachen und Übersetzer" der Universität Paderborn entwickelt wird. Dieses System kapselt Expertenwissen zur automatischen Generierung von anspruchsvollen graphischen Strukturediteren. So ist es möglich sehr schnell komplexe Editoren für visuelle Sprachen zu erstellen und damit zu experimentieren.

Der heutige Softwareentwurfsprozess basiert jedoch in der Regel auf statischen Diagrammen, Simulation einer visuellen Sprache, also die dynamische Repräsentation in desmselben Layout ist der nächste Schritt. Gerade für das Verständnis nebenläufiger Systeme sind animierte visuelle Repräsentationen geeignet. Das menschliche visuelle Verständnis ist hier, wo es keinen sequentiellen Kontrollfluss gibt, besonders ausgeprägt, wohingegen in imperativen Sequenzen ein textueller Ausdruck häufig besser geeignet ist.

Durch die Simulation und Animation visueller Sprachen kann nicht nur das Verständnis des spezifizierten Softwaresystems verbessert werden, es kann auch in einem frühen Stadium der Spezifikation bereits dazu führen, dass potentielle Fehler erkannt und beseitigt werden können.

In diesem Papier zeigen wir, wie das bestehende Generatorsystem für visuelle Sprachen – DEViL – um die Aspekte Simulation und Animation erweitert werden kann. Wir zeigen, dass dies mit bereits vorhandenen Konzepten realisiert werden kann und dass der Spezifikationsprozess durch spezielle DSLs, die die Simulation beschreiben, entscheidend erleichtert wird.

Im nächsten Kapitel geben wir zunächst eine kurze Einführung in das Spezifikationskonzept von DEViL. Es wird gezeigt, welche Teilschritte nötig sind um eine visuelle Sprache zu entwerfen.

Im dritten Kapitel wird das Simulationsframework beschrieben. Dazu werden zunächst kurz einige visuelle Sprachen dargestellt und diese hinsichtlich ihrer Simulierbarkeit kategorisiert. Danach werden die Anforderungen an den Simulator beschrieben und die Simulationssprache *DSIM* eingeführt.

Im Kapitel 7 wird ein Konzept zur Animation visueller Sprachen vorgestellt. Verwandte Arbeiten und ein Ausblick beschließen dieses Papier.

2. DEViL

Das Generatorsystem DEViL kann aus Spezifikationen hohen Niveaus graphische Strukturediteren generieren. Die generierten Editoren bieten eine Multi-Fenster Umgebung mit allen herkömmlichen Funktionen aktueller Editoren

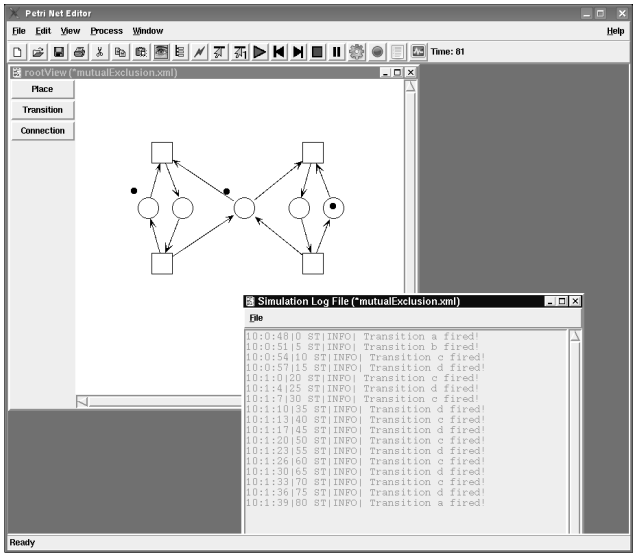

Figure 1. Ein generierter Petri-Netz Editor, der ein Netz simuliert

wie Copy-And-Paste, Drucken, Speichern und Laden von Beispielen sowie beliebig viele Sichten auf das Modell der Sprache. Die generierten Editoren basieren auf dem Konzept des strukturierten Editierens, d.h. es können jederzeit Sprachkonstrukte (auswählbar über eine Toolbar) eingefügt werden, die auch syntaktisch zur Sprache passen. Somit können nur syntaktisch korrekte Instanzen der Sprache existieren, siehe Abb. 1. (Eine Diskussion zu Eigenschaften generierter Umgebungen und Usability ist in [7] zu finden.)

Des Weiteren bietet DEViL umfangreiche Funktionen zur semantischen Analyse wie Karadinalitäten, Check Funktionen und das Einbinden von benutzerdefinierten Codes. Da DEViL das Übersetzergenerator-Framework Eli [4] benutzt, können alle Funktionen des Eli Systems ebenfalls benutzt werden, insbesondere die Spezifikation von Codegeneratoren, die eine Source-to-Source Übersetzung der visuellen Sprache in eine Zielsprache definieren. So kann beispielsweise aus einem UML Klassendiagrammeditor Java Code generiert werden.

Um die Strukturediteren zu spezifizieren muss zunächst das semantische Modell, also die abstrakte Struktur der Sprache definiert werden. Dies geschieht in DEViL mittels DSSL, einer Spezialsprache, die objektorientierte Konzepte wie Klassen, Assoziation, Aggregation, Vererbung und Definition von Attributen erlaubt (siehe Abb. 2). Nur mit der abstrakten Struktur kann DEViL bereits einen einfachen Strukturediteur generieren. Dieser erlaubt es in einem Strukturbaum Sprachkonstrukte einzufügen. Um eine anspruchsvollere Repräsentation zu erreichen, können sogenannte visuelle Muster auf das semantische Modell der

```
CLASS Root {
  nodes: SUB Node*;
  connections: SUB Connection*;
  setSize: VAL VLPoint? INIT "300 300";
}

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

CLASS Place INHERITS Node {
  marks: VAL VLInt INIT "0";
}

CLASS Transition INHERITS Node {

}

CLASS Connection {
  from: REF Node EDITWITH "None";
  to: REF Node EDITWITH "None";
  position: VAL VLPoint? EDITWITH "None";
  weight: VAL VLInt INIT "1";
}
```

Figure 2. Spezifikation der abstrakten Struktur für einen Petri-Netz Editor

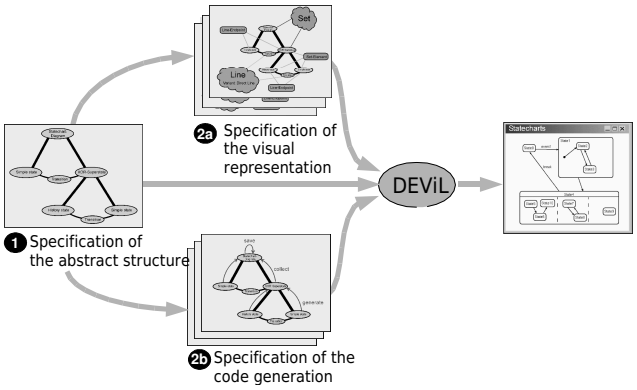

Figure 3. Spezifikationsprozess in DEViL

Sprache angewandt werden (siehe Abb. 3, Schritt 2a).

Visuelle Muster beschreiben, wie Teilbäume der Sprache graphisch dargestellt werden sollen und sind ein abstraktes Konzept. So kann deklarativ beschrieben werden, dass ein Teilbaum als Liste und eine Untermenge als Listenelemente dargestellt werden sollen. DEViL stellt eine ganze Reihe visueller Muster als Bibliothek zur Verfügung. Diese Muster können nahezu beliebig miteinander kombiniert werden. Eine Auswahl dieser Muster sind beispielsweise, Mengen, Listen, Formulare, Matrizen oder Tabellen. Alle Muster können in Aussehen und Layout durch Kontrollattribute parametrisiert werden.

Technisch betrachtet werden visuelle Muster mittels attributierter Grammatiken an einen Strukturbaum (der von DSSL generiert wird) gebunden. Der Attributauswertergenerator LIGA [5] des Eli Systems berechntet auf einer Instanz des Strukturbaums die Attributwerte und somit die graphische Darstellung.

Zur semantischen Analyse und zur Transformation der VL ist es wichtig, spezielle Konstrukte zu haben um im Strukturbaum zu wandern und Werte auszulesen. Dies wird in DEViL durch die Pfadausdrücke erreicht. Sie erlauben einen Zugriff auf Knoten und Knotenmengen und sind vergleichbar mit XPath Ausdrücken in XML Dokumenten [10].

3. Konzept des Simulatorframeworks

Bevor das Simulator-Framework hier vorgestellt wird, soll zunächst der Begriff Simulation zur Animation abgegrenzt werden. Die Simulation einer visuellen Sprache betrachten wir als eine Menge von Transformationen des semantischen Modells der Sprache. Hier wird zunächst von der graphischen Repräsentation abstrahiert, d.h. hier spielt eine Animation oder ein bestimmtes Layout noch keine Rolle. Die Simulation beschreibt die reine Ausführungssemantik einer visuellen Sprache.

Dies ist vergleichbar mit Simulationen wie man sie im herkömmlichen Sinne versteht, als Ausführung eines abstrakten Modells, das eine Datenmenge liefert, die dann mit der Visualisierung graphisch aufbereitet wird.

Ein wichtiger Begriff, wenn es um die Simulation geht, ist das Simulationsmodell. Das Simulationsmodell ist die Grundlage der Simulation, es stellt eine abstrakte Basis für die Simulation dar. Es speichert Zustände der Simulation und muss an die Anforderungen einer Simulation angepasst sein, d.h. es muss so einfach wie möglich sein um keine unnötig langen Simulationszeiten zu provozieren, es muss jedoch so umfangreich sein, dass es korrekte Ergebnisse liefert.

Das Simulationsmodell einer visuellen Sprache ist nicht immer mit dem semantischen Modell der VL gleichzusetzen, es kann zusätzliche Konstrukte benötigen, jedoch auch kleiner sein, da es ja zunächst von der graphischen Repräsentation entkoppelt ist und eventuell bestimmte Teile, die nur der Repräsentation dienen, gar nicht benötigt. In der Literatur zu Simulationen wird dem Simulationsmodell ein großer Stellenwert beigemessen. Ein großes Problem, ist es, ein Simulationsmodell herzuleiten, wobei es die verschiedensten Ansäze (deklarativ, funktional, räumlich, multi-model...[3]) gibt. Allen Ansätzen ist jedoch gemein, dass Simulationsobjekte existieren, die sich im Simulationsmodell bewegen und einen Zustand des Systems ändern.

Dies lässt sich gut auf visuelle Sprachen übertragen. Hier sind die Simulationsobjekte Teile der visuellen Sprache, die modifiziert werden. Jedoch lassen sich nicht alle visuellen Sprachen gleichermaßen gut simulieren bzw. animieren. Dies wollen wir nun zeigen und eine Charakterisierung visueller Sprachen vorschlagen. Dies ermöglicht es, Anforderungen an die Simulation/Animation bzw. an die Spezifikation zu definieren.

Insgesamt haben wir vier unterschiedliche Tpyen von Sprachen identifiziert, die alle unterschiedliche Anforderungen an ein Simulations- bzw. Animationsframework stellen:

1. Typus: nicht simulierbare/animierbare Sprachen, da keine Ausführungssemantik vorhanden ist, bzw. diese extern liegt und nicht emuliert werden kann bzw. Sprachen wobei das Simulationsmodell und das Animationsmodell eine viel zu hohe semantische Distanz haben. Ein Beispiel wäre Java Quelltext und das zugehörige ausführbare Programm.

2. Typus: simulierbar mit inhärentem Zustand. Keine Animationsabbildung notwendig

3. Typus: simulierbar und animierbar mit einem unterschiedlichen domänenspezifischen Layout. Die Sprache wird auf ihrer semantischen Struktur simuliert, die Animation benötigt eine zusätzliche

graphische Abbildung. Die Abbildung ist jedoch nicht sehr weit von der semantischen Struktur entfernt. Die Abbildung erfordert i.d.R. nur eine Zuordnung von Objekten der semantischen Struktur auf die domänenspezifische animierte Struktur.

4. Typus: Simulationsmodell und Animationsmodell haben eine erhebliche semantische Distanz. Die Abbildung ist hier nicht duch eine einfache Zuordnung möglich. Dabei ist es schwieriger, wenn das Animationsmodell ein wesentlich höheres Niveau als das Simulationsmodell hat.

Der erste Typus beschreibt Sprachen, deren Simulation wenig Sinn machen würde, weil sie von externen Faktoren abhängt. Ein Beispiel wäre eine Sprache zur Spezifikation von Webanwendungen. Eine Simulation müsste einen Websurfer emulieren, was i.d.R. zu aufwändig ist.

Ein Beispiel für den zweiten Typus wären Petri-Netze. Die visuelle Sprache bringt direkt eine Ausführungssemantik mit, nämlich das Schalten von Transitionen. Die Simulation wäre hier also das dekrementieren der Tokenanzahl in der Vorbedingung einer Transition und das inkrementieren der Tokenanzahl in der Nachbedingung. Die Animation entspricht dem Markenflug und kann unmittelbar in der VL animiert werden. Die Darstellung des visuellen Programms wird ausgeführt. Dies ist gleichzeitig ein schönes Beispiel um den Unterschied von Animation und Simulation zu verstehen, wir werden dieses Beispiel später nochmals aufgreifen.

Ein Beispiel für den dritten Typus sind ebenfalls die Petri-Netze, jedoch in einer konkreten Ausprägung, zum Beispiel ein Producer-Consumer Netz. Da Petri-Netze ebenfalls dem Typus zwei angehören, können wir jede Instanz eines Netzes simulieren. Zusätzlich möchten wir nun eine konkrete Instanz auf eine domänenspezifische Animationssicht abbilden. Hierzu ist eine Zuordnung von Objekten der Sprache auf Objekte der domänenspezifischen Sicht nötig. Dies erreichen wir durch Pattern Matching. (Den Begriff Animationssicht führen Ermel et al. in [2] ein, siehe auch "verwandte Arbeiten").

Der vierte Typus beschreibt Sprachen deren Animationsmodell wesentlich komplexer ist, als das Simulationsmodell. Ein Beispiel wäre eine Sprache zur Spezifikation von regelbasierten Spielen. Die VL an sich hat nur einige Konstrukte um Vorher-/Nachher Regeln zu spezifizieren. Die Simulation bzw. Animation würde jedoch auf ein viel komplexeres Layout abgebildet, nämlich ein zweidimensionales kachelbasiertes Spielfeld. Hier ist bereits zu erkennen, dass das Simulationsmodell um spezielle Konstrukte erweitert werden muss, die lediglich der Darstellung und der Simulation dienen.

Um einen Simulator und eine Simulationssprache zu entwerfen haben wir uns zunächst gängige Simulationssprachen und -bibliotheken angeschaut. Zu den Simulationssprachen gehören unter anderem Siman [6] und GPSS [8]. Sie erlauben es, insbesondere Warteschlangensysteme für Fertigungsanlagen zu spezifizieren. Dies wird erreicht durch eine Reihe von Spezialkonstrukten zur Generierung von Simulationsobjekten, die dann in einem Modell von (Fertigungs-) Station zu (Fertigungs-) Station wandern und an jeder Station randomisiert verzögert werden. Gleichzeitig werden in sogenannten Tracevariablen Durchgangszeiten und Zufallszahlen mitprotokolliert.

Simulationsbibliotheken wie beispielsweise CSIM [9] erlauben es allgemeine Programmiersprachen (in diesem Fall C) um Klassenbibliotheken für Simulationskomponenten zu eweitern. Durch Betrachtung der vorgestellten Simulationssprachen und der mit ihnen zu realisierenden Simulationen ergeben sich direkt Anforderungen an die zu entwickelnde Simulationsspezifikationssprache.

Alle Simulationssprachen und -umgebungen haben gemein, dass Simulationsobjekte erstellt werden und durch das Simulationsmodell wandern, wobei sie manipuliert werden. Eine Simulationssprache sollte also das einfache Erstellen bzw. Gruppieren von Simulationsobjekten erlauben. Der Zugriff auf Simulationsobjekte einer bestimmten Art oder mit bestimmten Eigenschaften ist insbesondere bei visuellen Sprachen wichtig (z.B. "alle Stellen eines Petri Netzes, die Vorbedingung einer bestimmten Transition sind"). Die Simulationsobjekte kapseln dabei Eigenschaften, sind also im Sinne der objektorientierten Programmierung Instanzen von Klassen.

Alle Simulationssprachen definieren Simulationsabläufe über die Simulationszeit, die sehr flexibel gehandhabt werden kann. Nicht nur Komprimierung von Zeit, sondern auch das beliebige Zurückspringen und das Ausführen von Aktionen in der Zukunft sind wichtig. Hierbei ist eine ereignisbasierte Simulation mit einer prioritätenbasierten Warteschlange am flexibelsten. Um aussagekräftige Simulationen zu erreichen, ist ein Simulationsmodell, das Zufallsvariablen beliebiger Verteilung bietet von Bedeutung. Alle untersuchten Simulationssprachen bieten hierfür spezialisierte Konstrukte. Auch ist das Protokollieren von Variablen über die Zeit wichtig um Aussagen zum Simulationsmodell zu treffen. Viele Simulationsumgebungen sind auf die Simulation von Fertigungsstraßen spezialisiert. Hier gibt es viele Sprachkonstrukte um den Fluss von Werkstücken zu spezifizieren. Eine Simulationssprache sollte somit Warteschlangen und Konstrukte zum Transport von Simulationsobjekten bieten.

4. Der Simulator

Bevor die Simulationssprache besprochen wird, soll auf den generierten Simulator eingegangen werden.

Die Anforderungen an den Simulator sind vielfältig:

er soll auf einem maßgeschneiderten Simulationsmodell arbeiten, d.h. das Simulationsmodell sollte so minimal wie möglich sein, damit auch langlaufende Simulationen möglich sind.

Des Weiteren sollte der Simulator vom semantischen Modell (abstrakte Syntax definiert durch DSSL) der VL soweit wie möglich entkoppelt sein. Dies hat mehrere Gründe: Zunächst soll das bestehende Generatorframework nicht beeinflusst werden. Es dürfen an vorhandenen Spezifikationen keine Fehler durch Seiteneffekte entstehen. Auch soll der Simulator, falls keine Simulationsspezifikation vorliegt, quasi nicht vorhanden sein. Der Simulator soll allein stehend simulieren können. Das heißt, nachdem das Simulationsmodell instanziiert ist, kann der Simulator ohne die bestehende visuelle Sprache auskommen und seine Ergebnisse erst bei Bedarf zurückschreiben.

Die getrennte Spezifizierbarkeit von Simulator und visueller Sprache ist ein weiterer Anforderungsaspekt. Wenn in einem Team an einer visuellen Sprache gearbeitet wird, kann es sein, dass Teile, die für die Simulation wichtig sind, noch nicht in der visuellen Sprache vorhanden sind. Dies sollte kein Problem darstellen, da das Simulationsmodell vollständig entkoppelt ist von der VL. Insbesondere können Teile, die zunächst nur im Simulationsmodell existieren nachträglich in die eigentliche VL übernommen werden und umgekehrt.

Der Simulator sollte eine genormte Schnittstelle zur Außenwelt bieten um evtl. andere Simulationsumgebungen oder externe Bibliotheken einbinden zu können. Bei bestimmten Simulationen kann es vorkommen, dass es sehr viele Simulationsobjekte gibt, die auf wenige Objekte der visuellen Sprache abgebildet werden. Auch hier sollte der Simulator geeignete Konstrukte bieten.

Prinzipiell erlaubt es der Simulator und die zugehörige Sprache *DSIM* zunächst Sprachen vom Typ zwei zu simulieren, also Sprachen, deren Animationslayout sehr nah an der Simulation liegt. Um auch Sprachen vom Typ drei und vier animieren zu können ist es nötig, dass das generierte Simulationsmodell eine bestimmte Struktur besitzt. DEViL benutzt zur Erstellung der graphischen Darstellung visuelle Muster, die als attributierte Grammatiken vorliegen und auf einem Strukturbaum arbeiten, den LIGA, der Attributauswertergenerator des Eli Systems, verarbeiten kann. Damit wir die visuellen Muster wiederverwenden können, um ein Animationslayout zu berechnen, muss das Simulationsmodell des Simulators ebenfalls die genormte Schnittstelle zu LIGA anbieten und entsprechende Baumaufbaufunktionen bereitstellen.

Insgesamt ergibt sich für den Simulator eine Struktur wie im Blockdiagramm aus Abb. 4 dargestellt. Der Simulator simuliert auf dem (ggf. erweiterten Simulationsmodell, Details dazu im nächsten Abschnitt). Er schreibt zunächst die Änderungen direkt in das eigene Simulationsmodell und

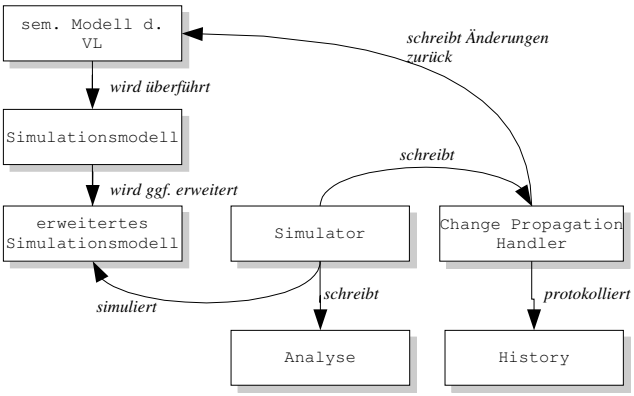

Figure 4. Konzept des Simulators

registriert diese Änderungen zusätzlich im Change Propagation Handler (CPH). Dieser schreibt die Änderungen auf Wunsch in das semantische Modell der visuellen Sprache zurück. Dies kann nach jeder Simulationsrunde passieren oder bei Langläufern beliebig spät. Der Change Propagation Handler selbst optimiert die Rückschreibeoperationen derart, dass Attribute von Objekten die irgendwann gelöscht werden nicht modifiziert werden. Des Weiteren schreibt der CPH in eine History, die es erlaubt die Simulation schrittweise rückgängig zu machen.

Ein weiterer Aspekt des Simulators ist, dass er ereignisbasiert arbeitet und dabei den "next-event-next Ansatz" verfolgt, bei dem in der Ereigniswarteschlange die Zeit bis zum nächsten Ereignis fortgeschaltet wird.

5. Die Sprache DSIM

Beim Entwurf einer Simulation auf Basis einer visuellen Sprache ist schnell zu erkennen, dass die semantische Struktur der visuellen Sprache nicht optimal zum Simulationsmodell passt, weil sie Klassen, Attribute oder Referenzen enthält, die als Zugeständnis an die Repräsentationsstruktur, also der visuellen Darstellung der visuellen Sprache hinzugefügt wurden. Es werden jedoch bestimmte Objekte der visuellen Sprache im Simulationsmodell benötigt.

Das Simulationsmodell einer visuellen Sprache benötigt also zunächst einfache Mechanismen um auf Objekte der visuellen Sprache zuzugreifen. Die Objekte der visuellen Sprache werden dann in Simulationsobjekte umgewandelt und können dann in der Simulation benutzt werden, was einer Manipulation der Simulationsobjekte entspricht. Simulationsobjekte haben gegenüber Objekten der visuellen Sprache einige andere Eigenschaften. Bei der Initialisierung sind sie genaue Kopien ihrer Pendants der visuellen Sprache. Beim Durchlaufen des Simulationsmodells werden die Modifikationen jedoch nicht unmittelbar auf

die Instanz des visuellen Sprachmodells übertragen. Dies passiert erst auf Wunsch des Simulationsspezifizierers oder spätestens nach jedem Durchlauf eines konkreten Simulationszeitpunktes (zu jedem Simulationszeitpunkt können beliebig viele Modifikationen durchgeführt werden). Dies erlaubt eine effizientere Simulation und auch langlaufende Simulationen sind so möglich. Der erwähnte Zugriff auf Objekte der visuellen Sprache erfolgt durch die bereits eingeführten Pfadausdrücke. Diese erlauben es, einfach durch den Strukturbaum zu wandern und Objekte, die bestimmte Eigenschaften erfüllen aufzusammeln (z.B. die Vorbedingung einer Transition eines Petri-Netzes).

Des Weiteren erlaubt die Simulationssprache durch einfache Sprachkonstrukte das Simulationsmodell zu modifizieren (dazu später mehr). Das Simulationsmodell für die visuelle Sprache ist ein 3-Tupel $\mathcal{S} = (\mathcal{C}_{\text{sim}}, \mathcal{A}, \mathcal{M})$, wobei $\mathcal{C}_{\text{sim}} = \mathcal{C}_{\text{semantic}} \cup \mathcal{C}_{\text{virtual}}$ die Menge der Klassen des visuellen semantischen Sprachmodells vereinigt mit der Menge der virtuellen (abstrakten) Simulationsklassen ist.

$\mathcal{A} = \mathcal{P} \cup \mathcal{V} \cup \mathcal{Q}$ ist die Menge der Attribute, bestehend aus der Menge der Pfadausdrücke, der Menge der Variablen und der Menge der Warteschlangen.

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

Das Zeitmodell kann durch den Benutzer der Simulation später beliebig fortgeschaltet werden. So sind mögliche Benutzerinteraktionen: “simuliere n Schritte” oder “simuliere x Sekunden, Minuten, Stunden...”.

Hier ist notwendig auf bestimmte “Unterbrechungen” korrekt reagieren zu können. Diese Unterbrechungen können Benutzereingaben, z.B. durch Tastatur oder Maus sein, die beispielsweise einen Agenten in eine Spielesimulation steuern oder es können bestimmte Simulationsmuster sein, die auf einen bestimmten Endzustand der Simulation verweisen. Hier könnte es sich beispielsweise um das Simulationsende (Spielende) handeln oder um einen Zyklus bzw. es könnte auch ein bestimmter Wertebereich als kritisch erachtet werden.

Die Anforderungen an die Simulationssprache sind vielfältig: sie soll möglichst einfach sein, jedoch eine große Menge an Simulationen visueller Sprachen durch einfache Sprachkonstrukte abdecken.

6. DSIM im Detail

Die Sprache DSIM besteht aus drei Teilen: der Definition des Simulationsmodells, einem Bedingungsblock, in dem Ereignisse ausgelöst werden können und einem Ereignisdefinitionsblock.

Das Simulationsmodell ermöglicht es, Klassen zu definieren und diesen Klassen Attribute zuzuordnen. Es können (müssen aber nicht) alle Klassen des semantischen Modells der VL übernommen werden. Wird eine Klasse übernommen, so ist sie ein Abbild ihres Pendants im sem. Modell der VL, hat also diesselben Attribute und Attributbelegungen. Des Weiteren können der Klasse zusätzliche Attribute in Form von Pfadausdrucksmengen, Warteschlangen, Zufallsvariablen oder zusätzliche primitiven Attributen zugeordnet werden.

Zusätzlich können im Simulationsmodell sogenannte virtuelle Simulationsklassen definiert werden. Diese Klassen haben kein Pendant im semantischen Modell der VL, haben jedoch den Zweck, dass sie der Animationsabbildung als Basis dienen können oder während der Entwicklung einer VL als Ersatz von Klassen des sem. Modells die noch nicht implementiert wurden. Diesen virtuellen Klassen können ebenso alle Arten von Attributen zugeordnet werden.

Im Bedingungsblock kann auf das Simulationsmodell zugegriffen und es modifiziert werden. Der Bedingungsblock wird vom Simulator in einer Endlosschleife durchlaufen. Hier können Ereignisse in die Ereigniswarteschlange eingefügt werden, die im Ereignisdefinitionsblock deklariert wurden. Im Bedingungsblock können spezielle Ausdrücke benutzt werden, die es erleichtern auf Mengen von Objekten zu arbeiten, z.B. FOREACH, IFSOME, IFEVERY.... Ereignisse können beliebig weit in der Zukunft eingeordnet werden, i.d.R. parametrisiert man diese jedoch durch Zufallsvariablen.

Abbildung 5 zeigt die einzelnen Teile einer Simulationsspezifikation für Petri-Netze.

Anzumerken bleibt, dass DSIM für die reine Simulation sorgt, d.h. nur abstrakt die Animation definiert, der Ausdruck ”MOVE token x,y” in der Simulation besagt nur, dass ein dynamisches Objekt existiert, dass ”token” heißt und von x nach y bewegt werden soll. Wie genau das Token aussehen soll und wo die konkreten Positionen x und y, so wie die Bewegungsfunktion wird in der Deklaration der Animationssicht spezifiziert:

```
ANIMATIONVIEW petriDSL {
 MOVE token = "tokenDrawing" (x,y),
slowInSlowOut, 2.5 sec;
}
```

Dies ist vergleichbar mit dynamischer Methodenbindung in einer objektorientierten Sprache. Hier wird auch zunächst in einer Oberklasse abstrakt das Konzept definiert und in den Unterklassen, die in unserem Fall den Animationssichten entsprechen, für eine konkrete Ausprägung gesorgt.

Zur Zeit müssen wir noch herausfinden, welche weiteren Komponenten in einer Animationssichtdeklaration definiert sein müssen. Hier sind insbesondere Initialisierungen des erweiterten Simulationsmodells sowie eine sinnvolle Startbelegung für Positionen statischer Objekte der Animation-

```
SimulationModel{
 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]";
 }
}
```

Simulationsmodell

```
Conditions {
  FOREACH t IN Transition RANDOM{
   IFEVERY t.incomingPlaces->marks > 0 {
    FIRE preTransitionFire(t.incomingPlaces, t);
    FOREACH p IN t.outgoingPlaces {
      FIRE postTransitionFire(t, p);
    }
   }
  }
}
```

Bedingungsblock

```
Events{
 preTransitionFire(
  SET incomingPlaces OF Place, Transition t) {
   incomingPlaces->marks = incomingPlaces->marks-1;
   FOREACH place IN incomingPlaces {
     MOVE "tokenDrawing"
      (place->position, t->position);
   }
  }

 postTransitionFire(Transition t, Place p) {
  p->marks = p->marks +1;
  MOVE "token" (t->position, p->position);
 }
}
```

Ereignisblock

Figure 5. Simulationsspezifikation für Petri-Netze

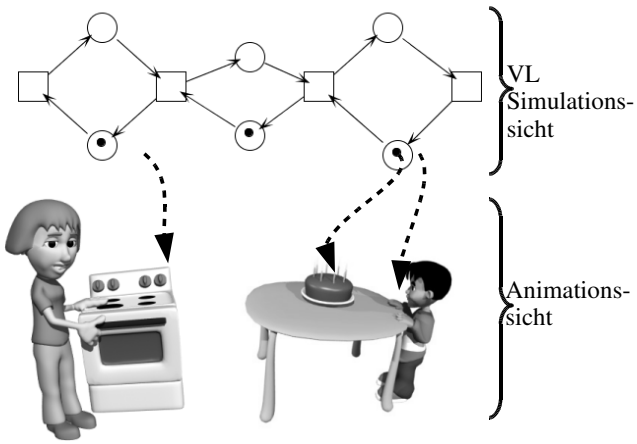

Figure 6. Abbildung auf das DSL Layout

ssicht zu nennen.

7. Das Animationsframework

Bis zu diesem Zeitpunkt ist nur die Simulation visueller Sprachen vom Typ zwei mit DSIM möglich, d.h. die Animation findet auf fast demselben Layout wie die Simulation statt. Um auch Sprachen animieren zu können, deren Animation sich von der Simulation stärker unterscheidet, ist es nötig eine Abbildung des Simulationsmodells auf eine Animation zu definieren.

Dazu betrachten wir nun zunächst noch einmal die Sprache der Petri-Netze und das domänenspezifische Layout “Producer/Consumer-Netz”. Eine denkbare Visualisierung wäre, das eine konkrete Petri-Netz Ausprägung, also ein Satz der Sprache auf die Konstrukte “Mutter backt für ihr Kind einen Kuchen” abgebildet werden. Teile des Petri-Netzes wie Stellen werden also auf statische Objekte der Animationssicht abgebildet wie z.B. ein Backofen oder ein Tisch. Das Token würde auf ein dynamisches Objekt der Animationssicht, z.B. ein Kuchen abgebildet. Diese Strukturen müssen im Petri-Netz erkannt werden und danach muss das domänespezifische Layout aufgebaut werden (vgl. 6) .

Der Aufbau des domänenspezifischen Layouts kann durchaus komplizierter sein. Betrachtet man die visuelle Sprache zur Spezifikation von regelbasierten Spielen, so fällt auf, dass das Spielfeld in der visuellen Sprache nicht existiert, es muss in der domänenspezifischen Sicht zunächst instanziert und mit sinnvollen Werten vorbelegt werden.

Insgesamt ergibt sich für das Animationsframework die Struktur die in Abbildung 7 zu sehen ist.

Das Simulationsmodell basiert zunächst auf (Teilen der) semantischen Struktur der ursprünglichen visuellen Sprache. Dieses Modell kann nun erweitert werden durch

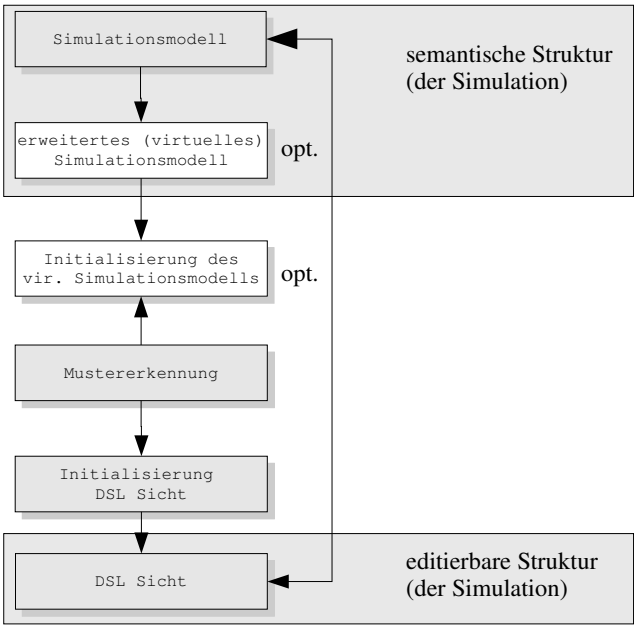

Figure 7. Animationskonzept

virtuelle Simulationsklassen. Diese können im nächsten Schritt initialisiert werden. Anschließend wird im nächsten Schritt auf dem Simulationsmodell eine Mustererkennung durchgeführt, die Konstrukte der domänenspezifischen Animationssicht im Simulationsmodell erkennt und letztendlich die DSL Sicht aufbaut. Diese Sicht muss beim ersten Aufbau noch initialisiert werden, so müssen zum Beispiel statischen Objekten (wie dem Tisch oder dem Backofen im Petri-Netz Beispiel) sinnvolle Anfangspositionen zugeordnet werden.

Das Aussehen der DSL Sicht wird vom Sprachspezifizierers durch die bekannten visuellen Muster erreicht. Wie bereits in Kapitel drei erwähnt, kann dieses Konzept benutzt werden, da das Simulationsmodell dieselbe Baumstruktur aufbaut, die auch vom Attributauswertergenerator LIGA des Eli Systems akzeptiert werden. Des Weiteren ist es möglich beliebig viele domänenspezifische Ansichten auf demselben Simulationsmodell zu spezifizieren.

In Abbildung 7 ist zu sehen, dass es eine Verbindung zwischen DSL Sicht und Simulationsmodell gibt. Diese ist nötig um Benutzerinteraktion weiterzuleiten, die auf der DSL Sicht ausgeführt wird. Bei der Spielesprache kann dies die Interaktion mit einer Spielfigur auf dem Spielfeld sein. Hier ist noch weiter zu untersuchen, wie eine Interaktion spezifiziert werden kann und wie sie auf das Modell wieder übertragen werden kann.

Des Weiteren ist im Animationsframework noch zu untersuchen, wie das Pattern Matching durchgeführt werden kann. Hier bieten sich ebenfalls Attributauswerter an, eine Systematik hierfür haben wir uns ebenfalls bereits überlegt.

8. Verwandte Arbeiten

Im Bereich der Simulation haben wir uns zunächst an der Funktionalität bekannter Simumationsumgebungen wie CSIM, Simscript, GPSS und ähnlichem orientiert. Prinzipiell möchten wir nämlich möglichst viele und umfangreiche Simulationen unterstützen.

Im Bereich der Animation visueller Sprachen ist besonders das GenGed System [1] und die Erweiterung von Ermel et al. [2] zu nennen. GenGed basiert auf dem Konzept der Graph Grammatiken. Die Simulation wird ebenfalls in Form einer Transformation von Graphen beschrieben. Dies führt zu einer Art regelbasierten Vorher-/Nachhe-Zustandsspezifikation. Ermel et. al führen ebenfalls eine Abbildung auf Animationssichten durch. Der Ansatz ist weitaus formaler, da die Abbildungen insbesondere vollständig und korrekt spezifiziert werden sollen und auf dem semantischen Modell der Sprache basieren. Die dazu nötigen Abbildungsregeln können automatisch generiert werden. Ermel et al. können jedoch keine Simulationen mit Warteschlangen oder Zufallszahlen generieren.

Häufig werden in der Literatur zur Simulation auch "Abstract State Machines" eingesetzt. Dies ist eine Art regelbasierte Sprache, die sehr allgemein gehalten ist. DSIM ist hier spezialisierter und bietet insbesondere viele Konstrukte um auf Mengen von (Simulations-) Objekten zu arbeiten.

9. Zukünftige Arbeit

Die Spezifikation der Simulationssprache DSIM ist soweit abeschlossen. In den bereits implementierten Beispielen Petri-Netze, Statecharts und einer Waschstraßensimulation hat sie sich bewährt. Noch zu untersuchen ist hier, wie sie sich bei weitaus komplexeren Simulationen verhält. Hier ist evtl. eine Modularisierung nötig.

Auf der Seite des Animationsframeworks ist insbesondere zu untersuchen, wann ein Pattern Matching nötig ist und wie es möglichst einfach implementiert werden kann. Auch muss noch die Interaktion zwischen Animation und Simulation genauer betrachtet werden.

References

[1] R. Bardohl. GenGed: A generic graphical editor for visual languages based on algebraic graph grammars. In *1998 IEEE Symp. on Visual Lang.*, pages 48–55, Sept. 1998.

[2] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, Universität Berlin, 2006.

[3] P. A. Fishwick. *Simulation Model Design and Execution*. Prentice Hall, 1995.

[4] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, Feb. 1992.

[5] U. Kastens. An attribute grammar system in a compiler construction environment. In *Proceedings of the International Summer School on Attribute Grammars, Application and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 380–400. Springer Verlag, 1991.

[6] C. D. Pegden, R. E. Shannon, and R. P. Sadowski. *Introduction to Simulation Using SIMAN*. McGraw Hill, 1995.

[7] C. Schmidt, B. Cramer, and U. Kastens. Usability evaluation of a system for implementation of visual languages. In *Symposium on Visual Languages and Human-Centric Computing*, pages 231–238, Coeur d'Alène, Idaho, USA, Sept. 2007. IEEE Computer Society Press.

[8] T. J. Schriber. *An introduction to simulation using GPSS/H*. John Wiley & Sons, Inc., New York, NY, USA, 1991.

[9] H. D. Schwetman. Introduction to process-oriented simulation and csim. In *Proceedings of the 1990 Winter Simulation Conference*, 1990.

[10] XML Path Language (XPath) Version 1.0, W3C Recommendation, Nov. 1999. `http://www.w3c.org/TR/xpath`.