Generierung von Editoren für interaktive animierte Sprachen

Ein modellgetriebener Ansatz unter Einsatz der Animation Modeling Language (AML)

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Torsten Strobl
im Juni 2014

Vorsitzender der Kommission:	Prof. Dr. Stefan Pickl
1. Berichterstatter:	Prof. Dr.-Ing. Mark Minas
2. Berichterstatter:	Prof. Dr. Gunnar Teege
1. Prüfer:	Prof. Klaus Buchenrieder, Ph.D.
2. Prüfer:	Prof. Dr.-Ing. Wolfgang Reinhardt

Tag der mündlichen Prüfung: 26. Mai 2014

Universität der Bundeswehr München
Fakultät für Informatik

Kurzfassung

Grafische Editoren sind unverzichtbare Werkzeuge für die Nutzung visueller Sprachen. Aufgrund der zunehmenden Verbreitung visueller Sprachen, v. a. in domänenspezifischen Bereichen, wurden daher zahlreiche Editor-Generatoren und -Frameworks entwickelt. Bisher existieren allerdings kaum Hilfsmittel zur Erzeugung von Editoren für animierte Sprachen, d. h. für visuelle Sprachen mit animierter Visualisierung. Derartige Sprachen können bspw. die Simulation oder Ausführung von Modellen darstellen. In manchen Fällen kann die animierte Visualisierung sogar interaktiv gesteuert werden.

Um den Entwicklungsprozess von Editoren für interaktive animierte Sprachen zu unterstützen, wird ein bewährter Generierungsansatz für Diagrammeditoren erweitert. Dieser Ansatz sieht grundlegend vor, dass Diagramme formal durch Hypergraphen repräsentiert werden. Dadurch können entsprechende Sprachen und Editoren auf Basis von Hypergraphen und Graphtransformationen spezifiziert werden. Im erweiterten Ansatz werden auch Sprachanimationen und dynamische Anteile auf dieser Basis spezifiziert. Diese Methode wird mit einer modellgetriebenen Herangehensweise kombiniert, d. h., interaktive animierte Sprachen können auf unterschiedlichen Abstraktionsebenen modelliert werden. Eingesetzt wird dabei die Animation Modeling Language (AML) und eine auf das Anwendungsgebiet zugeschnittene Erweiterung, in der Hypergraphspezifikationen innerhalb von AML-Modellen eingebettet werden können. Ausgehend von einem solchen AML-Modell können dann Editoren für interaktive animierte Sprachen generiert werden.

Abstract

Graphical editors are indispensable tools for using visual languages. Due to the increasing spread of visual languages, especially in domain-specific areas, numerous editor generators and frameworks have been developed. However, there are very few tools which facilitate the implementation of editors for animated languages, i. e., for visual languages with animated visualization. Languages of this type can show the simulation or execution of models, for instance. In some cases, the animated visualization can even be controlled interactively.

In order to support the development process of editors for interactive animated languages, an established generation approach for diagram editors is extended. The essential idea of this approach is to represent diagrams by hypergraphs formally. As a consequence, corresponding languages and editors can be specified by means of hypergraphs and graph transformations. In the extended approach also language animations and dynamic parts are specified by these means. This method is combined with a model-driven strategy, i. e., interactive animated languages can be modeled at different levels of abstraction. For this purpose, the Animation Modeling Language (AML) and an extension of this language are applied. The extension is tailored to the field of application by enabling hypergraph specifications inside of AML models. Based on such an AML model, editors for interactive animated languages can be generated.

Danksagung

Die Erstellung dieser Dissertation wäre ohne die Unterstützung von so vielen Seiten nicht möglich gewesen. An dieser Stelle möchte ich mich daher bei allen, die mich unterstützt haben, herzlichst bedanken.

Ein besonderer Dank gilt meinem Doktorvater Prof. Dr. Mark Minas. Sein Vertrauen in mich, seine fachkundige Betreuung sowie seine konstruktive Kritik haben entscheidend zum erfolgreichen Abschluss meiner Arbeit beigetragen. Ebenso möchte ich mich bei Prof. Dr. Gunnar Teege für die Erstellung des Zweitgutachtens bedanken sowie allen anderen Mitgliedern der Prüfungskommission.

Auch die langjährige Unterstützung der Ostbayerischen Technischen Hochschule Regensburg und insbesondere von Prof. Dr. Athanassios Tsakpinis waren außergewöhnlich und haben mich in meinem Vorhaben stets bestärkt.

Unterstützt haben mich ebenfalls meine Kollegen am Institut Dr. Sonja Maier, Dr. Steffen Mazanek und Dr. Florian Brieler. Sie haben mir in allen Phasen meiner Arbeit immer wieder mit hilfreichen Ratschlägen zur Seite gestanden.

Ich möchte außerdem Dr. Andreas Pleuß und Dr. Arnd Vitzthum für ihre Ideen und die inspirierenden Diskussionen danken. Ohne ihre wertvolle Vorarbeit und Anregungen wäre die Modellierungssprache AML in der präsentierten Form nicht entstanden.

Desweiteren möchte ich meine Dankbarkeit gegenüber der Infineon Technologies AG zum Ausdruck bringen, die mir eine nebenberufliche Promotion ermöglicht hat. Meinen Arbeitskollegen dort will ich für die angenehme und freundschaftliche Atmosphäre danken.

Schließlich danke ich auch meiner Familie und meinen Freunden, die mir in besonders arbeitsintensiven Phasen stets Rückhalt gegeben haben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Seit vielen Jahren sind Applikationen mit grafischen Benutzeroberflächen, die dem Benutzer die Interaktion mittels Steuerelementen und Maus erlauben, nicht mehr wegzudenken und ein wesentlicher Bestandteil moderner Benutzerführung. Mit dem Siegeszug von entsprechenden Betriebssystemen verbesserten sich Bedienkomfort und Verständlichkeit, so dass der Computer längst im Alltag angekommen ist. Mensch-Maschine-Schnittstellen werden dabei fortlaufend weiterentwickelt und immer weitere Technologien wie bspw. Gestensteuerung bei Spielekonsolen oder haptisches Feedback bei Smartphones kommen zum Einsatz. Auch grafische Effekte und Animationen haben einen hohen Stellenwert erreicht. Dabei sind Animationen nicht nur dekorativer Zusatz, der Hard- und Software inzwischen zu Lifestyle-Produkten macht, sondern auch Hilfsmittel, um wichtige Vorgänge hervorzuheben oder verständlicher darstellen zu können.

In Analogie dazu wird auch in der Softwareentwicklung nach Möglichkeiten gesucht, Programme zu visualisieren, damit diese verständlicher dargestellt werden können. Auch der Entwicklungsprozess soll dadurch vereinfacht werden. Bereits sehr früh wurden daher Sprachen für Flussdiagramme [DIN66], Struktogramme bzw. Nassi-Shneiderman-Diagramme [NS73] oder grafisch notierte Zustandsautomaten (z. B. Harel-Statecharts [Har87]) konzipiert und als Hilfsmittel in der Entwicklung eingesetzt. Die inzwischen auf breiter Ebene akzeptierte Unified Modeling Language (UML) [UML11] zählt ebenfalls zu diesen Sprachen. Zahlreiche Arbeiten und konkrete Projekte kamen zu dem Ergebnis, dass visuelle Sprachen in vielfältiger Weise nützlich für die Softwareentwicklung sein können, auch wenn einige Studien manche Vorteile immer wieder in Frage stellen (vgl. [Sch98]).

Dank immer leistungsstärkerer Computer werden verstärkt auch Animationen eingesetzt, um grafische Darstellungen ansprechender zu gestalten. Ein wichtiger Leitgedanke ist dabei auch, die Vorgänge und Informationen für Benutzer noch verständlicher zu präsentieren. Dies ist naheliegend, da sich Animationen in natürlicher Weise zur Darstellung kontinuierlicher Änderungen eignen. Zahlreiche Arbeiten beschäftigen sich daher mit dem Nutzen von Animationen für die Verbesserung der Anschaulichkeit, v. a. in der Lehre (z. B. [TMB02]). Auch in

Bezug auf visuelle Sprachen spielen Animationen eine zunehmend wichtigere Rolle. Dies trifft v. a. im Bereich der Algorithmenanimation und Datenvisualisierung zu oder bei Sprachen, die eine Ausführungssemantik besitzen oder für Programmieranfänger und insbesondere Kinder geeignet sein sollen.

Diese Entwicklung lässt den Schluss zu, dass Editoren für animierte Sprachen – oder schlicht animierte Werkzeuge – in bestimmten Bereichen der Forschung, Entwicklung und Lehre an Bedeutung gewinnen können. Die Umsetzung von derartigen Editoren ist allerdings aufwändig, wenn diese traditionell programmiert werden müssen. Editoren-Frameworks helfen dabei nur bedingt weiter und unterstützen Animationen in den seltensten Fällen direkt. Arbeiten wie [NRA+02] nennen den „lack of effective development tools“ als einen der Hauptgründe, warum dynamische Visualisierungen nur zögerlich eingesetzt werden oder nicht eingesetzt werden können. Auch [RV06] identifiziert die Schwierigkeit der Umsetzung von Werkzeugen als Hindernis für den Einsatz von „Design-time interactive model simulation“.

Aus diesem Grund beschäftigt sich diese Arbeit mit der Generierung von Editoren für interaktive animierte Sprachen. Ein modellgetriebener Ansatz soll dabei helfen, entsprechende Editoren effektiv umsetzen zu können.

1.1 Generierung von Editoren für interaktive animierte Sprachen

Der Begriff *„interaktive animierte Sprache“* ist durch einschlägige Literatur bisher noch nicht maßgeblich geprägt worden. Selbst einzelne Aspekte des Begriffs werden von Autoren unterschiedlich ausgelegt, d. h. auch *„visuelle Sprachen“*, die im Rahmen dieser Arbeit die Obermenge animierter Sprachen bilden, werden unterschiedlich definiert. Es folgt daher eine kurze Interpretation des Begriffs, der in dieser Arbeit Anwendung finden soll.

Als visuelle Sprachen sollen Sprachen gelten, die sich von textuellen Sprachen durch eine räumliche Anordnung der Sprachelemente abgrenzen. Zusätzlich müssen grafische Sprachelemente eingesetzt werden, d. h. Elemente, die sich von einfachen Zeichen, Ziffern, Symbolen oder darauf basierenden Zeichenfolgen unterscheiden. Für Wörter einer visuellen Sprache werden im Folgenden auch Begriffe wie *„Diagramm“* oder allgemeiner *„(grafisches) Modell“* verwendet. Dabei wird der Begriff „Diagramm“ vorrangig gebraucht, da in allen Beispielen zweidimensionale, diagrammorientierte Sprachen (auch Diagrammsprachen) betrachtet werden.

Im Falle von animierten Sprachen werden außerdem *Animationen* bei der Darstellung der Sprachelemente eingesetzt, wobei Animation im Sinne von sich schnell ändernden Bildfolgen zu verstehen ist. Animation kann dabei vielfältig eingesetzt werden. Sie kann genutzt werden, um Sprachen optisch ansprechender zu gestalten, aber auch immanenter Teil der Sprache und deren Verwendung sein. Oftmals ist mit dem Einsatz animierter Sprachen auch die Simulation eines Diagramms (bzw. Modells) verbunden. Der Begriff *„interaktiv“* wird in

diesem Zusammenhang verwendet, wenn ein animiertes Diagramm bzw. Modell zu jedem Zeitpunkt durch Ereignisse beeinflusst werden kann. Ereignisse können bspw. eintreten, wenn Sprachelemente bestimmte Nachrichten untereinander austauschen. Es ist ebenfalls möglich, dass bestimmte Ereignisse „von außen“ ausgelöst werden.

Zur Erstellung von animierten Diagrammen sollen *Editoren* genutzt werden, die möglichst alle Aspekte der Sprache unterstützen. Je nach Sprache kann dies z. B. bedeuten, dass die Simulation eines Modells animiert dargestellt werden muss oder der Editor die visuelle Programmierung in einer animierten Umgebung ermöglichen muss. Zusätzlich unterstützen viele Editoren typische Funktionalität im Umfeld visueller Sprachen wie z. B. Syntaxprüfung, Diagrammkorrektur, automatisches Layout usw.

Die Implementierung solcher Editoren – oder auch Tools – ist allerdings aufwändig, weshalb sogenannte *Meta-Tools* existieren. Diese ermöglichen die Umsetzung von Editoren auf der Grundlage einer *Sprach- und Editorspezifikation*. Unterschieden wird dabei meist auch die Spezifikation der konkreten und der abstrakten Sprachsyntax. Während durch die konkrete Syntax die genaue Darstellungsform und Notation der Sprache festgelegt wird, definiert die abstrakte Syntax lediglich die innere Struktur und erlaubte Wörter der Sprache. Die Editorspezifikation legt zusätzliche Details fest, die nicht Teil der Sprachspezifikation sind, aber für die Funktionalität eines Editors benötigt werden: Steuermöglichkeiten durch Mauseingabe, Editorlayout, Eingabehilfen etc. Welche Möglichkeiten in einer Editorspezifikation bereitgestellt werden, ist dabei abhängig vom jeweiligen Meta-Tool bzw. Editor-Framework. Ausgehend von derartigen Spezifikationen können Meta-Tools dann den Programmcode des Editors erzeugen, weshalb auch der Begriff *„Generierung“* verwendet wird.[1]

Für die Spezifikation von (visuellen) Sprachen gibt es zahlreiche Methoden. Etabliert hat sich dabei der Einsatz von Graphen, d. h., es werden sehr oft attributierte Graphen als interne Datenstruktur für Diagramme verwendet, was in den Sprachspezifikationen entsprechend festgelegt wird. Dies ermöglicht gleichzeitig den Einsatz formaler Methoden, z. B. für eine Diagrammvalidierung (bzw. Syntaxprüfung) auf der Grundlage von Graphen. Mögliche Diagrammänderungen können außerdem durch *Graphtransformationsregeln (GTRs)* festgelegt werden. Da bestimmte Animationen ebenfalls mit Diagrammänderungen verbunden sind, eignen sich GTRs auch zur Spezifikation animierter Sprachaspekte.

Obwohl animierte Sprachen mittels GTRs und animierter Visualisierung spezifiziert werden können (vgl. [SM09, SM10]), zeigt die Vorgehensweise bei komplexeren animierten Sprachen, in denen viele GTRs benötigt werden, einige Schwächen (vgl. [SMPV10]). So stellte es sich als problematisch heraus, die zahlreichen GTRs zu planen und miteinander zu verknüpfen. Außerdem mangelt es an Übersichtlichkeit und Dokumentationsmöglichkeiten, wodurch es schwierig ist, Fehler zu finden und zu beheben. Insgesamt leidet darunter auch Konsistenz und Erweiterbarkeit einer Sprachspezifikation. Aus diesem Grund wurde ein *mo-*

[1] Dieser Begriff soll aber auch für Ansätze verwendet werden, bei denen kein Code entsteht und Sprach- und Editorspezifikation direkt interpretiert werden können.

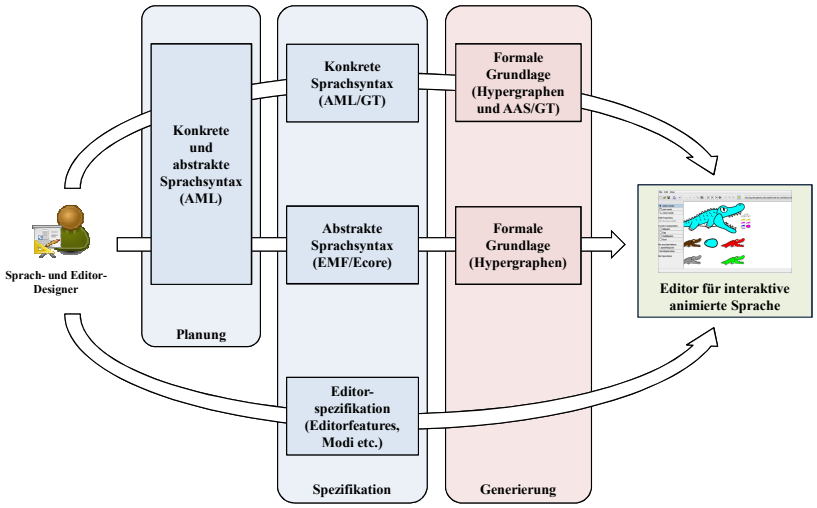


Abbildung 1.1: Generierung von Editoren aus Sprach- und Editorspezifikation

dellgetriebener Ansatz entwickelt. Verwendet wird dabei die *Animation Modeling Language (AML)*. Mit ihr kann die konkrete Sprachsyntax modelliert werden (auch visuell), wobei sowohl die visuelle, animierte Repräsentation als auch das Verhalten der Sprachelemente spezifiziert werden.[2] Außerdem ist ihr Einsatz auf unterschiedlichen Abstraktionsebenen (z. B. zur Grob- und Feinspezifikation) und mit unterschiedlichem Detailgrad möglich.

Eine auf das Anwendungsgebiet zugeschnittene Erweiterung von AML namens *AML for Graph Transformations (AML/GT)* ermöglicht die Verwendung graphspezifischer Konstrukte. AML/GT wurde entwickelt, um zuvor spezifizierte AML-Modelle[3] zu ergänzen und Spezifikationen für animierte Sprachen auf der Grundlage von (Hyper-)Graphen erstellen zu können. Von derartigen AML/GT-Modellen können Sprachspezifikationen für geeignete Meta-Tools abgeleitet werden, die dann in der Lage sind, entsprechende Editoren zu generieren.

1.2 Zielsetzung und Beiträge der Arbeit

Abb. 1.1 zeigt die Idealvorstellung und gleichzeitig Zielsetzung des durch diese Arbeit propagierten Konzepts. Bei der Entwicklung einer interaktiven animierten Sprache sollte diese zunächst auf höherer Abstraktionsebene geplant werden. Hierfür können in erster Linie AML-Modelle zum Einsatz kommen. Auf dieser

[2] Dies ist nicht zu verwechseln mit dem „echten“ Verhalten der Systeme, die ggf. durch Modelle der animierten Sprache festgelegt werden. Trotzdem ist es möglich, dass Sprache und modelliertes System ähnliches Verhalten aufweisen, z. B. falls die Sprache eine animierte Simulation des Modells unterstützt.

[3] Da damit das Modell einer (Modellierungs-)Sprache festgelegt wird, werden solche Modelle häufig auch „Metamodell“ genannt. Auf diesen Begriff wird im Folgenden aber meist verzichtet.

Grundlage kann ein AML/GT-Modell entwickelt werden, mit dem die konkrete Sprachsyntax festgelegt wird. Darin müssen neben der Struktur v. a. das Aussehen und das Verhalten (inkl. Animation) der Sprachelemente modelliert werden. Zusätzlich muss die abstrakte Sprachsyntax bestimmt werden. Diese wird für viele Meta-Tools in Form eines (Meta-)Modells spezifiziert. Für das Meta-Tool DiaMeta, das zur Umsetzung und Validierung der Ergebnisse dieser Arbeit verwendet wurde, kann hierfür das Eclipse Modeling Framework (EMF) (siehe [EMF12]) bzw. ein Ecore-Modell eingesetzt werden. Viele umgesetzte animierte Sprachen haben gezeigt, dass sich dieses Modell strukturell meistens nur in wenigen Punkten von dem Modell für die konkrete Syntax unterscheidet. Aus diesem Grund kann das während der Planung entstandene AML-Modell in der Regel auch als Ausgangspunkt für die Spezifikation der abstrakten Syntax dienen. Abschließend wird eine Editorspezifikation erstellt, die weitere Details in Bezug auf einen Editor enthält, d. h. Merkmale des Editors, die nur indirekt im Zusammenhang mit der Sprache selbst stehen.

Die erwähnte Editorspezifikation wird direkt für ein bestimmtes Meta-Tool erstellt, worauf nicht weiter eingegangen werden soll. Im Gegensatz dazu erfolgt die Spezifikation der konkreten Sprachsyntax durch AML/GT unabhängig vom Meta-Tool, was bei bestimmten Meta-Tools auch für die Spezifikation der abstrakten Sprachsyntax zutrifft. Um einen funktionsfähigen Editor generieren zu können, müssen diese Spezifikationen jedoch in eine Meta-Tool-spezifische Form transformiert werden (idealerweise vollautomatisch).

Der Fokus dieser Arbeit liegt auf der Spezifikation der konkreten (animierten) Sprachsyntax mittels AML/GT und deren Übersetzung in einen Formalismus namens AAS/GT, der die Verwendung von GTRs zur Umsetzung von Animationsmechanismen einschließt. Durch ein Meta-Tool, welches die Spezfikation einer konkreten Sprachsyntax auf der Grundlage von Konzepten des AAS/GT-Formalismus unterstützt, können dann animierte Editoren generiert werden. Die spezifische Verwendung von (Hyper-)Graphen, der Formalismus AAS/GT sowie die Modellierungssprachen AML und AML/GT werden daher im Rahmen dieser Arbeit ebenfalls vorgestellt.

Die Beiträge der Arbeit lassen sich somit wie folgt zusammenfassen:

- ein auf Hypergraphen und GTRs basierender Formalismus zur Realisierung von Editoren für interaktive animierte Sprachen (AAS/GT),
- eine Modellierungssprache zur Spezifikation von Struktur, Darstellung und Verhalten einer interaktiven animierten Sprache (AML),
- eine Erweiterung dieser Modellierungssprache um graph-spezifische Konstrukte (AML/GT),
- die Übersetzung von AML/GT-Modellen in den AAS/GT-Formalismus,
- ein modellgetriebener Ansatz zur Generierung von Editoren für interaktive animierte Sprachen, bei dem die zuvor beschriebenen Mittel eingesetzt werden.

Das beschriebene Konzept wurde dabei durch vollständige Implementierung und Fallstudien validiert:

- das Meta-Tool DiaMeta wurde um die Konzepte des AAS/GT-Formalismus erweitert, wodurch die Generierung von Editoren für interaktive animierte Sprachen ermöglicht wird,
- für die Modellierungssprachen AML und AML/GT wurden Editoren entwickelt, die den kompletten Sprachumfang unterstützen und praktisch eingesetzt werden können,
- es wurden Modelltransformationen umgesetzt, die AML/GT-Modelle in das DiaMeta-Spezifikationsformat (und zusätzlichen Editor-Code) übersetzen, woraus anschließend funktionsfähige Editoren generiert werden können, und
- mit dem präsentierten Ansatz wurden zahlreiche Editoren für interaktive animierte Sprachen generiert.

1.3 Überblick

Im Folgenden wird der Aufbau dieser Arbeit kurz dargestellt. In Kapitel 2 werden zunächst animierte Sprachen beschrieben. Das Kapitel soll Aufschluss darüber geben, welche Editoren mit dem präsentierten Ansatz generiert werden können. Darüber hinaus werden die animierten Sprachen betrachtet, die als Beispiele innerhalb der Arbeit dienen. Kapitel 3 zeigt detailliert, wie Graphen als interne Repräsentation für Diagramme und Modelle verwendet werden können. Zudem werden Meta-Tools vorgestellt, die aus graphbasierten Sprachspezifikationen Editoren generieren können. Danach wird in Kapitel 4 ein graphbasierter Ansatz und Formalismus (AAS/GT) beschrieben, mit dem animierte Sprachen spezifiziert werden können. Zur Erzeugung entsprechender Spezifikationen soll auch ein modellgetriebener Ansatz präsentiert werden, weshalb in Kapitel 5 die Modellierungssprache AML eingeführt wird. Eine Erweiterung der Sprache (AML/GT) um graph-spezifische Konstrukte wird in Kapitel 6 gezeigt. Der modellgetriebene Ansatz selbst und die Übersetzung von AML/GT-Modellen in eine graphbasierte Sprachspezifikation auf der Grundlage von AAS/GT wird in Kapitel 7 beschrieben. Wichtige Ergebnisse der Arbeit werden abschließend in Kapitel 8 zusammengefasst.

Anhang A zeigt vollständige AML/GT-Modelle zur Umsetzung der in Abschnitt 2.3 skizzierten Sprachbeispiele. Aus diesen Modellen können mit der Beispiel-Implementierung für DiaMeta funktionsfähige Editoren generiert werden. Im Anhang B werden Details der vorgestellten Modellierungssprachen AML und AML/GT durch die Auflistung wichtiger Zusicherungen und Operationen dargestellt. Außerdem werden im Anhang C Ausschnitte von generiertem Code präsentiert.

Kapitel 2

Animierte Sprachen

In der Einleitung wurden bereits einige Merkmale animierter Sprachen beschrieben, allerdings ohne detailliert einzugrenzen, wann eine Sprache als animiert klassifiziert wird. Tatsächlich gibt es sowohl in der Literatur als auch unter Autoren und Nutzern solcher Sprachen ein unterschiedliches Verständnis für den Begriff „animierte Sprache“. Dementsprechend variieren auch die Vorstellungen von Editoren für animierte Sprachen.

Ziel dieses Kapitels ist es, ein Verständnis zu schaffen, welche Arten von Editoren mit der in späteren Kapiteln behandelten Vorgehensweise erstellt werden können. Allerdings wird keine allgemeingültige Begriffsdefinition für animierte Sprachen aufgestellt. Stattdessen werden unterschiedliche Merkmale und Aspekte visueller und animierter Sprachen betrachtet und beschrieben, wie diese eingeordnet werden können (Abschnitt 2.1). Anschließend werden existierende animierte Sprachen, Editoren und Entwicklungsumgebungen vorgestellt, um weitere Eindrücke über Geschichte, Einsatzbereiche und die vielfältigen Möglichkeiten animierter Sprachen zu vermitteln (Abschnitt 2.2). Im letzten Teil folgen ausführlichere Beschreibungen von animierten Sprachen, die als Fallstudie gedient haben und in späteren Kapiteln als wiederkehrende Beispiele zur Beschreibung der Konzepte verwendet werden (Abschnitt 2.3).

2.1 Taxonomie, Begriffsdefinition, Motivation und Einsatzbereiche

Wie einleitend erwähnt, wird in diesem Abschnitt keine allgemeingültige Begriffsdefinition für animierte Sprachen aufgestellt. Es werden allerdings wichtige Aspekte genannt, um den Begriff „animierte Sprachen“ im Rahmen dieser Arbeit einordnen zu können. Dadurch soll verdeutlicht werden, welche Arten von Editoren sich mit dem präsentierten Ansatz realisieren lassen. Zusätzlich sollen in diesem Abschnitt auch die Motivation hinter solchen Sprachen und mögliche Einsatzgebiete beschrieben werden.

Der Begriff „animiert“ bezieht sich in fast allen betrachteten Sprachbeispielen und Arbeiten über solche Sprachen auf grafische Animation, also visuelle Aspekte. Manchmal wird daher auch direkt die Formulierung „animierte visuelle Sprache“ benutzt. Unter diesem Gesichtspunkt sind die betrachteten animierten Sprachen eine echte Teilmenge von visuellen Sprachen, da sie in der Klasse der visuellen Sprachen enthalten sind. Aus diesem Grund wird zunächst Taxonomie und Motivation hinter visueller Sprachen betrachtet, bevor in späteren Unterabschnitten auf animierte Sprachen und deren Einsatzbereiche eingegangen wird. Zusätzlich wird beschrieben, in welcher Form auch „Interaktivität“ in animierten Sprachen eine Rolle spielen kann.

Visuelle Sprachen – Taxonomie und Definition

Unter *visuellen Sprachen* werden im Bereich der Informationstechnik meist formale Sprachen verstanden, die sich durch visuelle Sprachelemente und deren räumliche Anordnung von textuellen Sprachen abgrenzen. Visuelle Sprachen werden dabei häufig verwendet, um Diagramme oder visuelle Modelle zu erstellen und in Verbindung damit auch zur *visuellen Programmierung (VP)*.

Bekannte und vielfach verwendete Taxonomien für visuelle Sprachen, und speziell auch VP als eines der wichtigsten Anwendungsgebiete solcher Sprachen, wurden von Chang [Cha87], Shu [Shu88], Myers [Mye90], Burnett und Baker [BB94] entwickelt. Die Vorstellung von visuellen Sprachen, v. a. in Bezug auf textuellen Sprachen, unterscheidet sich jedoch teilweise.

Aus verschiedenen Gründen werden *textuelle Sprachen* – also rein auf Zeichenketten basierende Sprachen – oftmals nicht zu den visuellen Sprachen gezählt. Nach Schiffer [Sch98] sind visuelle Sprachen ausschließlich Sprachen, die „visuelle Ausdrücke“ enthalten. Sie stehen somit in Kontrast zu „verbalen Sprachen“, wie Schiffer textuelle Sprachen auch nennt. So sei Text zwar ebenfalls visuell, allerdings ist die Visualisierung der Zeichenketten „syntaktisch und semantisch bedeutungslos“. Lakin [Lak86] und Myers [Mye90] nennen einen weiteren Grund, warum textuelle Sprachen nicht zu den visuellen Sprachen gezählt werden sollen. In ihrer Definition benötigen visuelle Sprachen eine mehrdimensionale Anordnung ihrer visuellen Elemente, d. h., Sprachen mit eindimensionaler Anordnung gehören nicht in diese Klasse. Chang [Cha87] und v. a. Myers [Mye90] nennen daher Sprachen, bei denen die Zeichenfolge sequenziell verarbeitet wird, wie es bspw. bei einem Compiler der Fall ist, auch *lineare Sprachen*. Nach klassischer Vorstellung sind textuelle Sprachen solche linearen Sprachen, aber es gehören auch Sprachen dazu, in denen visuelle Elemente wie Piktogramme oder Icons anstatt Wörter linear angeordnet werden.

Auf der anderen Seite gibt es einige Autoren, die textuelle (oder lineare) Sprachen ganz oder teilweise als Untermenge von visuellen Sprachen betrachten. Beispielsweise darf nach Shu [Shu88] jede Sprache zur Bearbeitung visueller Informationen selbst als visuelle Sprache bezeichnet werden. Andere Argumente beziehen sich erneut auf die Visualisierung der Sprache selbst. Texte und Symbole sind selbst visuelle Elemente, die unterschiedlich dargestellt werden können. Viele Texteditoren bieten bspw. die Möglichkeit, Schlüsselwörter der Sprache

mit anderer Farbe oder Schriftauszeichnung anzuzeigen. Schmidt [Sch06] definiert textuelle Sprachen vollständig als eine Obermenge von visuellen Sprachen, wodurch auch jede traditionelle textuelle Programmiersprache (z. B. Java [Ull11] oder C++ [Str00]) eine visuelle Programmiersprache ist. Ein wichtiges Argument für ihn ist dabei, dass es textuelle Sprachen gibt, bei denen die Zeileneinrückung eine entscheidende Rolle spielt (z. B. bei Python [Het02]). Dadurch wird eine Art zweidimensionales Layout verwendet und ein vielfach gefordertes Kriterium für visuelle Sprachen erfüllt (vgl. Lakin [Lak86] und Myers [Mye90]). Er bildet daher eine Untermenge namens „echt visuelle Sprachen“, in der textuelle Sprachen nicht mit eingeschlossen werden.

Im Rahmen dieser Arbeit sollen sowohl textuelle als auch lineare Sprachen in der Klasse der visuellen Sprachen enthalten sein, in der Klasse der *echt visuellen Sprachen* hingegen nicht. Der Begriff „echt visuell“ wird also ähnlich, aber noch restriktiver benutzt als in [Sch06]. In Tab. 2.1 wird diese Auslegung noch einmal kurz und bündig dargestellt.

An dieser Stelle sei allerdings erwähnt, dass der Begriff „echt visuell“ nicht an allen Stellen dieser Arbeit konsequent verwendet wird. In der Regel sollte aber davon ausgegangen werden, dass sich alle Aussagen auf echt visuelle Sprachen beziehen, außer es wird explizit anders angegeben.

visuelle Sprache	$\supset$	echt visuelle Sprache	
$\cup$		$\cup$	Animation $\downarrow$
animierte Sprache	$\supset$	echt animierte Sprache	
keine textuellen oder linearen Sprachen $\rightarrow$			

Tabelle 2.1: Echt visuelle und animierte Sprachen

Im Folgenden sollen noch einmal Definitionen verschiedener (bereits erwähnter) Autoren gelistet werden, die zu der beschriebenen Klasse von echt visuellen Sprachen passen. Die Definition von Myers lautet, wenn auch mit Bezug auf visuelle Programmiersprachen:

> „Visual Programming“ (VP) refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Although this is a very broad definition, conventional textual languages are not considered two dimensional since the compilers or interpreters process them as long, one-dimensional streams. ([Mye90])

Eine ähnliche Definition in Bezug auf visuellen Sprachen wurde von Lakin aufgestellt: „A visual language is a set of spatial arrangements of text-graphic symbols with a semantic interpretation that is used in carrying out communicative actions in the world.“ ([Lak86]) Eine andere Definition liefert Schiffer: „Eine visuelle Sprache ist eine formale Sprache mit visueller Syntax oder visueller Semantik und dynamischer oder statischer Zeichengebung.“ ([Sch98]) Darin bezieht er sich

vor allem auch auf den theoretischen Hintergrund des Begriffs „Sprache“. Der Ausdruck „dynamische Zeichengebung“ verweist außerdem auf die Möglichkeit, flüchtige Vorgänge darstellen zu können, was auch als Animation bezeichnet werden kann. Im Anschluss an diese Definition merkt er ebenfalls an, dass sich Syntax und Semantik visueller Sprachen auf den zwei- oder dreidimensionalen Raum beziehen und nicht etwa auf „eindimensionale Zeichenketten“.

Beispiel 2.1 (Einordnung visueller Sprachen)
Verwendet man die formulierten Kriterien, um existierende Sprachen einzuordnen, so ist dies trotzdem nicht immer problemlos möglich. Nassi-Shneiderman-Diagramme [NS73] haben bspw. einen beträchtlichen textuellen Anteil. Da aber Verzweigungen im Gegensatz zu klassischen „if/else“-Blöcken nebeneinander platziert werden, ist die Einordnung als echt visuelle Sprache naheliegend. Bereits die Verwendung des Begriffs „Diagramm“ weist auf die Wichtigkeit räumlicher Beziehungen hin.

Im Gegensatz dazu sollten Programmiersprachen, wie sie in Alice, Scratch und AgentSheets verwendet werden (vgl. Abschnitt 2.2, S. 16ff), nicht als echt visuelle Sprachen bezeichnet werden. Darin können zwar (farbige) Textblöcke, die auch Steuerelemente zur Eingabe oder Auswahl enthalten können, per Maus erstellt und platziert werden. Aber es entsteht ein Programm, für das Code in äquivalenter Form auch per Tastatur eingegeben werden könnte. Komplexe visuelle Editoren oder integrierte Entwicklungsumgebungen (IDEs) sollen also kein Kriterium für die Einordnung der Sprache darstellen. △

Die meisten IDEs und damit verbundene Texteditoren bieten heutzutage Assistenzsysteme, um mit wenigen Mausklicks komplexen Code erzeugen zu können. Dies ist, wie im Beispiel erwähnt, allerdings kein Kriterium für die Einordnung der darin verwendeten Sprache. Trotzdem werden für solche Sprachen, die keine echt visuellen Sprachelemente bieten, manchmal Namen wie bspw. Visual C++ [Kai09] verwendet.

Visuelle Sprachen – Motivation

Für den Einsatz von visuellen Sprachen sprechen viele Argumente, allerdings sind die wenigsten Vorteile absolut unstrittig. In einigen Fällen bringen visuelle Sprachen sogar Nachteile mit sich, weshalb die Nützlichkeit von visuellen Sprachen in der Literatur sehr kontrovers diskutiert wird. Disputiert werden immer wieder Vor- und Nachteile hinsichtlich Verständlichkeit, Übersichtlichkeit, Platzbedarf, Ausdrucksstärke, Interpretationsspielraum, Darstellbarkeit und Ergonomie. Manche Probleme lassen sich kompakt und einfach mit einer bestimmten visuellen Sprache darstellen, während es bei anderen Problemen zu typischen Platz- und Layoutproblemen kommt. Einige Grafikelemente und Layoutkonzepte können zwar intuitiv verstanden werden, bieten aber gleichzeitig die Möglichkeit für Fehlinterpretationen oder Mehrdeutigkeiten.

Dies waren nur einige beispielhafte Argumente für und gegen visuelle Sprachen. In der Regel sind die Vor- und Nachteile in hohem Maße abhängig von der Problemstellung und der konkret verwendbaren Sprache. Es sollte daher immer

sorgfältig untersucht und abgewogen werden, ob der Einsatz einer visuellen Sprache sinnvoll ist. In Bezug auf visuelle Programmierung schreibt Schiffer in der Einleitung seines Buches:

> Visuelle Programmierung ist zu einem Synonym für intuitive und mühelose Softwareentwicklung geworden. Das Ziel visueller Programmierung ist vor allem die Erhöhung der Verständlichkeit von Programmen und die Erleichterung der Programmierung selbst. Durch visuelle Programmierung sollen auch Anwender in die Lage versetzt werden, Applikationen für den eigenen Bedarf zu erstellen, für deren Programmierung beim Einsatz konventioneller Werkzeuge und Sprachen professionelle Softwareentwickler benötigt würden. ([Sch98])

Da die genannten Ziele nicht immer erreicht werden, hat er folgende fünf Thesen kritisch analysiert:

- visuelle Programme erleichtern Kommunikation,
- visuelle Programme sind leicht verständlich,
- visuelle Programmierung ist leicht erlernbar,
- visuelle Programmierung überwindet Sprachbarrieren,
- visuelle Programmierung heißt optimale Kognition.

Er listet dabei Autoren und Arbeiten auf, welche die jeweilige These stützen oder eine Gegenposition beziehen. In einem abschließenden Fazit werden meist Voraussetzungen geschildert, unter denen die positiven Aspekte deutlich überwiegen.

Insofern ist auch einer der Leitgedanken dieser Arbeit, dass visuellen Sprachen gegenüber textuellen Sprachen für bestimmte Bereiche und Aspekte, aber sicherlich nicht alle, im Vorteil sein können.

Konkrete und abstrakte Syntax

In Anlehnung an textuelle (formale) Sprachen werden auch bei visuellen Sprachen die Begriffe „abstrakte Syntax“ und „konkrete Syntax“ oder auch „konkrete/abstrakte Sprachsyntax“ verwendet (bspw. in [FB05, Maz10, Erm06]). Die *konkrete Syntax* legt dabei fest, wie die Darstellung der Sprachelemente erfolgt. Im Falle von textuellen Sprachen werden auf dieser Ebene bspw. die notwendige Formatierung, die Zeichenketten für Schlüsselwörter oder Symbole festgelegt. Für Diagrammsprachen werden grafische Elemente vorgegeben wie Rechtecke, Linien, Text etc. Zusätzlich muss definiert werden, wie derartige Sprachelemente angeordnet werden.

Die konkrete Syntax stellt somit auch die Schnittstelle zum Benutzer dar. Im klassischen Compilerbau wird diese meist durch Grammatiken spezifiziert, z. B. durch die Erweiterte Backus-Naur-Form (EBNF) [ISO96]. Basierend auf der Grammatik kann die Syntax des Programmcodes geprüft werden, indem der

Code (die konkrete Syntax) zunächst in Symbole umgewandelt wird (*lexikalische Analyse*), um danach zu versuchen einen (konkreten) Syntaxbaum abzuleiten (*syntaktische Analyse*).

Der konkrete Syntaxbaum enthält im Regelfall viele Symbole, die semantisch unwichtig sind, wie bspw. Klammern oder Zeichen, die den Abschluss einer Anweisung markieren. Aus diesem Grund wird der konkrete Syntaxbaum häufig in einem weiteren Schritt vereinfacht bzw. abstrahiert. Hierdurch entsteht der abstrakte Syntaxbaum, mit dem semantische Analyse, Interpreter, Optimierer und Codegeneratoren effizienter arbeiten können. Die *abstrakte Syntax* der Sprache legt dabei fest, wie dieser Baum aufgebaut sein muss. Sie bestimmt daher lediglich die interne Struktur der dargestellten Daten und ist unabhängig von der grafischen Repräsentation der Sprachelemente.

Auch bei visuellen Sprachen ist dies ein verbreitetes Verfahren. Allerdings wird in diesem Fall meist ein konkreter Syntaxgraph (KSG) und ein abstrakter Syntaxgraph (ASG) abgeleitet, da Bäume bei visuellen Sprachen aufgrund der Möglichkeit von mehrdimensionaler Anordnung nicht ausreichen. Aus diesem Grund werden für die Sprachspezifikation häufig auch Graphgrammatiken genutzt. Ein alternativer Ansatz, der auch bei der Generierung von Editoren in dieser Arbeit verwendet wird, ist die Verwendung von Metamodellen zur Spezifikation der abstrakten Syntax. Ausführlich beschrieben wird diese Methode in [Min06b].

Im Hinblick auf animierte Sprachen ist festzuhalten, dass Animation nach Auffassung dieser Arbeit klar der konkreten Syntax zuzuordnen ist. Daher spielt die abstrakte Syntax im weiteren Verlauf eine untergeordnete Rolle und die präsentierten Techniken beziehen sich meist auf die konkrete Syntax. Allerdings sind Animationen häufig mit strukturellen Änderungen verknüpft, die sich dann auch in der abstrakten Repräsentation widerspiegeln.

Animierte Sprachen – Taxonomie und Definition

Im Gegensatz zum Begriff „visuelle Sprache“, wird der Begriff „animierte Sprache“ in der Literatur sehr selten verwendet oder gar definiert. So wurde der Begriff in [MK08] in Zusammenhang mit „animierter Programmierung“ verwendet, bei welcher der „Code selbst animiert ist“. Im Rahmen dieser Arbeit soll der Begriff aber allgemeiner und einfacher aufgefasst werden. Als *(echt) animierte Sprache* wird daher jede (echt) visuelle Sprache bezeichnet, die Animationen bei ihrer Visualisierung einsetzt (vgl. Tab. 2.1, S. 9).

Animation wird dabei meist als schnelle Bildfolge definiert, die Veränderungen oder Bewegung von Objekten vermitteln soll. Nach Keller [KK93] ist Animation bspw. eine „sequence of related images viewed in rapid succession to see and experience the apparent movement of objects.“ Noch allgemeiner ist die Definition von Henning speziell für Computeranimation: „Eine Animation ist ein Satz von Multimedia-Daten, die paketweise räumlich korreliert sind und von Paket zu Paket eine zeitliche Korrelation aufweisen.“ ([Hen07]) Durch diese Definition werden neben Bildern auch Text oder sogar Klang einbezogen, wo-

bei letztgenannter in animierten Sprachen zwar möglich ist, allerdings wird im Folgenden nicht mehr darauf eingegangen.

Trotz dieser Definitionen ist es schwierig, eine Sprache eindeutig als animiert oder nicht animiert einzuordnen. Beispielsweise ist in der Formulierung „schnelle Bildfolge“ nicht genau festgelegt, was „schnell“ bedeutet und wie viele Bilder eine solche Folge mindestens haben muss. Im Bereich der Computeranimation ist ein Wert um die zehn Bilder pro Sekunde zwar ein gebräuchlicher Minimalwert [OO99], allerdings auch nur, um eine Animation als „flüssig“ zu bezeichnen. Interessant ist daher die Frage, ob eine Sprache als animiert bezeichnet werden kann, die Änderungen nur in „wenigen“ Bildfolgen und Bewegungen bzw. in „großen“, diskreten Schritten anzeigt. Es gibt viele Sprachen, die Zustandsänderungen nicht flüssig animieren, sondern nur den Zustand vorher und nachher darstellen. In diesem Fall würde man vermutlich eher von Simulationsschritten als von Animation sprechen. Wenn also Zustandsübergänge ohne Zwischenschritte visualisiert werden und solche Zustandsübergänge nur langsam (z. B. länger als eine Sekunde) oder auf Benutzeranforderung durchgeführt werden, wird die Sprache im Rahmen dieser Arbeit nicht als animiert bezeichnet.

Eine solche Sprache kann allerdings als *animierbar* bezeichnet werden, falls es möglich ist, die Sprache durch Erweiterungen zu einer animierten Sprache zu machen. Als Beispiel können Petri-Netze (siehe Abschnitt 2.3.1, S. 32) oder visuelle Sprachen für Zustandsautomaten (z. B. UML-Zustandsdiagramme [UML11]) betrachtet werden. Falls eine Sprache, wie die genannten Beispiele, eine Ausführungssemantik besitzt, so ist die Simulation einer solchen Ausführung möglich. Im Rahmen einer solchen Simulation können dann meist zusätzliche Informationen visualisiert werden. Im Falle von UML-Zustandsdiagrammen sind dies bspw. aktive Zustände und schaltende Zustandsübergänge. Wie solche Zustände oder Zustandsübergänge dargestellt werden, ist für die Sprache selbst allerdings nicht definiert. Tatsächlich lassen sich Zustandsübergänge mit Animation aber gut visualisieren, d. h., entsprechende Sprachen sind animierbar. Wird ein konkreter Editor für eine solche Sprache implementiert und die Sprache durch Animationsspezifikationen und ggf. zusätzlich notwendige Konstrukte ergänzt, wird die animierbare Sprache zur animierten Sprache erweitert.

Allerdings müssen weitere Kriterien festgelegt werden, denn nach obiger Auslegung ist jede visuelle Sprache animierbar. Für eine Sprache wie UML-Klassendiagramme [UML11] könnte bspw. spezifiziert werden, dass der Rahmen einer aktuell selektierte Klasse innerhalb des Editors rot blinken soll. Es sollen daher nur Sprachen als „animierbar“ bezeichnet werden, die eine Ausführungssemantik enthalten, welche mit Animationen gekoppelt werden kann. Derartige Spracherweiterungen werden auch in [BSE11] beschrieben.

Auch der Begriff „dynamisch“ wird in Bezug auf visuelle Sprachen häufig gebraucht. In [Min98] wird der Begriff „dynamische Diagrammsprache“ verwendet, um anzudeuten, dass sich Diagramme dieser Sprache bspw. im Rahmen von Simulationen oder Auswertungen ändern können. Dies kann durch Animation visualisiert werden. Auch Schiffer [Sch98] spricht von „dynamischer Zeichengebung“ als mögliches Merkmal visueller Sprachen, was Animation ebenfalls einschließt.

Gesondert muss der Begriff „dynamisch“ allerdings in Zusammenhang mit Programmiersprachen betrachtet werden. Bei Verwendung einer sogenannten *dynamischen Programmiersprache* kann das Programm selbst während der Laufzeit verändert werden (vgl. [NBD+05]), d. h., es können z. B. Codeblöcke, Datenstrukturen oder sogar Typsystem modifiziert werden. Einige animierte Programmiersprachen fallen ebenfalls in diese Klasse, z. B. falls die Programmierumgebung mit der Umgebung der Programmausführung zusammenfällt (siehe ToonTalk in Abschnitt 2.2, S. 18 oder Alternate Reality Kit in Abschnitt 2.2, S. 20).

Interaktivät in animierten Sprachen

Interaktivität ist eines der wichtigsten Kriterien zur Einordnung einer animierten Sprache. Im Rahmen dieser Arbeit sollen drei Fälle unterschieden werden. Zum einen ist es möglich, dass Animationen einer Sprache während des Ablaufs in keiner Weise gesteuert werden können, d. h., sie ist *nicht interaktiv*. Dazu zählen bspw. Sprachen, die Animationen ausschließlich als Blickfang visueller Elemente nutzen. Ein anderes Beispiel sind Sprachen, in denen ein Diagramm zunächst vollständig erstellt werden muss, bevor eine Simulation inkl. Animationen in einer selbstablaufenden Visualisierung abgespielt werden kann.

Die zweite Möglichkeit ähnelt der ersten. Allerdings gibt es nach den selbstablaufenden Abschnitten (z. B. nach einem Simulationsschritt) immer wieder die Möglichkeit, Einfluss auf den weiteren Ablauf zu nehmen. Hierfür pausiert die Animation, bis bspw. ein Editornutzer eine entsprechende Aktion ausgeführt hat. Solche Sprachen werden im Folgenden als *teilweise interaktiv* bezeichnet.

Abschließend können animierte Sprachen noch als *vollständig interaktiv* bezeichnet werden, falls der Ablauf zu beliebigen Zeitpunkten beeinflusst werden kann. Es ist allerdings zulässig, wenn es bestimmte Passagen während des Ablaufs gibt, in denen der Editornutzer keinerlei Einfluss nehmen kann.

Animierte Sprachen – Motivation

Häufig wird Animation in Sprachen eingesetzt, um Änderungen (oder konkret einen Datenfluss) für den Benutzer nachvollziehbar darzustellen. Denn Animation kann dann wichtig sein, wenn Änderungen nur schwierig erkannt werden können, z. B. falls viele gleichzeitig durchgeführt werden. Eine These besagt daher, dass Visualisierungen oder Algorithmen durch Animation verständlicher dargestellt werden können. Unter bestimmten Voraussetzungen werden solche positiven Effekte bei animierten Simulationen von einigen Studien auch bestätigt (z. B. [Rie96]).

Ein bekannter Ausspruch diesbezüglich stammt von Knuth: „An algorithm must be seen to be believed“ ([Knu73]). Auch Kahn und Saraswat betonen die Wichtigkeit von Animation: „A computer system needs to do more than produce some stream of outputs given some stream of inputs. The ability to observe its intermediate states is essential for understanding what a computation is doing.“ ([KS90b]) Kahn schreibt in einer späteren Arbeit außerdem: „Animation is much better suited for dealing with the dynamics of computer programs

than static icons or diagrams.“ ([Kah96]) Arbeiten wie [RAK06] beschäftigen sich daher mit Möglichkeiten und dem Potential von Algorithmenvisualisierung und -animation. Allerdings konnte die Nützlichkeit von Algorithmenanimation [Bro88] für bessere Lernergebnisse nicht eindeutig nachgewiesen werden (vgl. [SBL93, BCS96, KST01, NRA+02, Hun02]). In einigen Arbeiten wird aber auch darauf hingewiesen, dass Interaktionsmöglichkeiten während der Algorithmenanimation zu besseren Lernergebnissen bei Studenten führen. Naps et al. schreiben diesbezüglich: „Our thesis is that visualization technology, no matter how well it is designed, is of little educational value unless it engages learners in an active learning activity.“ ([NRA+02]) Auch Hundhausen weist darauf hin: „Our most significant finding is that how students use algorithm visualization technology has a greater impact on effectiveness than what algorithm visualization technology shows them.“ ([Hun02]) Und Tversky et al. schreiben: „Interactivity may be the key to overcoming the drawbacks of animation as well as enhancing its advantages.“ ([TMB02])

Neben Gründen wie die der Verständlichkeit kann Animation auch schlicht genutzt werden, um eine Sprache optisch ansprechend darzustellen. Vor allem für Laien können ästhetische Gesichtspunkte ausschlaggebend sein, ob die Sprache gern genutzt wird oder deren Nutzung sogar Spaß vermittelt: „By presenting programming through objects in a visual and animated way, students find programming fun.“ ([Rod02])

Schließlich gibt es noch Sprachen, die ohne Animation nicht realisiert werden können. Dies sind bspw. Sprachen, in denen eine virtuelle, animierte Welt nachbildet wird. Auch Physik kann in solchen Sprachen eine Rolle spielen. Beispiele hierfür ist ToonTalk von Kahn (Abschnitt 2.2, S. 18) oder das Alternate Reality Kit von Smith (Abschnitt 2.2, S. 20). Kahn beschreibt die Idee seiner Sprache folgendermaßen und nennt auch die Motivation hinter solchen Sprachen:

> The programmer controls a „programmer persona“ in this video world to construct, run, debug and modify programs. We believe that ToonTalk is especially well suited for giving children the opportunity to build real programs in a manner that is easy to learn and fun to do. ([Kah96])

Animierte Sprachen – Einsatzbereiche

Im Folgenden werden einige Einsatzbereiche für animierte Sprachen beispielhaft aufgelistet. Es wird allerdings weder Anspruch auf Vollständigkeit der Einsatzmöglichkeiten gestellt, noch darauf, dass die einzelnen Bereiche klar voneinander abgegrenzt werden können.

Wie im vorherigen Abschnitt bereits angedeutet, eignen sich animierte Sprachen für den Bereich der Algorithmenanimation [Bro88]. Sie werden in diesem Zusammenhang auch häufig in der Lehre eingesetzt. Die Hoffnung dabei ist, dass die Sprachen den Schülern und Studenten Algorithmen oder Programmierkonzepte auf anschauliche Weise näherbringen. Es ist daher nicht verwunderlich, dass es sich bei einem großen Teil existierender animierter Sprachen um Pro-

grammiersprachen für Kinder handelt. Auch viele der in Abschnitt 2.2, S. 16ff, beschriebenen Sprachen (Alice, ToonTalk, Scratch etc.) haben Kinder, Studenten und Einsteiger als Zielgruppe.

Häufig tritt dabei auch ein spielerischer Hintergrund hervor, d. h., dass die Programmierung spielerisch erfolgt oder die animierte Sprache zur Programmierung einfacher Spiele dient. Interessant sind hier z. B. Sprachen, mit denen Level von Spielen erstellt und angepasst werden können. Es existieren bereits unzählige Editoren, um bspw. Strecken von Autorennspielen zu erstellen oder Tische von Flipperautomaten, die sich aus relativ frei platzierbaren Komponenten zusammensetzen. Gegebenenfalls kann der erstellte Level dann mit einer animierten Sprache getestet bzw. gespielt werden (siehe Avalanche in Abschnitt 2.3.3, S. 43).

Animierte Sprachen können aber auch im professionellen Umfeld eingesetzt werden. Für Simulationen und die Visualisierung von Abläufen können Animationen ein wichtiges Hilfsmittel darstellten, v. a. in Bezug auf Verständlichkeit und Nachvollziehbarkeit. Animationen können auch speziell im Rahmen des Debuggings eingesetzt werden (vgl. [MS94]). In diesem Zusammenhang wird manchmal auch „Rapid Prototyping“ erwähnt, da animierte Sprachen zeitnahe Tests ermöglichen. Häufig werden animierte Sprachen auch im Bereich eingebetteter Systeme und Robotersteuerung verwendet, wie z. B. LabView (siehe Abschnitt 2.2, S. 17) und Ruru (siehe Abschnitt 2.2, S. 22). Ebenfalls interessant ist Animation im Bereich visueller domänenspezifische Sprachen (DSLs) und sprachorientierter Programmierung [War94, Dmi04], in dem es zahlreiche animierbare Sprachen gibt, wie bspw. Ruru und Machinations (siehe Abschnitt 2.2, S. 25).

Ein weiteres professionelles Einsatzgebiet liegt im Bereich der Datenvisualisierung, z. B. um bestimmte Entwicklungen durch Animation hervorzuheben. Beispiele animiert visualisierter Daten und zugehöriger Sprachen gibt es dabei sehr viele aus zahlreichen Fachgebieten, wie bspw. dem Finanzwesen [TK07], Softwareentwicklung [LSP08], Geographie, Medizin etc.

2.2 Existierende animierte Sprachen und Editoren

Im Folgenden werden einige existierende Sprachen vorgestellt, die ganz, teilweise oder nur unter bestimmten Gesichtspunkten als animierte Sprache eingeordnet werden können. Meist sind diese Sprachen in zugehörigen Editoren oder ganzen Entwicklungsumgebungen integriert. Im Falle von Programmiersprachen werden diese manchmal auch VP-System genannt.

Die Sprachen selbst tragen dabei nicht immer einen eigenen Namen. Ein Grund dafür ist, dass sie manchmal gar nicht als Sprache wahrgenommen werden. Stattdessen betrachtet man das komplette System, in dem auch entsprechende Editoren, Compiler/Interpreter und Framework integriert sind. Schiffer schreibt hierzu: „Viele VP-Systeme integrieren die Sprache so stark, daß die Grenzen zwischen Programmiersprache und Programmierumgebung verschwimmen.“ ([Sch98]) Auch in den aufgeführten Beispielen ist die Trennung zwischen System und Spra-

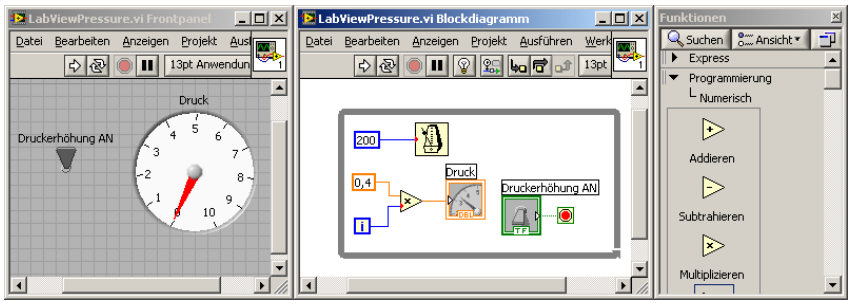


Abbildung 2.1: Screenshot – LabView

che nicht immer einfach. Dennoch wird der Versuch unternommen, diese Teile voneinander abzugrenzen, um die Merkmale der jeweiligen Sprachen gesondert betrachten und im Kontext der vorliegenden Arbeit einordnen zu können.

Bei der Auswahl der Sprachbeispiele wurden meist bekannte Vertreter einer bestimmten Kategorie ausgewählt, in denen sich Sprachen durch bestimmte Charakteristika ähneln. Zusätzlich war auch die Verfügbarkeit entsprechender Literatur wichtig. Insgesamt zeigen sie viele unterschiedliche Ideen. Allerdings wird kein Anspruch auf Vollständigkeit möglicher Eigenschaften solcher Sprachen gestellt. Am Ende dieses Abschnitts wird noch einmal ein knapper Überblick in Form einer Tabelle dargestellt (siehe Tab. 2.2, S. 31).

LabView

Bei LabView [JH01, Lab12] handelt es sich um eine kommerzielle IDE und ein VP-System, das hauptsächlich für den Bereich der elektronischen Mess-, Regel- und Automatisierungstechnik entwickelt wurde und professionell eingesetzt wird. Es war in der ersten Version bereits 1986 verfügbar, hat sich im Laufe der Jahre aber ständig weiterentwickelt und ist inzwischen auf den meisten aktuellen Plattformen lauffähig.

Die IDE bietet mehrere integrierte Editoren für die Erzeugung der Programme. Abb. 2.1 zeigt die beiden wichtigsten Editoren: das Frontpanel (links) und das Blockdiagramm (mittig). Außerdem wird ein Fenster mit verwendbaren Elementen dargestellt (rechts), das je nach Editierkontext verfügbare Bausteine in unterschiedlichen Kategorien anzeigt.

Das *Frontpanel* dient zur Platzierung und Anpassung visueller Bedien- und Anzeigeelemente der grafischen Benutzeroberfläche (GUI). Neben klassischen Steuerelementen wie Knöpfen etc. sind dies vor allem auch Anzeigekomponenten, deren Aussehen sich an realen elektronischen Anzeigen orientiert. Im Kontext von LabView werden diese daher auch virtuelle Instrumente (VIs) genannt. Bei der Ausführung der Applikation werden die VIs des Frontpanels dann genutzt, um einerseits Daten zu visualisieren oder als Bedienelement für den Nutzer eingesetzt zu werden. Die VIs sind dabei sogar teilweise animiert, z. B. wenn sie kontinuierlich ihr Anzeige ändern, falls sich die zugrunde liegenden Daten

ändern. Auf der einen Seite ist das Frontpanel bei Programmausführung also die GUI. Auf der anderen Seite handelt es sich beim Frontpanel um einen typischen Editor einer Art, deren Vertreter auch als GUI-Designer bezeichnet werden, wie bspw. auch der GuiBuilder [SE07]. Solche GUI-Designer sind bereits in vielen anderen IDEs integriert. Die VIs können dabei als visuelle Sprachkomponenten interpretiert werden, die bei Ausführung sogar interaktiv reagieren und animiert dargestellt werden müssen.

Im *Blockdiagramm* wird die eigentliche Anwendungslogik „programmiert“. Die VIs und sonstigen Elemente sind nach Erstellung im Frontpanel automatisch im Diagramm als Icon/Piktogramm verfügbar. Zusätzlich können darin Kontrollstrukturen (z. B. Schleifen, Sequenzen etc.) oder typische Funktionsblöcke (z. B. arithmetische Funktionen etc.) erstellt werden. Durch Verbindung der Elemente an unterschiedlichen Ein- und Ausgängen wird letztendlich der Datenfluss spezifiziert. Es wird festgelegt, wie Daten verarbeitet werden oder die Verarbeitung gesteuert werden kann und wie sie durch die VIs angezeigt werden müssen. Außerdem lassen sich darin, ähnlich wie in typischen Debuggern für textuelle Sprachen, Haltepunkte setzen oder Einzelschritte ausführen. Dabei können auch sogenannte Sonden verwendet werden, um den Datenfluss zu beobachten.

Die Sprache, die innerhalb des Blockdiagramms eingesetzt wird, hat den Namen G erhalten. Es handelt sich dabei um eine visuelle Datenflusssprache [Sch98]. Einige der wichtigsten Elemente der Sprache wurden bereits genannt: Icons als Repräsentation der VIs, Funktionsblöcke, Kontrollstrukturen oder grafisch dargestellte Konstanten, Variablen und Listen. Viele dieser visuellen Elemente besitzen zusätzlich Attribute, die meist textuell angezeigt und editiert werden können. Wird ein Programm ausgeführt, so ist es möglich, den Datenfluss im Blockdiagramm in animierter Form anzeigen zu lassen, was vor allem zur Beobachtung des Datenflusses und damit für Fehlererkennung und -beseitigung hilfreich sein soll. Es ist daher naheliegend, G als animierbare Sprache zu bezeichnen.

ToonTalk

ToonTalk ist eine VP-Umgebung, in der eine virtuelle Welt sowohl zur Programmierung als auch zum Programmablauf genutzt wird. Die Umgebung, deren erste Version auf das Jahr 1995 zurückgeht, gilt unter vielen Autoren als Prototyp einer animierten Programmiersprache. Der Entwickler der Umgebung, Ken Kahn, schreibt, dass der Quellcode der Programme selbst animiert ist [Kah96]. Er charakterisiert ToonTalk außerdem als eine Kombination von Programmiersprache und Computerspiel. Neben der Tatsache, dass die Sprache vor allem für Kinder konzipiert wurde, ist sie vor allem experimentell einzustufen. Einerseits wurde untersucht, wie ein solches System als Lernumgebung eingesetzt werden kann [Kah98, Kah04]. Andererseits ist sie auch Vertreter der logischen Sprachen für constraintbasierte und parallele Programmierung.

Die Programmierung erfolgt in einer animierten virtuellen Welt, die isometrisch dargestellt wird, was manchmal auch als 2,5D-Darstellung bezeichnet wird. Der Programmierer kann seinen Avatar (vgl. [MK08]) innerhalb dieser Welt per Maus bewegen. Die Welt besteht in erster Linie aus Häusern, „in“ denen

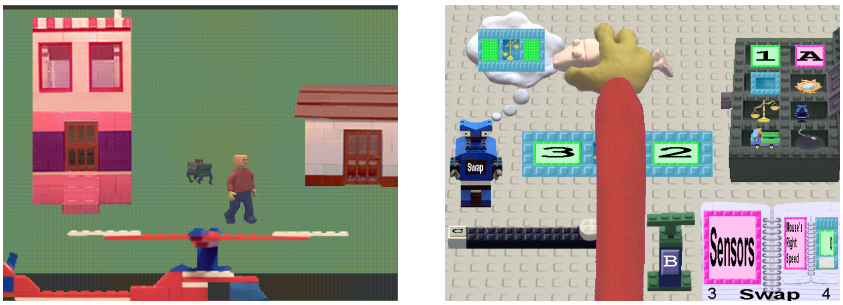


Abbildung 2.2: Screenshots – ToonTalk

Berechnungen durchgeführt werden. Sie können daher als eigenständige Prozesse betrachtet werden. Es können außerdem neue Häuser erbaut werden, wodurch ganze Städte entstehen, in der sich der Avatar mittels Helikopter fortbewegen kann (siehe Abb. 2.2, links).

Innerhalb eines Hauses erfolgt die eigentliche Programmierung. Nachdem sich der Avatar dort auf den Boden gekniet hat, sieht der Benutzer den Boden als Arbeitsfläche vor sich. Zusätzlich öffnet sich eine Toolbox und ein Notizbuch, die verschiedene Werkzeuge, Programmbausteine oder bereits programmierte Elemente (Notizen) bieten. Der Benutzer kann diese mit einer durch die Maus gesteuerten Hand greifen und so Komponenten auf den Boden ablegen oder mit auf dem Boden liegenden Elementen interagieren (siehe Abb. 2.2, rechts). Werkzeuge sind bspw. ein Zauberstab, mit dem Elemente kopiert, eine Fahrradpumpe, mit der Objekte vergrößert oder verkleinert (z. B. angewendet auf Zahlen werden diese inkrementiert bzw. dekrementiert), oder eine Bombe, mit der Element zerstört werden können. Interaktive Hilfe wird von einem Marsmenschen namens Marty angeboten. Zu möglichen Programmbausteinen zählen u. a. folgende Elemente, wobei jeweils in Klammern vermerkt ist, welches Konzept repräsentiert wird: Blöcke mit Zahlen (Konstanten und Variablen), Kisten (Arrays), Waage (Vergleiche), Roboter (wiederverwendbare Prozeduren), Vögel (Versenden von Nachrichten), Nest (Empfang von Nachrichten) etc. Eine detaillierte Übersicht über verfügbare Sprachelemente kann in [MK08] gefunden werden. Die genannte Arbeit hat v. a. den Zweck ToonTalk als Sprache genauer zu spezifizieren.

Abb. 2.2 (rechts) zeigt bspw. die Programmierung eines Roboters, der den Namen „Swap“ trägt. Roboter können sich bestimmte Aktionen „merken“, die zuvor vorgeführt wurden. Diese Vorgehensweise ähnelt also der Erstellung von Makros. Nachdem ein Roboter eine Aktionssequenz aufgezeichnet hat, wird dies entsprechend mit einer Gedankenblase angezeigt. Die Aufzeichnungen können im Anschluss auch (ähnlich wie bei einem Makro) editiert werden, was in manchen Fällen sogar notwendig ist. Im gezeigten Fall hat der Roboter gelernt, wie zwei (beliebige) Zahlen innerhalb einer Kiste mit zwei Fächern vertauscht werden können. Der Roboter kann diese Aktion danach jederzeit auf Abruf erneut (animiert) durchführen.

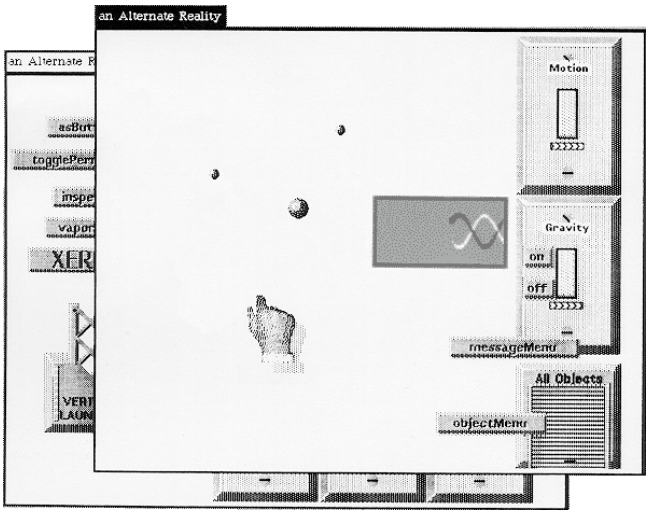


Abbildung 2.3: Screenshot – ARK

Alternate Reality Kit (ARK)

Das Alternate Reality Kit (ARK) [Smi87], das 1987 von Randall Smith veröffentlicht wurde, ist ein weiteres Beispiel einer VP-Umgebung, in der eine animierte, virtuellen Welt sowohl zur Programmierung als auch zur Ausführung von interaktiven Simulationen dient. Das verfolgte Ziel des ARK war, dass die meisten Mechanismen der VP-Umgebung auf (v. a. physikalischen) Gegebenheiten der realen Welt beruhen und somit auch Kinder oder unerfahrene Benutzer mit dem System umgehen können.

Insofern basieren ARK und ToonTalk auf einer ähnlichen Idee, sind ansonsten aber relativ unterschiedlich. Während ToonTalk versucht, eine reale Welt zu simulieren, wird durch ARK eine einfache zweidimensionale Welt nachgebildet, die große Ähnlichkeiten zu typischen GUIs aufweisen kann, d. h., es können darin auch GUI-Elemente wie Zeichenflächen, Knöpfe etc. genutzt werden. Innerhalb der VP-Umgebung wird kein Avatar durch eine Welt gesteuert. Die Sprachelemente können auf übliche Weise mit der Maus platziert oder verschoben werden. Lediglich eine Hand wird als Mauszeiger eingesetzt, um den Eindruck einer virtuellen Welt zu verstärken (siehe Abb. 2.3).

Der wichtigste Grundsatz von ARK unterscheidet das System ebenfalls von ToonTalk. Dieser lautet, dass alle Objekte (also Sprachelemente) innerhalb der Umgebung einigen physikalischen Gesetzen gehorchen müssen. Diese Gesetze werden von der virtuellen Welt entsprechend simuliert. Die Objekte besitzen hierfür neben der Position auch eine Geschwindigkeit und unterliegen verschiedenen Kräften (z. B. Gravitation). Dies gilt sowohl für einfache grafische Elemente wie Kreise als auch für Steuerelemente, die eigentlich zur Benutzerinteraktion dienen. So kann der Benutzer bspw. eine Schaltfläche per Maus nicht einfach

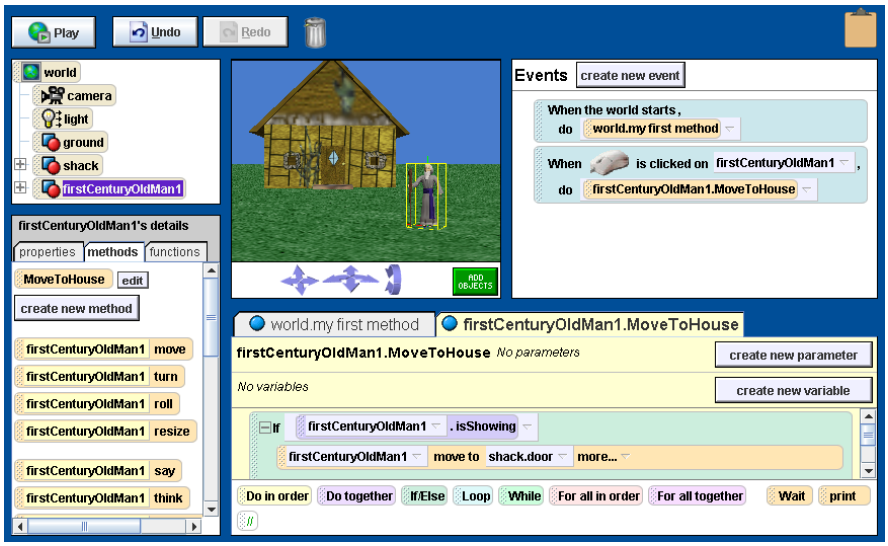


Abbildung 2.4: Screenshot – Alice

nur verschieben, sondern „werfen“, d. h., nachdem der Mausknopf losgelassen wurde, bewegt sich der Knopf weiter.

Knöpfe sind eines der wichtigsten Sprachelemente von ARK. So existieren Knöpfe für sehr viele verschiedene Aktionen. Beispielsweise existiert eine Knopf-Klasse zum Färben grafischer Elemente. Um färben zu können, muss ein entsprechender Knopf erstellt (bzw. aus dem sogenannten „Warehouse“ geholt) und auf dem Element platziert werden. Alternativ kann der Knopf auch durch eine Linie mit dem Element verbunden werden. Mit der Betätigung des Knopfes wird die Färbe-Operation dann ausgeführt.

Alice

Alice [Con97, CDP03] ist eine IDE, deren erste Version im Jahr 1997 veröffentlicht wurde. Sie ermöglicht auf einsteigerfreundliche Art und Weise interaktive 3D-Animationen in einer virtuellen Welt zu programmieren. Im Gegensatz zu ToonTalk und ARK erfolgt die Programmierung aber (größtenteils) nicht in der virtuellen Welt selbst. Die Möglichkeiten reichen dabei von selbstablaufenden Videos bis hin zu einfachen Spielen. Alice wird auch in der Lehre und Forschung eingesetzt, wobei vor allem untersucht wird, wie gut Studenten objektorientierte Programmierung und elementare Konzepte mit Hilfe der grafischen Repräsentationsform der Sprache, der IDE und den erstellten 3D-Animationen erlernen können.

Die IDE besteht aus mehreren Teilen (siehe Abb. 2.4). Neben typischen Sichten für grundlegende Optionen, Programmstruktur und Objektdaten gibt es Sichten für die Darstellung von definierten Ereignissen und Programmteilen (bzw.

Methoden), die vor allem aus Blöcken und Text bestehen. Im Zentrum befindet sich außerdem eine 3D-Ansicht, welche die Objekte des Programms jederzeit darstellt und für Selektion oder Modifikation von Objektattributen genutzt werden kann. Der Programmierer kann sich in der dargestellten virtuellen Welt außerdem frei bewegen und Objekte erzeugen, verschieben oder löschen. Um das Programm ablaufen zu lassen, klickt der Programmierer auf den „Play“-Knopf, und die 3D-Ansicht wird zur Darstellung des ablaufenden Programms.

Entscheidend für die Einordnung von Alice ist, welcher Teil der IDE eigentlich eine Sprache repräsentiert. Dies ist nicht eindeutig zu bestimmen, da die IDE viele unterschiedliche, miteinander verknüpfte Aspekte darstellt. Dennoch sollte man primär die Teile mit grafischen Textblöcken betrachten, da dort die eigentliche Programmierung erfolgt. Die durch die Blöcke dargestellte Sprache[1] ist objektorientiert und ähnelt anderen objektorientierten Sprachen wie Java oder C++. Neben den Zugriffsmöglichkeiten auf Objekte, deren Attribute und Methoden sind auch Elemente der strukturierten Programmierung zu finden, d. h., es existieren Blöcke für Sequenzen, Schleifen und Verzweigungen. Die Code-Blöcke werden wie in textuellen Sprachen untereinander und ineinander verschachtelt angeordnet. Um den Text innerhalb der Blöcke anpassen zu können, enthalten sie zum Teil zusätzliche Steuerelemente wie Textfelder oder Comboboxen.

Alice ist somit eine IDE einschließlich einer eng damit gekoppelten visuellen Sprache (gemeint sind damit die grafischen Textblöcke), mit der sehr einfach Animationen und Abläufe in einer dreidimensionalen Welt programmiert werden können, aber keine echt animierte Sprache an sich. Trotz grafischer Blöcke innerhalb des Abschnitts zur Programmierung handelt es sich darin nach der Auslegung in Abschnitt 2.1, S. 8ff, aber nicht um eine echt visuelle Sprache. Die zusätzliche 3D-Darstellung innerhalb der IDE, welche Objekte und programmierte Animationen anzeigen kann, oder die in der Sprache integrierte 3D-API rechtfertigen nicht die Einordnung als echt animierte Sprache. Shu [Shu88] klassifiziert eine solche Sprache als visuelle Sprache, um visuelle Information oder Interaktion zu verarbeiten.

Die 3D-Sicht und die Interaktionsmöglichkeiten können allerdings gesondert betrachtet werden. Es handelt sich dabei um einen weiteren in die IDE integrierten Editor. Die darin enthaltenen Objekte können als visuelle und animierte Elemente einer eigenen Sprache interpretiert werden. Diese Elemente können genutzt werden, um das Programm zu manipulieren. Dies geschieht allerdings auf anderer Ebene als durch die Sprache mit Textblöcken. Eine Analogie zur Erzeugung von Figuren in dieser 3D-Sicht wäre bspw. die Erzeugung neuer Klassen inkl. Quellcode-Datei, welche mit wenigen Mausklicks in einer Java-IDE mit entsprechenden Assistenzsystemen erfolgt.

Ruru

Ruru [DMH11] ist eine um das Jahr 2011 entwickelte, interaktive animierte Sprache zur Programmierung von Robotern. Aufgrund ihres sehr angepassten

[1] Für die Sprache selbst wurde in der Literatur kein eigener Name gefunden.

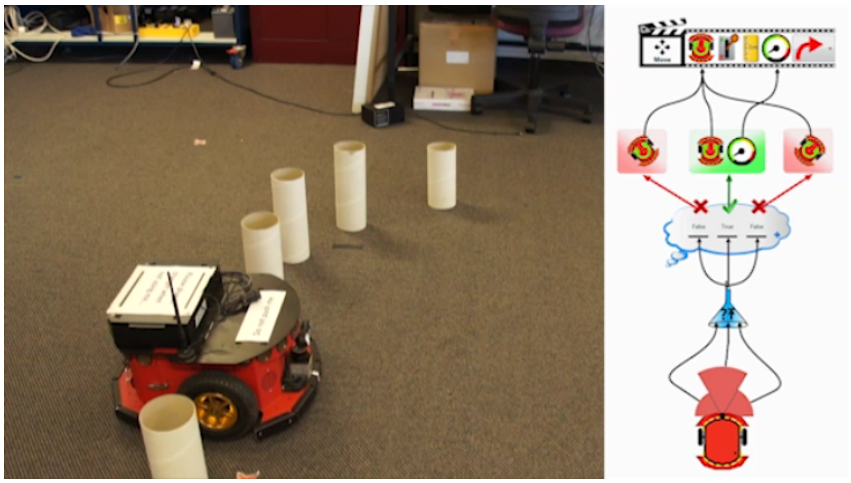


Abbildung 2.5: Ausschnitt aus einem Demonstrationsvideo – Ruru

Einsatzgebietes fällt sie daher auch unter die Kategorie der visuellen DSLs. Die Zielgruppe der Sprache sind vor allem Programmieranfänger. Daher werden grafische Elemente eingesetzt, die intuitiv verständlich sein sollen. Die Notation wurde dahingehend in [DMH11] sogar ausgiebig untersucht und verglichen. Wichtig war bei der Entwicklung der Sprache außerdem, dass Anfänger nicht durch irrelevante und komplizierte Details überfordert werden. Unter anderem deswegen ist die Anzahl der Sprachelemente gegenüber anderen Sprachen derselben Domäne stark reduziert. Sie bietet aber Konstrukte, um typische Programme für Roboter zu entwerfen: Hindernisse umgehen, Wänden oder Objekten folgen, auf bestimmten Pfaden fahren, Bewegungen melden oder externe Befehle befolgen etc.

Interessant ist die Sprache vor allem wegen den Möglichkeiten, die sie zur Laufzeit bietet, d. h., während ein realer Roboter das Programm befolgt. In diesem Fall wird das dargestellte Programm animiert. Es visualisiert bspw., welche Messwerte die Sensoren des Roboters liefern. Der Eindruck einer Animation ergibt sich dabei vor allem durch die kontinuierliche Aktualisierung. Basierend auf den gemessenen Werten, zeigt die animierte Sprache ebenfalls, welche Entscheidungen im Programm getroffen wurden und welche Aktion der Roboter infolgedessen ausführt.

Der Screenshot in Abb. 2.5 (links) zeigt einen Roboter, der zwischen Hindernissen steht. Direkt vor ihm wurde allerdings kein Hindernis platziert. Abb. 2.5 (rechts) zeigt das Programm, das der Roboter ausführen soll. Darin ist der Roboter selbst schematisch ganz unten rechts abgebildet. Direkt darüber befinden sich drei rote Kreissektoren. Sie repräsentieren die Sensoren des Roboters, die entfernte Hindernisse im Umkreis des Roboters erkennen können (links, vorne, rechts). Die Größe der Sektoren wird durch aktuelle Messergebnisse bestimmt. Da sich kein Hindernis vor dem Roboter befindet, ist der mittlere Sektor im Gegensatz zu den anderen sehr groß dargestellt.

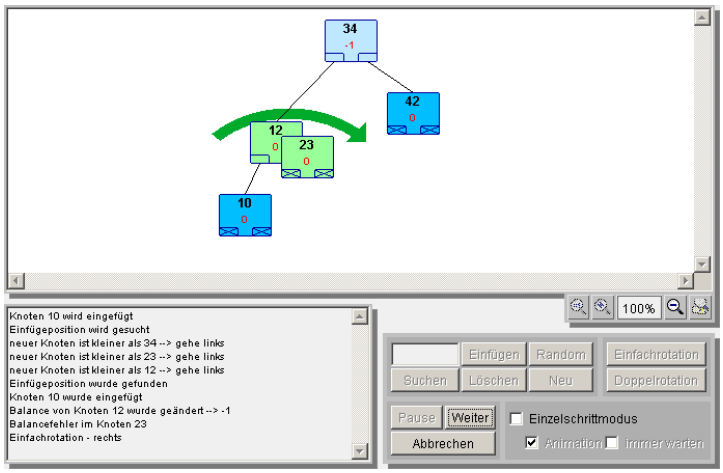


Abbildung 2.6: Screenshot – Java-Applet für AVL-Bäume

Die Abbildungen darüber (Trichter und Wolke) stellen eine Programmverzweigung dar. Darin wird ausgewertet, welcher Sensor aktuell kein Hindernis meldet. Da vor dem Roboter kein Hindernis erkannt wird, ist der mittlere Zweig aktiv. Dieser zeigt daher ein Häkchen und wird grün dargestellt. In dem verknüpften Rechteck darüber sind die Anweisungen zu finden, dass der Roboter sich vorwärts bewegen soll. In den zwei anderen Fällen würde der Roboter die Anweisung erhalten, sich nach links bzw. rechts zu drehen.

AVL-Bäume

Bei AVL-Bäumen [AVL62] handelt es sich nicht um eine Sprache, sondern um eine Datenstruktur. AVL-Bäume sind balancierte, binäre Suchbäume, bei denen die Arbeitsweise der Operationen für das Einfügen, Löschen und Suchen von Elementen im Baum festgelegt ist. Gleichzeitig werden solche Bäume aber relativ häufig visualisiert, um sie anschaulich darzustellen. Visuelle Sprachen zur Visualisierung von binären Suchbäumen, die sich meist nur in Details voneinander unterscheiden, sind daher in der Fachwelt bekannt und werden meist direkt mit der Datenstruktur in Verbindung gebracht. Knoten werden häufig als Kreise oder Rechtecke dargestellt, die außerdem den Schlüssel des Knotens anzeigen. Vater- und Kindknoten werden außerdem mit einfachen Linien verbunden.

Für visuelle Sprachen zur Darstellung von AVL-Bäumen wurden bereits unzählige Editoren implementiert, die es ermöglichen, einen AVL-Baum durch Hinzufügen von Elementen zu erzeugen. Auch das Löschen oder Suchen von Elementen wird oftmals unterstützt. In erster Linie dienen diese Editoren der Visualisierung der Algorithmen, die AVL-Bäumen zugrunde liegen. Sie werden daher v. a. in der Lehre eingesetzt. Einige Editoren unterstützen dabei nur eine schrittweise Darstellung der Algorithmen. Andere animieren die Vorgänge vollständig, z. B. die „Einfachrotationen“ und „Doppelrotationen“ zur Wieder-

herstellung der Balance-Bedingung von AVL-Bäumen. Durch die Animation soll der Zuschauer die Vorgänge einfacher verfolgen und nachvollziehen können (vgl. Algorithmenanimation in Abschnitt 2.1, S. 15). Abb. 2.6 zeigt einen solchen Editor.[2] Die im Screenshot dargestellte Situation erfordert eine „Einfachrotation“, die innerhalb der Applikation auch flüssig animiert dargestellt wird.

Machinations

Abschließend wird ein Beispiel vorgestellt, das noch einmal verdeutlichen soll, dass auch in jüngster Zeit viele Ideen für animierte Sprachen entstanden sind und entsprechende Editoren umgesetzt werden. Machinations [Dor11] ist eine Diagrammsprache, deren Entwicklung auf das Jahr 2009 zurückgeht. Mit der Sprache kann die Mechanik von (v. a. strategischen) Computer- und Brettspielen modelliert und anschließend simuliert werden. Auf diese Weise kann der Spielmechanismus bereits in einem frühen Stadium analysiert und optimiert werden.

Durch die Sprache wird in erster Linie der Fluss von „Ressourcen“ modelliert, z. B. das Geld des Spielers, verfügbare Rohstoffe, Munition etc. Die Sprache bietet hierfür spezielle Knoten, die Ressourcen produzieren, konsumieren oder tauschen. Verschiedene Arten von Pfeilen, die einen Fluss darstellen und die Knoten miteinander verbinden, können ebenfalls modelliert werden. Zusätzlich können Bedingungen für den Fluss der Ressourcen, Wahrscheinlichkeiten und ähnliche Attribute angegeben werden.

Abb. 2.7 zeigt den Editor für Machinations-Diagramme. Das Diagramm selbst modelliert den Ressourcenfluss eines minimalistischen Kriegsspiels für zwei Spieler. Die Elemente der Spieler werden dabei in unterschiedlichen Farben (blau und rot) dargestellt. Im Diagramm existieren bspw. Fabriken (*Factories*), die bestimmte Ressourcen produzieren. Ein Spieler kann zusätzliche Fabriken „kaufen“ (*BuyF*), was Ressourcen kostet, aber gleichzeitig die Produktion der Ressourcen steigert.

Mit diesem Editor ist es außerdem möglich, den Spielablauf zu simulieren. Dieser Ablauf wird animiert dargestellt, d. h., Ressourcen bewegen sich von Knoten zu Knoten, Elemente wechseln zustandsbedingt ihre Farben, blinken usw. Durch das ebenfalls im Screenshot gezeigte Chart – ein Sprachelement – wird außerdem der Verlauf bestimmter Ressourcenmengen in animierter Art und Weise dargestellt. Während der Simulation arbeiten spezielle Elemente nicht automatisch, sondern können vom Editornutzer (bzw. dem Spieler) bedient werden. Durch einen Klick auf *BuyF*, der jederzeit während des Ablaufs möglich ist, wird bspw. eine Fabrik gekauft, sofern der Spieler über genügend Ressourcen verfügt. Es handelt sich also auch um eine interaktive Sprache, worauf in [Dor11] ausdrücklich hingewiesen wird. Dadurch lassen sich bestimmte Aspekte eines Spiels bereits mit dem Machinations-Editor ausprobieren bzw. das Spiel kann in seiner abstrakten Repräsentationsform gespielt werden.

[2] Unter `http://fbim.fh-regensburg.de/~saj39122/bruhi/` können Java-Applet und Beschreibung gefunden werden.

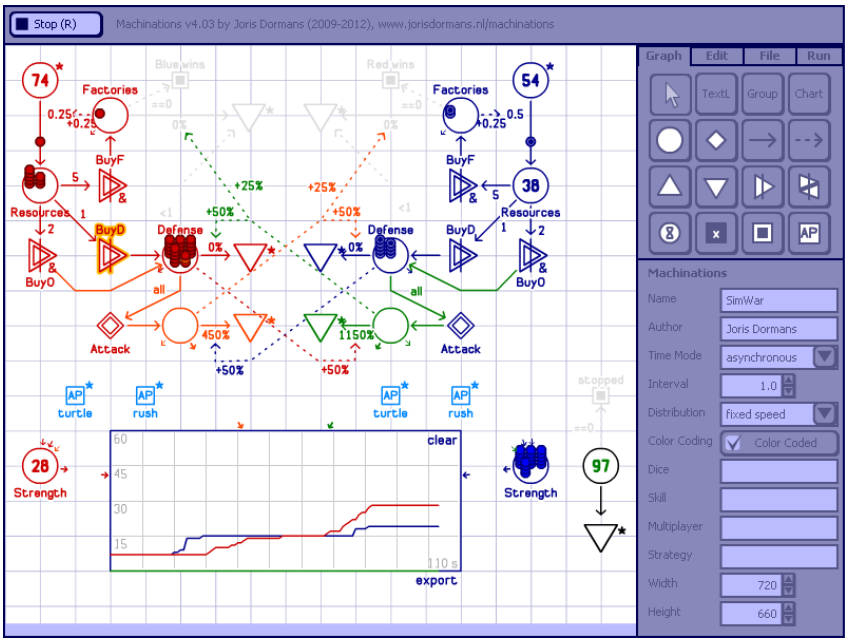


Abbildung 2.7: Screenshot – Machinations

Weitere Sprachen und Editoren

Die beschriebenen Sprachen sind nur Beispiele einer ganzen Fülle von visuellen Sprachen, die teilweise animiert oder zumindest animierbar sind. Einige der folgenden Sprachen sind den bereits vorgestellten Sprachen aber ähnlich oder sind mit ihnen verwandt.

Es gibt einige Sprachen und IDEs, die auf denselben Ideen wie Alice basieren. Die meisten dieser Sprachen sind als Programmiersprachen für Kinder und Lehre konzipiert. Als erstes Beispiel sei die Sprache Scratch [PK07, RMMH+09] genannt, die auf der SmallTalk-Implementierung Squeak basiert. Auch in Scratch wird die Oberfläche in zwei Teile geteilt (siehe Abb. 2.8). In einem Teil, oft auch „Bühne“ genannt (rechts), können unterschiedliche Figuren erstellt und angezeigt werden. Ein wesentlicher Unterschied zu Alice ist allerdings, dass dort nur zweidimensionale Grafiken dargestellt werden. Der zweite Teil der Oberfläche (links) dient zur Programmierung des Verhaltens der Figuren auf der Bühne. Die Programmierung soll dabei wie in Alice einsteigerfreundlich mittels Textblöcken erfolgen. Eine weitere, etwas ältere Sprache, die Scratch sehr ähnelt und ebenfalls auf Squeak basiert, ist Etoys [Kay05]. Neben Alice hat Etoys die Entwicklung der Sprache Scratch wesentlich beeinflusst (vgl. [RMMH+09]).

Ein weiterer geistiger Vorgänger ist AgentSheets [Rep00] (siehe Abb. 2.9). Auch innerhalb dieses VP-Systems kann mittels grafischen Blöcken inkl. Steuerelementen und Drag & Drop programmiert werden. Die entsprechende Sprache wird Visual AgenTalk genannt. Sie dient dazu, das Verhalten der sogenannten

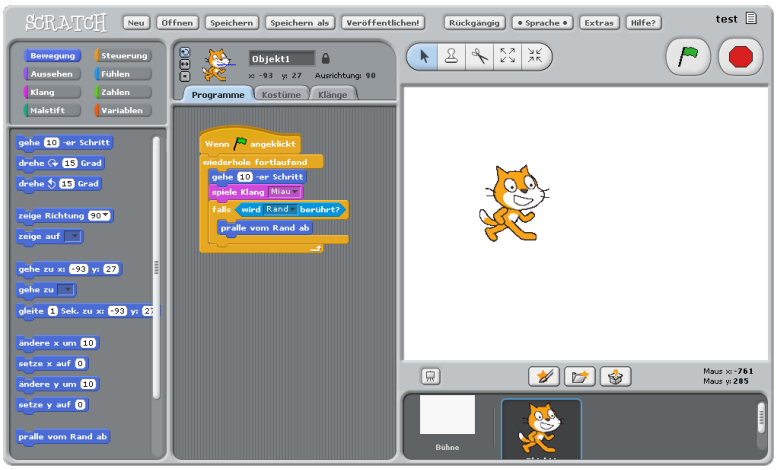


Abbildung 2.8: Screenshot – Scratch

„Agenten“ zu definieren. Diese werden im „Arbeitsblatt“ (engl. worksheet) angezeigt. Ein großer Unterschied zur Bühne in Scratch ist, dass das Arbeitsblatt vollständig aus Kacheln (engl. tiles) besteht, was die Möglichkeiten in AgentSheets auch wesentlich beeinflusst. Dass sich Agenten bspw. nur von Kachel zu Kachel bewegen können, vereinfacht zwar die Programmierung, schränkt allerdings auch die Freiheiten ein. In den meisten Fällen wird trotzdem von Animation gesprochen, auch wenn keine flüssigen Bewegungsabläufe dargestellt werden. Außerdem erlaubt die spezielle Architektur von AgentSheets visuelle DSLs zu erstellen [RS95].

Es existiert auch eine dreidimensionale Variante von AgentSheets namens AgentCubes [RI06]. In dieser Version werden sämtliche Grafiken in 3D dargestellt. Zusätzlich können darin die Bewegungen von Kachel zu Kachel (bzw. dreidimensionalen Block zu Block) und Drehungen um 90 Grad animiert werden, was im Kontext von AgentCubes auch „Incremental Animation“ genannt wird.

Die oben genannten Sprachen (Scratch, AgentSheets etc.) wurden maßgeblich von einer Sprache namens Logo [BLL85] bzw. der Logo-Familie beeinflusst. Die Sprache ist eigentlich rein textuell, d. h., die meisten Interpreter bzw. GUIs bieten im Gegensatz zu Scratch oder AgentSheets keine grafischen Blöcke. Interessant ist Logo v. a. aufgrund seiner Besonderheit TurtleGraphics. Diese Erweiterung zeigt eine bzw. mehrere virtuelle Schildkröten innerhalb der Entwicklungsumgebung grafisch an. Durch verschiedene Logo-Anweisungen kann diese Schildkröte bewegt werden. Dabei kann die Schildkröte auch Linien hinter sich herziehen und dadurch zeichnen. Eine Variante von Logo mit visueller Programmierung heißt Turtle Blocks [Tur12].

Weitere Sprachen, die auf ähnlichen Ideen beruhen und an dieser Stelle erwähnt werden sollten, sind Karel the Robot [Pat94], Hands [Pan02], Visuit [HP03], ComiKit [KM07], Kodu [Sto10] oder Stagecast Creator [SCT00] (ursprünglich KidSim).

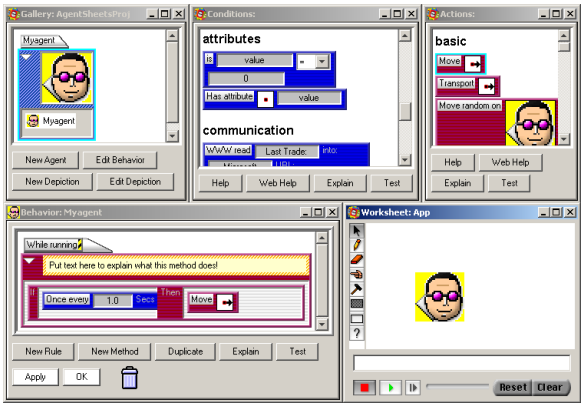


Abbildung 2.9: Screenshot – AgentSheets

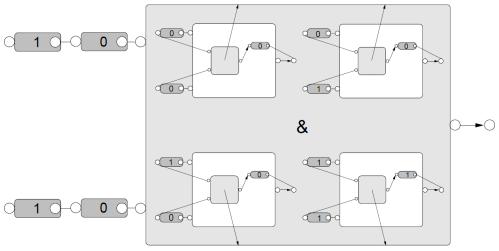


Abbildung 2.10: Pictorial Janus – Logisches UND mit Konstanten (aus [GM98])

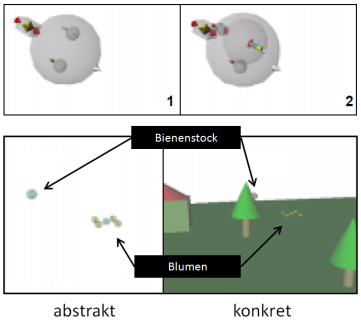


Abbildung 2.11: SAM – Animation und Sichten (teilweise aus [GMR98])

Pictorial Janus [KS90b] ist eine diagrammorientierte, visuelle Syntax für die textuelle, logische Programmiersprache Janus [KS90a]. Die Syntax wurde 1989 von Kahn, dem Autor von ToonTalk, und Saraswat entwickelt. Janus selbst ist eine constraintbasierte, parallele Programmiersprache. Damit basiert die Sprache im Kern auf den gleichen Ideen wie ToonTalk. Präsentationsform und Vorgehensweise beim Programmieren weisen jedoch gravierende Unterschiede auf, da Pictorial Janus lediglich die Programmausführung animiert. Die Programmierung in Pictorial Janus erfolgt ausschließlich mittels abstrakten Formen, die zweidimensional angeordnet werden müssen. In Abb. 2.10 wird bspw. ein sogenannter „Agent“ dargestellt, der die Funktionalität einer binären logischen UND-Funktion nachbildet. Diese Funktionalität wird erreicht, indem die im Agenten enthaltenen Muster mit den „außen anliegenden“ Elementen verglichen werden (Pattern-Matching), um danach das passende Muster auszusuchen

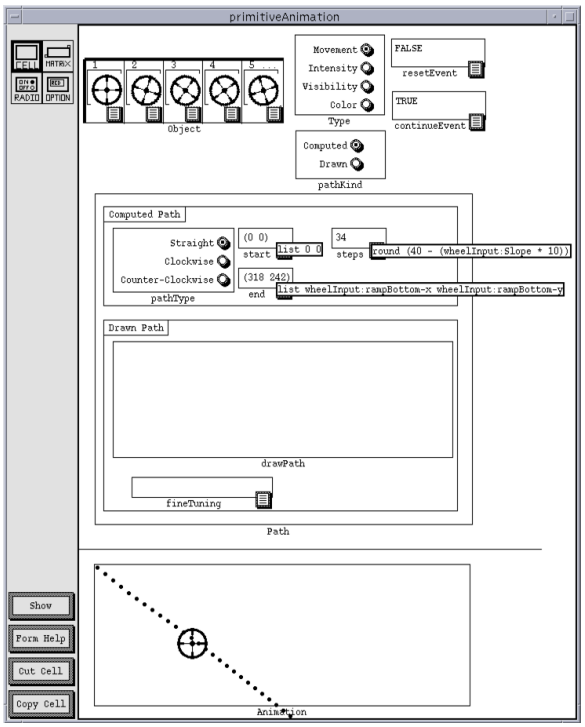


Abbildung 2.12: Screenshot – Forms/3 (aus [CBC96])

und anzuwenden, d. h., das Diagramm wird gemäß Muster angepasst.[3] Alle Bewegungen und Verformungen werden dabei animiert dargestellt.

In [GM98] wird ein Ansatz präsentiert, in dem die Ausführung und Animation eines Pictorial Janus-Programms in eine dreidimensionale Ansicht übertragen wird. In dieser Sicht kann sogar eine konkretere Darstellungsform für das Programm verwendet werden. Da die Programme in Pictorial Janus meist miteinander kommunizierende Agenten festlegen, ist deren Darstellung als dreidimensionale, interagierende Objekte der Zieldomäne naheliegend. Im Rahmen der genannten Arbeit und [GMR98] wird zusätzlich eine „animierte 3D-Programmiersprache“ namens SAM (Solid Agents in Motion) vorgestellt, die auf denselben Ideen wie Pictorial Janus basiert. Abb. 2.11 soll einige Eindrücke von der Sprache vermitteln.

Als letztes Beispiel soll Forms/3 genannt werden. Forms/3 ist eine deklarative, visuelle Programmiersprache [BA94], in der Formulare auf Basis von Zellen erstellt werden. Die Funktionalität der Zellen orientiert sich dabei an den Zellen typischer Tabellenkalkulationsprogramme, in denen bspw. Konstanten oder Berechnungsvorschriften hinterlegt werden können. Interessant ist Forms/3 an dieser

[3] Unter `ftp://ftp.parc.xerox.com/pub/PictorialJanus` und `http://www.youtube.com/user/nubarp/` können einige animierte Beispiele gefunden werden.

Stelle aufgrund der Möglichkeit, die Sprache durch Animationen zu erweitern. In [CBC96] wird beschrieben, wie Algorithmenanimation in der Sprache und in den entsprechenden Formularen integriert werden kann. Abb. 2.12 zeigt, wie eine Animation entlang eines Pfades definiert wird: ein Wagenrad soll einen Hang hinab rollen.

Überblick über die vorgestellten Sprachen

Der Überblick in Tab. 2.2 enthält viele der in diesem Abschnitt betrachteten Sprachen, wobei die in der Tabelle dargestellten Spracheigenschaften natürlich nur eine stark vereinfachte Sicht auf die Sprachen bieten können. Im unteren Teil werden außerdem die wiederkehrenden Beispiele dieser Arbeit aufgelistet, die im folgenden Abschnitt behandelt werden. Dadurch ist ein Vergleich mit den bisher betrachteten Sprachen möglich.

Die Tabelle zeigt u. a. den Zweck der Sprachen. Die Möglichkeiten reichen hier von Sprachen zur Softwareentwicklung (Programmierung, Modellierung, Editoren für GUI etc.) über Sprachen, welche die entwickelte Software selbst repräsentieren, bis hin zu Sprachen, die lediglich der Visualisierung von Daten dienen. Als animierbar werden hierbei Sprachen bezeichnet, wie in Abschnitt 2.1, S. 13ff, erläutert. Der Begriff „inhärent“ wird in der Tabelle verwendet, um zu verdeutlichen, dass Animation sogar Bestandteil der Sprachspezifikation ist. Zuletzt wird noch angegeben, ob der Benutzer eines entsprechenden Editors interaktiv in die Animation bzw. den Ablauf eingreifen kann. Auch hier werden die Begriffe aus Abschnitt 2.1, S. 14, verwendet.

2.3 Wiederkehrende Beispielsprachen

Die vorangehenden Abschnitte haben vor allem gezeigt, dass animierte Sprachen auf sehr unterschiedlichen Ideen basieren und vielfältige Konzepte umgesetzt werden können. Dementsprechend wurden im Vorfeld dieser Arbeit zahlreiche Konzepte untersucht und entsprechende Editoren umgesetzt. Als wiederkehrende Beispiele dieser Arbeit und zur Stützung der präsentierten Ansätze kann allerdings nur eine kleine Untermenge der implementierten Sprachen und Fallstudien vorgestellt werden. Es mussten daher Sprachen gewählt werden, in denen wichtige Sachverhalte kompakt darstellbar sind. Gleichzeitig sollen durch die gewählten Sprachen aber die wichtigsten Aspekte animierter Sprachen repräsentiert werden.

Aus diesem Grund fiel die Wahl auf die folgenden drei animierten oder animierbaren (echt) visuellen Sprachen. Die Sprache der Bedingungs-/Ereignis-Netze (Abschnitt 2.3.1) wurde gewählt, da sie eine präzise Ausführungssemantik besitzt und somit ein klassisches Beispiel darstellt. Entsprechende Editoren bieten oft die Möglichkeit zur Simulation und Animation. Deren Umsetzung wird in anderen Artikeln beschrieben, wodurch ein Vergleich mit dem hier präsentierten Ansatz ermöglicht wird. Auch die Sprache Alligator Eggs (Abschnitt 2.3.2) hat dank des Lambda-Kalküls eine präzise Ausführungssemantik. Gleichzeitig ist für die animierte Darstellung selbst ein exakter – und im Vergleich zur Sprache

Sprache	**Sprachzweck**	**Darstellungsform**	**echt visuell**	**animiert**	**interaktiv**
AgentSheets – Visual AgenTalk	Programmierung	grafische Textblöcke	nein	nein	-
AgentSheets – Worksheet	Programmausführung, 2D-Editor	virtuelle Welt (2D)	ja	animierbar	teilweise
Alice – Programmierung	Programmierung	grafische Textblöcke	nein	nein	-
Alice – 3D-Darstellung	Programmausführung, 3D-Editor	virtuelle Welt (3D)	ja	inhärent	vollständig
Alternate Reality Kit	Programmausführung, Programmierung	virtuelle Welt (2D)	ja	inhärent	vollständig
LabView – Blockdiagramm (G)	Programmierung, Ablaufvisualisierung	diagrammorientiert	ja	animierbar	nein
LabView – Frontpanel	Programmausführung, GUI-Designer	Steuerelemente	ja	inhärent	vollständig
Logo – Programmierung	Programmierung	Code	nein	nein	-
Logo – TurtleGraphics	Ablaufvisualisierung	virtuelle Welt (2D)	ja	animierbar	nein
Machinations	Modellierung, Ablaufvisualisierung	diagrammorientiert	ja	inhärent	vollständig
Ruru	Programmierung, Ablaufvisualisierung	diagrammorientiert	ja	inhärent	vollständig
Scratch – Programmierung	Programmierung	grafische Textblöcke	nein	nein	-
Scratch – Bühne	Programmausführung, 2D-Editor	virtuelle Welt (2D)	ja	inhärent	vollständig
ToonTalk	Programmausführung, Programmierung	virtuelle Welt (2,5D)	ja	inhärent	vollständig
AVL-Bäume (Algorithmenanimation)	Modellierung, Ablaufvisualisierung	diagrammorientiert	ja	animierbar	teilweise
Klassendiagramm (UML)	Modellierung	diagrammorientiert	ja	nein	-
Zustandsdiagramm (UML)	Modellierung	diagrammorientiert	ja	animierbar	nein
Alligator Eggs	Programmierung, Ablaufvisualisierung	diagrammorientiert	ja	inhärent	nein
Avalanche	Programmausführung, Level-Editor	diagrammorientiert	ja	animierbar	vollständig
B/E-Netze	Programmierung, Ablaufvisualisierung	diagrammorientiert	ja	animierbar	teilweise
Traffic – Modellsicht	Modellierung	diagrammorientiert	ja	-	-
Traffic – Animationssicht	Ablaufvisualisierung	diagrammorientiert	ja	inhärent	vollständig
ToonProg	Programmierung	virtuelle Welt (2D)	ja	inhärent	vollständig

Tabelle 2.2: Beispiele animierter Sprachen

der B/E-Netze komplexerer – Animationsverlauf festgelegt. Um den Aspekt von Interaktionsmöglichkeiten während der Animation zu berücksichtigen, wurde zuletzt auch die Sprache Avalanche (Abschnitt 2.3.3) aufgenommen. Für die genannten Sprachen befinden sich im Anhang A die vollständigen Diagramme, die zur Implementierung der Editoren genutzt wurden und deren Teile im weiteren Verlauf mittels Beispielen auch beschrieben werden. Weitere Sprachen und untersuchte Aspekte, die in dieser Arbeit aber weniger sind, werden noch einmal kurz im letzten Unterabschnitt aufgegriffen (Abschnitt 2.3.4). Die darin beschriebenen Sprachen Traffic und ToonProg bieten neben erwähnenswerten Details im Grunde dieselben Merkmale wie die anderen Sprachen. Sie werden daher nur selten in den späteren Kapiteln als Beispiele aufgegriffen.

Einige der vorgestellten Sprachen können in animierter Form einschließlich Kommentaren zu verwendeten Techniken und den zugehörigen Editoren unter `http://www.youtube.com/user/diametaanimated/` als Video betrachtet werden.

2.3.1 Bedingungs-/Ereignis-Netze

Als erste Beispielsprache wurde die Sprache der Bedingungs-/Ereignis-Netze (B/E-Netze) gewählt. Sie gehört zu den Sprachen der Petri-Netze, deren Entwicklung maßgeblich in den 1960er Jahren von Carl Adam Petri vorangetrieben wurde. Petri-Netze gelten als wichtige Vertreter visueller Sprachen, die auch formal exakt definiert sind. Sie dienen meist zur Modellierung verteilter, diskreter Systeme, können aber auch in anderen Fällen Verwendung finden, z. B. zur Modellierung nebenläufiger Systeme etc. Einige wichtige Konzepte von Petri-Netzen finden sich auch in späteren Entwicklungen wie UML-Aktivitätsdiagramme, der Business Process Model and Notation (BPMN) oder auch Machinations (vgl. Abschnitt 2.2, S. 25) wieder.

Die Analyse der Umsetzbarkeit von animierten Editoren für Petri-Netze ist im Rahmen dieser Arbeit aus zwei Gründen wichtig. Zum einen wird die Sprache der Petri-Netze in zahlreichen anderen Arbeiten aufgegriffen, in denen ebenfalls die Generierung von Editoren aus einer Sprachspezifikationen beschrieben wird. Beispiele hierfür sind [Min04, KC09, Erm06, PR08, Cra10]. Einige dieser Arbeiten gehen dabei auch auf Spezifikation von Dynamik und Animation ein (siehe Abschnitt 4.4, S. 113), wodurch ein Vergleich mit dem in dieser Arbeit präsentierten Ansatz möglich ist. Andererseits basieren viele animierte Sprachen auf den Ideen von Petri-Netzen und deren Animationen, z. B. die Visualisierung der Schaltvorgänge. Auch Machinations ist bspw. ein Editor für Diagramme mit ähnlichem Aufbau und Animationsabläufen. Unter der Annahme, dass sich Ansätze, mit denen Petri-Netz-Editoren auf einfache Weise generiert werden können, auch für Editoren ähnlicher Sprachen eignen, sind Petri-Netze daher ein wichtiges Beispiel.

In diesem Abschnitt sollen Petri-Netze und B/E-Netze zunächst kurz beschrieben werden. Danach wird gezeigt, wie die Sprache innerhalb eines animierten Editors umgesetzt werden kann.

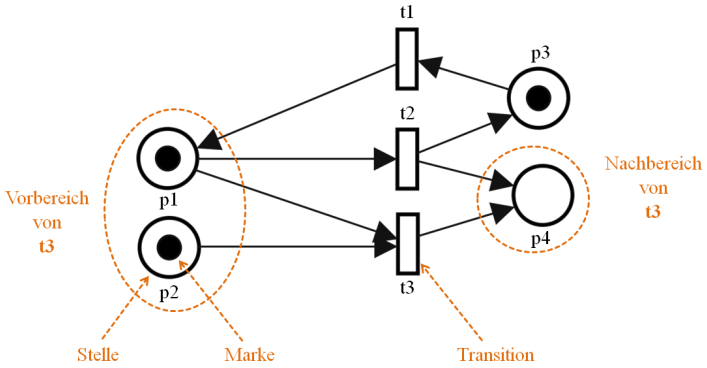


Abbildung 2.13: Einfaches Petri-Netz mit Anmerkungen

B/E-Netze - Formale Definition und Beschreibung

B/E-Netze sind eine der zahlreichen Varianten von Petri-Netzen. Daher folgt eine kurze Definition der allgemeinen Form von Petri-Netzen mit Stellenkapazität und Kantengewichtung (häufig auch Stellen-Transitions-Netze genannt), die in entsprechender Literatur häufig beschrieben wird. Weiterführende Details über Petri-Netze können bspw. in [Rei85] nachgelesen werden.

Definition 2.1 (Petri-Netz)
Ein *Petri-Netz* mit Stellenkapazität und Kantengewichtung ist ein 6-Tupel $PN = (S, T, F, K, V, m_0)$ mit folgenden Komponenten:

- S ist die nichtleere endliche Menge der *Stellen*.
- T ist die nichtleere endliche (zu S disjunkte) Menge der *Transitionen*.
- $F \subseteq (S \times T) \cup (T \times S)$ ist die nichtleere endliche Menge der *Kanten*.
- $K : S \to \mathbb{N}^+ \cup \{\infty\}$ ist die *Stellenkapazität*, die jeder Stelle $s \in S$ eine positive ganze Zahl oder den Wert ∞ zuordnet.
- $V : F \to \mathbb{N}^+$ ist die *Kantengewichtung*, die jeder Kante $f \in F$ eine positive ganze Zahl zuordnet.
- m_0 ist die *(Anfangs-)Markierung* von S, d. h. eine Funktion $m : S \to \mathbb{N}_0$, die jeder Stelle eine natürliche Zahl zuordnet, wobei für alle $s \in S$ gelten muss $m_0(s) \leq K(s)$. △

Abb. 2.13 zeigt ein solches Petri-Netz grafisch. Es ist gleichzeitig ein Beispiel, wie Petri-Netze häufig visuell notiert werden, d. h., es zeigt eine konkrete Syntax für Petri-Netze. Die Stellen werden dabei als Kreise und die Transitionen als Rechtecke dargestellt. Die Markierung wird durch Punkte bzw. Marken (engl. *tokens*) innerhalb der Stellen dargestellt. Stellenkapazität und Kantengewicht werden durch kleine Zahlen an den Elementen notiert, wobei Kapazität bzw.

Gewicht mit Betrag 1 nicht dargestellt werden. Die Anzahl der Punkte innerhalb einer Stelle entspricht der natürlichen Zahl, die der Stelle im Rahmen ihrer Markierung zugeordnet wird.

Das dargestellte Petri-Netz lässt sich textuell auch wie folgt beschreiben:

- $S = \{s1, s2, s3, s4\}$
- $T = \{t1, t2, t3\}$
- $F = \{(p1, t2), (p1, t3), (p2, t3), (p3, t1), (t1, p1), (t2, p3), (t2, p4), (t3, p4)\}$
- $\forall s \in S : K(s) = 1$
- $\forall f \in F : V(f) = 1$
- $m_0 = \{(p1,1), (p2,1), (p3,1), (p4,0)\}$

Auf Petri-Netzen kann zudem eine Ausführungssemantik definiert werden. Hierbei wird festgelegt, wie die Markierung in einem Petri-Netz verändert wird, wenn eine Transition *schaltet*, wobei lediglich *aktive* Transitionen schalten dürfen. Schalten aktive Transitionen automatisch und können dabei mehrere Transitionen gleichzeitig aktiv sein, so erfolgt der Ablauf nichtdeterministisch.

Folgende Definitionen beschreiben, wann eine Transition aktiv ist, und wie die Markierung bei einem Schaltvorgang verändert wird:

Definition 2.2 (Vor- und Nachbereich einer Transition)
Gegeben sei ein Petri-Netz PN. Für jede Transition $t \in T$ seien folgende Stellenmengen definiert:

- $\bullet t = \{s \mid (s,t) \in F\}$ heißt Vorbereich von t.
- $t \bullet = \{s \mid (t,s) \in F\}$ heißt Nachbereich von t. △

In Abb. 2.13 werden Vor- und Nachbereich von *t3* beispielhaft dargestellt.

Definition 2.3 (Schaltvorgang einer Transition)
Gegeben seien ein Petri-Netz PN und seine aktuelle Markierung $m : S \to \mathbb{N}_0$. Eine Transition $t \in T$ ist *aktiv*, wenn für alle Stellen $s \in \bullet t$ gilt $m(s) \geq V(s,t)$, für alle Stellen $s \in t\bullet \setminus \bullet t$ gilt $m(s) + V(t,s) \leq K(s)$ und für alle $s \in t\bullet \cap \bullet t$ gilt $m(s) - V(s,t) + V(t,s) \leq K(s)$. Sobald eine aktive Transition *schaltet*, wird dem Petri-Netz eine Nachfolgemarkierung m' zugeordnet, so dass für alle $s \in S$ gilt

$$m'(s) = \begin{cases} m(s) - V(s,t) & \text{, falls } s \in \bullet t \setminus t\bullet \\ m(s) + V(t,s) & \text{, falls } s \in t\bullet \setminus \bullet t \\ m(s) - V(s,t) + V(t,s) & \text{, falls } s \in t\bullet \cap \bullet t \\ m(s) & \text{, sonst} \end{cases} . \qquad \triangle$$

Wie eingangs erwähnt, soll als wiederkehrendes Beispiel dieser Arbeit eine eingeschränkte Klasse von Petri-Netzen namens B/E-Netze verwendet werden. In einem solchen Netz werden Stellen auch *Bedingungen* und Transitionen auch *Ereignisse* genannt. Die Einschränkungen sind dabei wie folgt festgelegt:

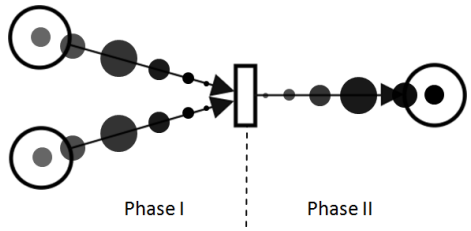


Abbildung 2.14: Animationsphasen in B/E-Netzen

Definition 2.4 (Bedingungs-/Ereignis-Netz)
Ein *Bedingungs-/Ereignis-Netz* (engl. condition/event net) ist ein Petri-Netz, wobei

- $V(f) = 1$ für jede Kante $f \in F$,
  d.h., das Gewicht aller Kanten beträgt 1,

- $K(s) = 1$ für jede Stelle $s \in S$,
  d.h., die Kapazität aller Stellen beträgt 1. △

Das Petri-Netz in Abb. 2.13 ist ein solches B/E-Netz. Darin ist Transition *t3* aktiv, da jede Stelle des Vorbereichs (*p1*, *p2*) jeweils eine Marke und keine Stelle des Nachbereichs (*p4*) eine Marke enthält. Die Transitionen *t1* und *t2* sind hingegen nicht aktiv, da es jeweils eine Stelle im Nachbereich gibt (*p1* bzw. *p3*), die aufgrund der beschränkten Kapazität keine weitere Marke mehr aufnehmen kann.

B/E-Netze - Animation

Bei der Simulation eines B/E-Netzes wird der zuletzt beschriebene Ablauf von Schaltvorgängen (engl. auch „token game“) oftmals grafisch dargestellt. Für den Zuschauer ist es manchmal allerdings schwierig, entsprechende Änderungen zu verfolgen, wenn jeder Schaltvorgang nur durch Vorher/Nachher-Zustände visualisiert wird. Das Verfolgen von Änderungen wird insbesondere dadurch erschwert, dass eine beliebige Anzahl von Marken vom Vorbereich entfernt und im Nachbereich wieder hinzugefügt werden können.

Dieses Problem soll dadurch abgeschwächt werden, indem ein Schaltvorgang animiert dargestellt wird. Die Animation läuft dabei in zwei Phasen ab (siehe Abb. 2.14). Zunächst sollen sich die Marken von den Stellen des Vorbereichs zur schaltenden Transition bewegen. Danach bewegen sich „neue“ Marken[4] zu den Stellen des Nachbereichs. Die Bewegung soll dabei synchron erfolgen, d. h., alle Marken kommen gleichzeitig am Ziel an, obwohl der notwendige Weg unterschiedlich lang sein kann. Außerdem sollen die Marken während ihrer

[4]Der Begriff „neu“ wird verwendet, da sich die Anzahl der Stellen des Nachbereich von der Anzahl der Stellen des Vorbereichs unterscheiden kann und daher evtl. mehr bzw. weniger Marken benötigt werden.

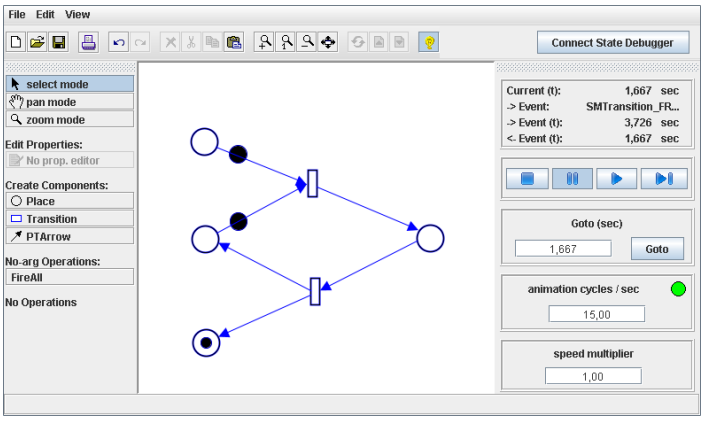


Abbildung 2.15: Screenshot – B/E-Netz-Editor

Bewegung kurzzeitig vergrößert werden, um als Blickfang für den Schaltvorgang zu dienen.

Abb. 2.15 zeigt den Editor, der im Rahmen dieser Arbeit entstanden ist. Darin können B/E-Netze inkl. Anfangsmarkierung erstellt werden. Danach kann das B/E-Netz simuliert bzw. animiert werden. Der Benutzer kann dabei aktive Transitionen wählen, um einen Schaltvorgang auszulösen oder die Schaltvorgänge vollständig automatisiert ablaufen zu lassen.

2.3.2 Alligator Eggs

Als nächste Beispielsprache wurde eine visuelle Sprache gewählt, die auf dem untypisierten Lambda-Kalkül von Church und Kleene [Chu36] basiert. Das Kalkül ist ein formales System, das benutzt werden kann, um bestimmte Aspekte von Berechenbarkeit, Rekursion, funktionale Programmierung etc. zu erforschen.

Für das Kalkül existieren neben der häufig genutzten textuellen Form mit Symbolen, Variablen und Klammern auch zahlreiche grafische Darstellungsformen. Letztere sollen das Kalkül und dessen Regeln zugänglicher und häufig auch spielerisch präsentieren. Eine dieser Darstellungsformen heißt Alligator Eggs von Victor [Vic08]. Sie ermöglicht es, Ausdrücke des untypisierten Lambda-Kalküls mit farbigen Alligatoren und Alligatoreiern darzustellen. Die anspruchsvollere animierte Darstellung von fressenden Alligatoren, wackelnden Eiern etc. ist auch einer der Gründe, warum Alligator Eggs in Rahmen dieser Arbeit beschrieben wird.

In diesem Abschnitt wird zunächst das untypisierte Lambda-Kalkül beschrieben. Danach wird die darauf aufbauende Sprache Alligator Eggs vorgestellt, und es wird erläutert, wie ein entsprechender Editor für Alligator Eggs umgesetzt wurde.

Untypisiertes Lambda-Kalkül – Definition und Beschreibung

Zur Beschreibung des Lambda-Kalküls wird im Folgenden zunächst angegeben, wie wohlgeformte Ausdrücke des Kalküls aufgebaut sind und wie diese textuell dargestellt werden können. Danach zeigen Transformationsregeln, wie solche Ausdrücke umgeformt werden können. Als weiterführende Literatur diesbezüglich sei [Bar84] erwähnt.

Definition 2.5 (Lambda-Ausdrücke)
Ein *Lambda-Ausdruck* ist entweder

- eine *Variable*, geschrieben x,
- eine *Applikation*, geschrieben FG, wobei F und G jeweils Lambda-Ausdrücke sind, oder
- eine *Abstraktion*, geschrieben $\lambda x.F$, wobei x eine Variable und F ein Lambda-Ausdruck ist. △

In der konkreten, textuellen Syntax für Lambda-Ausdrücke ist außerdem die Verwendung von Klammern erlaubt und in manchen Fällen auch notwendig, um den Aufbau des Ausdrucks eindeutig darstellen zu können, z. B. im Falle von $(\lambda x.(\lambda y.y)x)$. Äußere Klammerung kann jedoch immer weggelassen werden. Es kann also FG anstelle von (FG) geschrieben werden.

Definition 2.6 (Freie Variable)
Eine Variable y ist *frei vorkommend* in einem Lambda-Ausdruck E, wenn

- E eine Variable x ist und $x = y$ gilt,
- E eine Applikation FG ist und y in F und G frei vorkommt, oder
- E eine Abstraktion $\lambda x.F$ ist, wobei $x \neq y$, und y frei in F vorkommt. △

Auf der Menge der Lambda-Ausdrücke werden Transformationsregeln definiert, mit denen diese umgeformt werden können. Manchmal werden diese Regeln auch Kongruenzregeln genannt, da ursprünglicher und transformierter Ausdruck semantisch äquivalent sind. Zu den wichtigsten Regeln gehören die α-Konversion und die β-Konversion. Für deren exakte Definition wird die *Ersetzungsfunktion* benötigt.

Definition 2.7 (Ersetzungsfunktion)
Die Ersetzungsfunktion $X[x := B]$ ist induktiv definiert durch

$$
\begin{aligned}
x[x := B] &= B, \\
y[x := B] &= y \text{ , falls } x \neq y, \\
FG[x := B] &= F[x := B]G[x := B], \\
(\lambda x.F)[x := B] &= \lambda x.F, \\
(\lambda y.F)[x := B] &= \lambda y.(F[x := B]) \text{ , falls } x \neq y \text{ und} \\
&\qquad y \text{ keine frei vorkommende Variable in } B \text{ ist}, \\
(\lambda y.F)[x := B] &= \lambda z.((F[y := z])[x := B]) \text{ , falls } x \neq y \text{ und} \\
&\qquad y \text{ eine frei vorkommende Variable in } B \text{ ist}, \\
&\qquad \text{wobei } z \text{ eine neue, sonst nirgends} \\
&\qquad \text{vorkommende Variable ist.}
\end{aligned}
$$

△

Definition 2.8 (α-Konversion)
Wird auf einen Lambda-Ausdruck der Form $\lambda x.F$ eine *α-Konversion* angewendet, so wird daraus ein Lambda-Ausdruck der Form $\lambda y.F'$, wobei $F' = F[x := y]$ und y in F nicht frei vorkommen darf. △

Ein einfaches Beispiel einer α-Konversion ist die Umformung von $\lambda x.yx$ in $\lambda z.yz$. Beide Ausdrücke sind semantisch äquivalent. Die Transformationsregel verdeutlicht also, dass die in Lambda-Ausdrücken verwendeten Namen von gebundenen Variablen jederzeit durch beliebige andere Namen (unter Achtung der in den Definitionen festgelegten Regeln) ersetzt werden dürfen, ohne dass sich die Semantik des Ausdrucks ändert.

Definition 2.9 (β-Konversion)
Wird auf einen Lambda-Ausdruck der Form $(\lambda x.F)G$ eine *β-Konversion* angewendet, so wird daraus ein Lambda-Ausdruck der Form F', wobei $F' = F[x := G]$.△

Häufig wird in diesem Zusammenhang davon gesprochen, dass ein Lambda-Ausdruck der Form $(\lambda x.F)$ eine (anonyme) Funktion repräsentiert. Bei einer β-Konversion (manchmal auch β-Reduktion) von Ausdruck $(\lambda x.F)G$ spricht man daher auch von Funktionsanwendung, bei der G auf die Funktion angewendet und entsprechende Vorkommnisse der Variablen x in F ersetzt werden. Wichtig ist dabei auch, dass Funktionsanwendungen in Lambda-Ausdrücken immer linksassoziativ sind, d. h. $EFG = (EF)G$.

Beispiel 2.2 (Lambda-Kalkül – Beispielreduktion)
In folgendem Beispiel werden einige α- und β-Konversionen durchgeführt. Die Anwendung der Regeln soll dabei der Auswertung eines booleschen Terms entsprechen. Hierfür wird die sogenannte Church-Kodierung (benannt nach dem Erfinder des Lambda-Kalküls, der diese Kodierung ebenfalls einsetzte) verwendet, um $wahr$, $falsch$ und boolesche Funktionen als Lambda-Ausdrücke darzustellen. Abb. 2.16 (oben) zeigt die verwendete Kodierung.

Als Beispiel soll der Term $\overline{falsch} \wedge wahr$ ausgewertet werden. Wird dieser Term mittels Church-Kodierung als Lambda-Ausdruck dargestellt, muss die Präfixnotation verwendet werden: *und nicht falsch wahr*. Die Berechnung des Ergebnisses kann danach durch α- und β-Konversionen durchgeführt werden, wobei in folgender Darstellung an manchen Stellen mehrere α-Konversionen in einem Schritt zusammengefasst wurden:

$$
\begin{aligned}
& \mathit{und}\ \mathit{nicht}\ f\mathit{alsch}\ \mathit{wahr} \\
= \ & (\lambda x.\lambda y.xyx)((\lambda x.\lambda y.\lambda z.xzy)(\lambda x.\lambda y.y))(\lambda x.\lambda y.x) && (2.3.2.1)\\
\to_\alpha \ & (\lambda p.\lambda q.pqp)((\lambda x.\lambda y.\lambda z.xzy)(\lambda x.\lambda y.y))(\lambda x.\lambda y.x) && (2.3.2.2)\\
\to_\beta \ & (\lambda q.((\lambda x.\lambda y.\lambda z.xzy)(\lambda x.\lambda y.y))q((\lambda x.\lambda y.\lambda z.xzy)(\lambda x.\lambda y.y)))(\lambda x.\lambda y.x) && (2.3.2.3)\\
\to_\alpha \ & (\lambda q.((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r))q((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r)))(\lambda x.\lambda y.x) && (2.3.2.4)\\
\to_\beta \ & ((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r))(\lambda x.\lambda y.x)((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r)) \\
\to_\alpha \ & ((\lambda x.\lambda y.\lambda z.xzy)(\lambda p.\lambda r.r))(\lambda x.\lambda y.x)((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r)) \\
\to_\beta \ & (\lambda y.\lambda z.(\lambda p.\lambda r.r)zy)(\lambda x.\lambda y.x)((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r)) \\
\to_\alpha \ & (\lambda q.\lambda z.(\lambda p.\lambda r.r)zq)(\lambda x.\lambda y.x)((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r)) \\
\to_\beta \ & (\lambda z.(\lambda p.\lambda r.r)z(\lambda x.\lambda y.x))((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r)) \\
\to_\alpha \ & (\lambda q.(\lambda s.\lambda t.t)q(\lambda x.\lambda y.x))((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r)) \\
\to_\beta \ & (\lambda s.\lambda t.t)((\lambda p.\lambda r.\lambda z.pzr)(\lambda p.\lambda r.r))(\lambda x.\lambda y.x) \\
\to_\beta \ & (\lambda t.t)(\lambda x.\lambda y.x) \\
\to_\beta \ & (\lambda x.\lambda y.x) \\
= \ & wahr && \triangle
\end{aligned}
$$

Alligator Eggs – Beschreibung

Um das Lambda-Kalkül und seine Konzepte für Anfänger zugänglicher zu gestalten, wurden bereits einige visuelle Notationen vorgeschlagen, bspw. Visual Expressions (VEX) [CHZ95]. Auch Alligator Eggs [Vic08] ist ein solcher Beitrag und gehört sicherlich zu den verspielteren Varianten. Tab. 2.3 zeigt die darin verwendeten Elemente. Die gezeigten Komponenten (Alligatoren und Eier) werden diagrammorientiert notiert und repräsentieren auf diese Art und Weise Lambda-Ausdrücke.

Alligator Eggs			Lambda-Kalkül	
Alligatorei		=	Variable (Farbe des Eis entspricht Bezeichner)	a
Alligator		=	Abstraktion (Alligatorfarbe entspricht Bezeichner)	$\lambda a.\ldots$
Alter Alligator		=	Klammern	$(\ldots)$

Tabelle 2.3: Alligator Eggs – Elemente

Alligatoreier werden als Variablen verwendet, wobei die Farbe eines solchen Eis dem Bezeichner der Variable entspricht, z. B. ein rotes Ei für a, ein grünes Ei

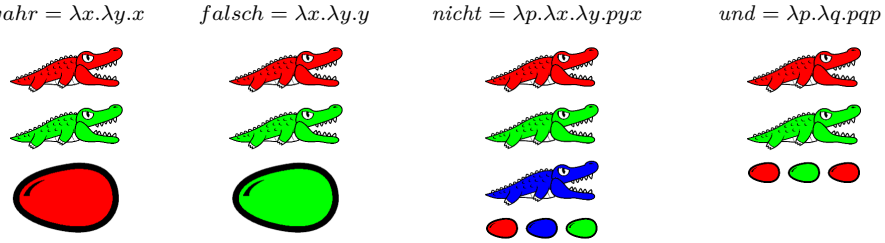


Abbildung 2.16: Boolesche Werte repräsentiert durch Alligator-Familien

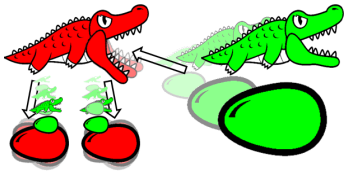


Abbildung 2.17: Alligator Eggs – Fressregel

für b etc. *Hungrige Alligatoren* entsprechen einer Abstraktion, wobei auch hier die Farbe des Alligators eine Rolle spielt. Tab. 2.4 zeigt, wie eine vollständige Abstraktion aufgebaut sein muss: wird ein (roter) Alligator über andere Elemente F gesetzt, so entspricht dies dem Lambda-Ausdruck $\lambda a.F$. Man spricht hierbei auch davon, dass $\lambda a.F$ eine *(Alligator-)Familie* ist und der Alligator die unter ihm stehenden Elemente *beschützt*. Eine Applikation FG wird in Alligator Eggs dargestellt, indem die entsprechenden Repräsentationen von F und G nebeneinander gezeichnet werden, wobei die beiden Gruppen jeweils an der oberen Kante ausgerichtet sein müssen.

Im Gegensatz zu [Vic08] wird allerdings nicht verlangt, dass Eier immer von Alligatoren beschützt werden und auch nur Farben der beschützenden Alligatoren haben dürfen. In Bezug auf Lambda-Ausdrücke würde dies bedeuten, dass es keine frei vorkommenden Variablen geben darf.

Schließlich gibt es noch *alte Alligatoren*. Diese entsprechen der Klammerung in der textuellen Syntax des Lambda-Kalküls. Indem ein alter Alligator über bestimmte Elemente gesetzt wird, werden diese Elemente „gruppiert“, d. h., es würden Klammern um die textuellen Entsprechungen dieser Elemente gesetzt werden. Auch in diesem Fall wird von einer Familie gesprochen.

Als Beispiele können an dieser Stelle noch einmal die Konstrukte der Church-Kodierung für Elemente der booleschen Algebra dienen. Abb. 2.16 zeigt neben der textuellen Repräsentation auch die entsprechende Form in Alligator Eggs.

Die Transformationsregeln des Lambda-Kalküls können ebenfalls auf Alligator Eggs übertragen werden. Victor hat hierfür drei Regeln vorgesehen: die Fressregel (engl. eating rule), die Farbregel (engl. color rule) und die Regel für alte Alligatoren.

Die *Fressregel* besagt, dass (hungrige) Alligatoren ein nebenstehendes Element fressen können (siehe Abb. 2.17). Ist dieses Element ein Alligator (alt oder

Alligator Eggs			Lambda-Kalkül
Abstraktion		=	$\lambda a.F$
Applikation		=	FG
Klammerung		=	(F)

Tabelle 2.4: Alligator Eggs – Layout

hungrig), so wird auch die ganze Familie gefressen. Die gefressenen Elemente werden anschließend aus dem Diagramm gelöscht, genau wie der fressende Alligator. Die gefressenen Elemente werden aber „wiedergeboren“. An den Stellen von Alligatoreiern, die zur Familie des fressenden Alligators gehören und zusätzlich dieselbe Farbe haben, werden Kopien der gefressenen Elemente erzeugt. Das ursprüngliche Alligatorei wird aus dem Diagramm gelöscht. Somit entspricht diese Regel der β-Konversion. Bei der Anwendung der Fressregel muss allerdings darauf geachtet werden, dass die Lambda-Ausdrücke linksassoziativ sind, d. h., die Regelanwendung beginnt immer bei Alligatoren, die in einer Reihe ganz links stehen.

Wichtig ist ebenfalls die Einhaltung der *Farbregel*. Diese legt fest, dass Farben innerhalb von Alligatorfamilien geändert werden müssen, falls ein Alligator eine Familie frisst und in der gefressenen Familie eine Farbe vorhanden ist, die auch in der Familie des fressenden Alligators vorkommt. In diesem Fall muss eine solche Farbe durch eine noch nicht verwendete Farbe ersetzt werden. Diese Ersetzung repräsentiert die α-Konversion. In dieser Arbeit werden dabei immer die Farben der fressenden Familie ausgetauscht. Außerdem dürfen keine Eier neu eingefärbt werden, die nicht durch einen Alligator derselben Farbe beschützt werden (frei vorkommende Variablen, siehe oben). Die an dieser Stelle verwendete Farbregel unterscheidet sich also leicht von der in [Vic08] beschriebenen.

Schließlich gibt es die *Regel für alte Alligatoren*. Diese besagt, dass ein alter Alligator, der lediglich ein Element beschützt, das direkt unter ihm platziert wurde (weitere Elemente unterhalb des direkten „Kindes“ sind hierbei nicht relevant), nicht mehr benötigt wird und vom Diagramm entfernt werden kann.

Abb. 2.18 zeigt die ersten Schritte aus Beispiel 2.2 in der Sprache Alligator Eggs. Es wird zunächst die Farbregel angewendet (von 2.3.2.1 zu 2.3.2.2), dann die Fressregel (von 2.3.2.2 zu 2.3.2.3) und danach wieder die Farbregel (von 2.3.2.3 zu 2.3.2.4) etc.

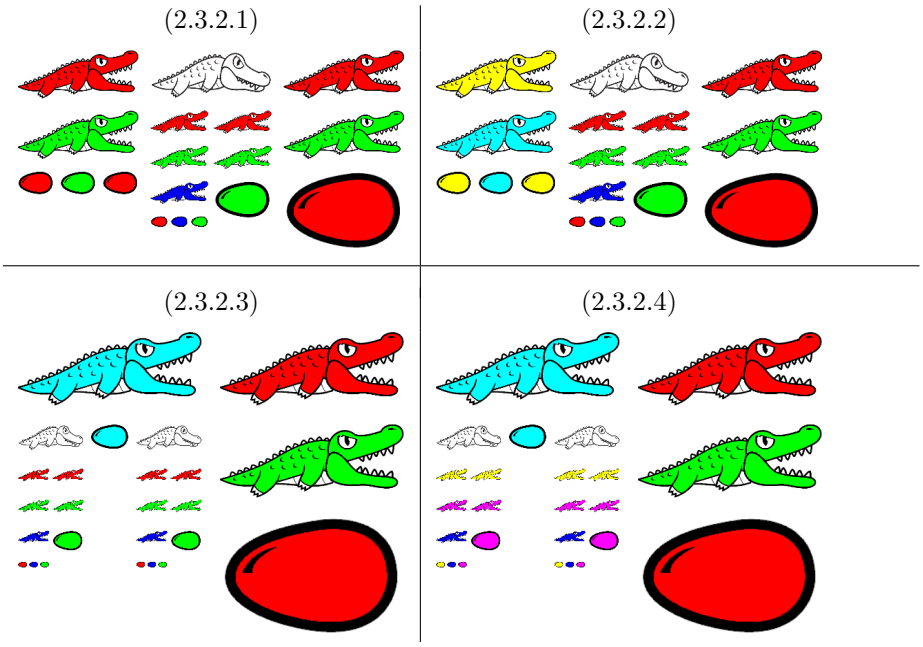


Abbildung 2.18: Anwendung von Fress- und Farbregel

Alligator Eggs – Animation

Aus vielen bereits genannten Gründen ist es naheliegend, Alligator Eggs und die Anwendung seiner Regeln animiert darzustellen. Ohne direkt auf die Beschreibung von Animationen einzugehen, ist die Anwendung der Regeln von Alligator Eggs in [Vic08] so metaphorisch beschrieben, dass sich folgende im Rahmen dieser Arbeit festgelegten Animationsvorgänge für Alligator Eggs fast automatisch ergeben haben.

Die Farbregel kann vergleichsweise einfach animiert werden. Anstatt die neuen Farben von Diagrammkomponenten unmittelbar umzuschalten, werden die Farben (von alter zu neuer Farbe) weich überblendet.

Die Fressregel beinhaltet mehrere Animationsphasen und betrifft unterschiedliche Elemente. Einzelne Phasen und Animationsvorgänge überlappen sich dabei zeitlich. Während der Phase des Fressvorgangs bewegen sich alle Opfer des hungrigen Alligators direkt auf dessen Maul zu, wobei sie gleichzeitig verkleinert werden (vgl. Abb. 2.17). Der Alligator öffnet und schließt währenddessen sein Maul, was das „Fressen“ darstellen soll. Danach werden alle Opfer entfernt, und es beginnt die Neugeburt, d. h., Kopien der Opfer bewegen sich auf die entsprechenden Eier des Alligators zu. Diesmal sollen sich die bewegenden Elemente vergrößern. Gleichzeitig sollen die Eier wackeln, was das „Ausschlüpfen“ andeuten soll. Um die Eier im Anschluss verschwinden zu lassen, wurde auch hier ein Schrumpfvorgang festgelegt.

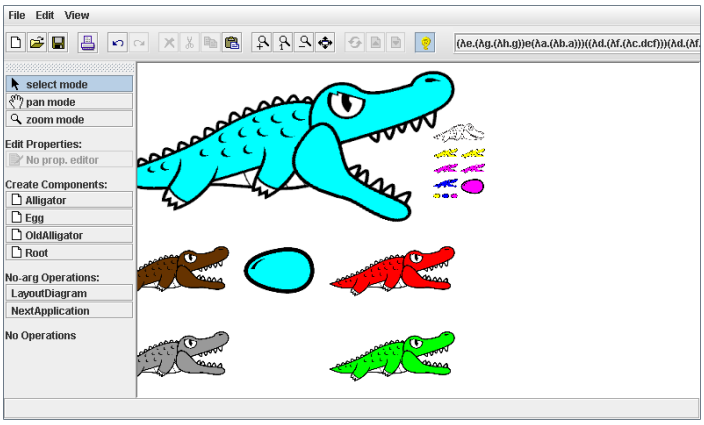


Abbildung 2.19: Screenshot - Alligator Eggs

Zusätzlich gibt es noch sterbende Alligatoren. Zum einen stirbt der hungrige Alligator gegen Ende des Fressvorgangs. Dies wird dargestellt durch einen hungrigen Alligator, der sich langsam auf seinen Rücken dreht, sein Maul schließt und ebenfalls schrumpft, um anschließend zu verschwinden. Zum anderen sterben auch alte Alligatoren, falls sie nicht mehr benötigt werden. Auch diese rotieren, bis sie auf ihren Rücken liegen und werden verkleinert.

Bei sämtlichen Vorgängen ist es wichtig, dass sich die Diagrammkomponenten nach Abschluss der Animationsphasen wieder an einer gültigen Position befinden, d. h., das Layout des Diagramms spiegelt die nach der Regelanwendung vorgeschriebene Semantik wieder (vgl. Tab. 2.4). Deshalb muss während der Animation auch ein neues Layout berechnet werden. Alle Komponenten des Diagramms bewegen sich dann – zeitgleich zu anderen Animationen – auf die berechneten Positionen zu.

Alligator Eggs – Editor

Abb. 2.19 zeigt den realisierten Editor für Alligator Eggs. Darin können zunächst Alligator Eggs-Diagramme gezeichnet werden. Daraufhin kann der damit erstellte Lambda-Ausdruck ausgewertet werden. Die Auswertung erfolgt dabei wie geschildert vollständig animiert. Weitere Details über Alligator Eggs und einer ersten Implementierung des Editors werden in [SM09] beschrieben.

2.3.3 Avalanche

Avalanche ist ursprünglich ein Brettspiel für zwei oder mehr Spieler. Das Spiel wurde bereits von einigen Spieleverlagen unter verschiedenen Namen und in unterschiedlichen Varianten veröffentlicht. An dieser Stelle soll allerdings nur die grundlegende Spielmechanik beleuchtet und umgesetzt werden. Weitere Aspekte wie Spielziel, Regeln für mehrere Spieler etc. werden im Folgenden ignoriert. Stattdessen wird eine Avalanche-Variante beschrieben, die als Beispiel für ein

einfaches dynamisches System dienen kann, dessen Struktur zunächst per Editor aufgebaut werden muss. Im gleichen Editor kann die Spielmechanik anschließend getestet werden. Dabei reagiert der Editor auf Benutzereingaben zu jedem Zeitpunkt, auch während eine Animation aktiv ist. Avalanche ist in dieser Arbeit somit das wichtigste Beispiel eines Editors für eine interaktive animierte Sprache.

Avalanche – Beschreibung

Das Spielbrett von Avalanche wird schräg aufgestellt, so dass es eine schiefe Ebene bildet (siehe Abb. 2.20) und die auf das Spielbrett gesetzten Murmeln nach unten rollen. Das Brett selbst besteht aus mehreren vertikal ausgerichteten Bahnen, an deren oberen Enden die Murmeln hineingesetzt werden können (a). Eine solche Bahn kann außerdem durch Weichen unterbrochen werden. Diese werden in der Mitte zwischen zwei benachbarten Bahnen am Spielbrett angebracht. An einer solchen Stelle werden die beiden Bahnen nicht durch eine Trennwand voneinander abgegrenzt. Innerhalb einer Weiche befindet sich ein Hebel, der nach rechts oder links gelegt werden kann. Sobald dieser auf eine Seite gelegt wird, wird die entsprechende Bahn von ihm blockiert. Rollt eine Murmel diese Bahn hinunter, wird sie aufgehalten und bleibt auf dem Hebel liegen (b). Falls eine Murmel die benachbarte Bahn hinunterrollt, trifft sie die Unterseite des Hebels, legt ihn dadurch auf die andere Seite und kann danach weiterrollen (c). Nachdem der Hebel auf die andere Seite gekippt wurde, kann durch diesen Mechanismus außerdem eine andere Murmel weiterrollen, die zuvor durch den Hebel blockiert wurde (d). Es ist ebenfalls möglich, dass eine rollende Murmel auf eine durch einen Hebel blockierte Murmel trifft. In diesem Fall wechselt die rollende Murmel ihre Bahn (e), rollt dort weiter, legt den Hebel um und gibt so die blockierte Murmel frei. Kommt eine Murmel an das Ende der Bahn, verlässt sie das Spielbrett (f).

Jedes Avalanche-Spielbrett besteht aus vier verschiedenen Bauteilen (siehe Abb. 2.21): dem Anfangsstück (*Start*), das den Beginn jeder Bahn markiert, der geraden Bahn (*Straight*), der Weiche inkl. Hebel (*Switch*) und dem Endstück (*End*), welches das Ende einer Bahn darstellt.

Avalanche – Editor

Ein Editor zur Erstellung individueller Avalanche-Spielbretter wird in Abb. 2.22 abgebildet. Dieser Editor ist zusätzlich in der Lage, die Mechanik des Spiels auf gültigen Spielbrettern (d. h. alle Bauteile ergeben ein vollständiges Spielbrett) zu testen. Der Benutzer kann hierfür Murmeln zu beliebigen Zeitpunkten auf den Anfangsstücken platzieren. Danach beginnt die Murmel zu rollen und verhält sich entsprechend der Spielmechanik. Bereits während der Animation kann der Benutzer weitere Murmeln platzieren oder Hebel manuell umlegen. Insgesamt entspricht dies zwar nicht den Avalanche-Spielregeln, aber es genügt zur Konstruktion einer einfachen Fallstudie für interaktive animierte Sprachen. Der Editor lässt sich, v. a. mit dem später vorgestellten Ansatz, problemlos erweitern, so dass Avalanche darin auch gespielt werden kann. Hinzugefügt werden müsste

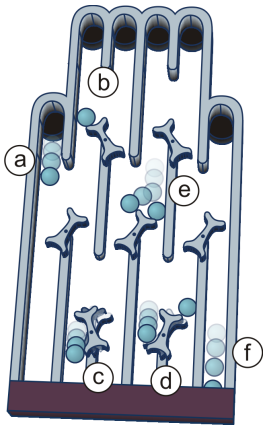


Abbildung 2.20: Avalanche-Spielbrett

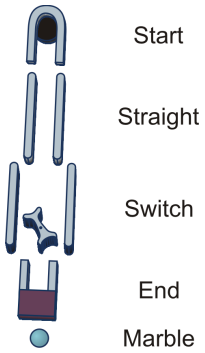


Abbildung 2.21: Avalanche-Bauteile

lediglich die Unterstützung für mehrere Spieler inkl. Zugreihenfolge und die Registrierung von Punkten.

Die Umsetzung von Avalanche wird bereits in zwei Artikeln beschrieben. Der grundlegende Ansatz zur Umsetzung wird in [SM10] präsentiert. Darin wird zunächst gezeigt, wie Animationen mittels GTs beschrieben werden können. Danach werden GTRs manuell definiert, um einen entsprechenden Editor zu generieren. Der Folgeartikel [SM12] beschäftigt sich mit dem gleichen modellgetriebenen Ansatz, der auch in dieser Arbeit präsentiert wird, d. h., die benötigten GTRs werden aus einem Modell, in dem u. a. Zustände spezifiziert werden, automatisch abgeleitet.

2.3.4 Weitere untersuchte Sprachen und Konzepte

In den folgenden Abschnitten werden zwei weitere Sprachen und damit verbundene Konzepte aufgelistet, die in dieser Arbeit nicht vollständig beschrieben werden. Die Sprachen wurden jedoch untersucht und können mit dem später präsentierten Ansatz modelliert und in Form von Editoren generiert werden. Sie sollen noch einmal verdeutlichen, dass die Menge an umsetzbaren Sprachen nicht auf die zuvor erwähnten Beispiele beschränkt ist, sondern ein relativ breites Spektrum an Möglichkeiten und Ideen in Bezug auf animierte Sprachen realisiert werden kann.

Traffic

In [SMPV10] wird eine visuelle Sprache namens Traffic spezifiziert. In einem Editor für diese Sprache soll es Benutzern möglich sein, Straßennetze inkl. Kreuzungen zu modellieren. Einstellbar sind außerdem Schaltzyklen der Ampelanlagen

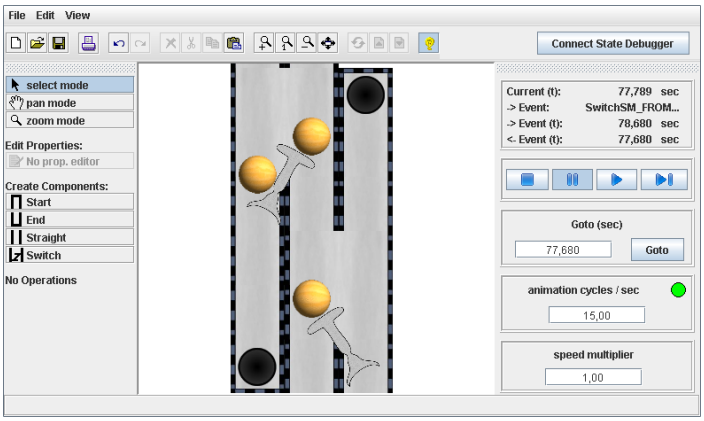


Abbildung 2.22: Screenshot – Avalanche

mit vier Signalzuständen sowie der Erwartungswert einer Wahrscheinlichkeitsverteilung für die Ankunft neuer Fahzeuge an Zugängen zum Straßennetz.

Ähnlich wie bei Petri-Netzen erfolgt die Animation der Sprache dabei während der Simulation, die in diesem Fall eine typische Verkehrssimulation ist. Im Gegensatz zur Simulation für ein B/E-Netz (vgl. Abschnitt 2.3.1, S. 32) ist die Simulation aber vergleichsweise komplex. Darin müssen viele simulierte Fahrzeuge berücksichtigt werden, die sich gleichzeitig über die Straßen bewegen und unterschiedliche Aufgaben erfüllen, die nur teilweise unabhängig voneinander sind, z. B. den Abstand zum vorausfahrenden Fahrzeug einhalten, Entscheidungen über Abbiegevorgänge treffen (inkl. Verwendung der Blinker), Verkehrsregeln an Ampelanlagen achten – wobei nur angehalten wird, falls dies noch möglich ist – oder Gegenverkehr und Stausituationen beobachten. Selbst Beschleunigungs- und Bremsvorgänge werden berücksichtigt, anstelle eines etwa abrupten Bewegungsstarts oder -endes. Anders als im Editor für B/E-Netze gibt es für Traffic außerdem interaktive Möglichkeiten für den Editornutzer. Er kann während des Verkehrsflusses bspw. Ampelanlagen manuell umschalten, um Stausituationen besser zu lösen.

Betrachtet man Traffic als animierbare Sprache, so zeichnet sie sich also v. a. durch ihre Komplexität aus sowie durch die Tatsache, dass sich darin viele Sprachelemente (Agenten/Akteure) bewegen, die unabhängig voneinander Entscheidungen treffen müssen. Darüber hinaus bietet Traffic zu wenige interessante Merkmale, die nicht in irgendeiner Weise von den anderen wiederkehrenden Beispielen dieser Arbeit (z. B. Avalanche) besprochen werden, weshalb von einer kompletten Ausarbeitung abgesehen wurde.

Wie für alle anderen Sprachen wurde auch für Traffic ein Editor generiert, dessen Spezifikation auf GTRs beruht. Der Screenshot des Tools zur Sprachspezifikation in Abb. 2.23 zeigt auf der linken Seite die Liste der erstellten GTRs. Wie zu erkennen ist, waren viele GTRs notwendig und es wäre sehr schwierig gewesen, die GTRs ohne zusätzliches Hilfsmittel zu erstellen. Unter Verwendung von

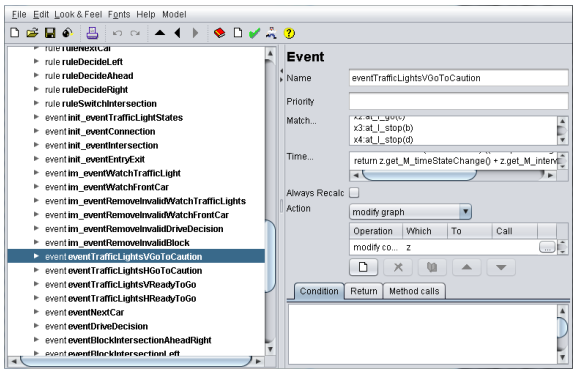


Abbildung 2.23: Regel- und Ereignisspezifikation für Traffic

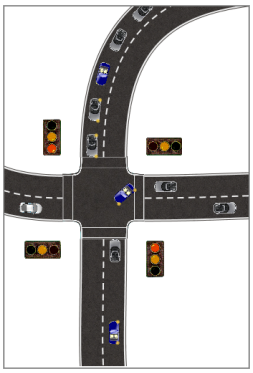


Abbildung 2.24: Screenshot - Traffic (vergrößert)

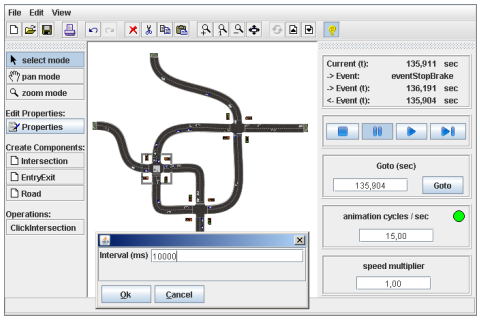


Abbildung 2.25: Screenshot - Traffic (Animationssicht)

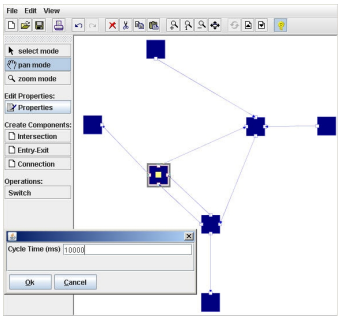


Abbildung 2.26: Screenshot – Traffic (abstrakte Sicht)

GTRs fehlt ein Überblick, wie die einzelnen Regeln zusammenhängen und welche Zustände Verkehrsteilnehmer oder Ampelanlagen überhaupt einnehmen können. Um sich einen solchen Überblick zu verschaffen, wurde die Modellierungssprache AML zur Planung eingesetzt (siehe Kapitel 5). Später konnte das resultierende AML-Modell genutzt werden, um entsprechende GTRs teilweise abzuleiten. Dies führte zu der Idee, die in dieser Arbeit aufgegriffen wird: die Erweiterung der Modellierungssprache AML, um einen modellgetriebenen Ansatz entwickeln zu können, der die vollständige Ableitung der Editoren aus solchen Modellen erlaubt (siehe Kapitel 6 und Kapitel 7).

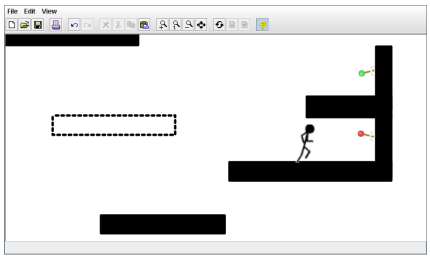


Abbildung 2.27: Screenshot – ToonProg

Abb. 2.25 bildet einen Screenshot des Traffic-Editors ab. Es sollte ebenfalls erwähnt werden, dass im Rahmen dieses Editors untersucht wurde, ob es mit dem verwendeten Ansatz auch möglich ist, unterschiedliche Sichten zu nutzen. Dies konnte durch Erweiterung des verwendeten Editorframeworks und zusätzliche grafische Spezifikationsmöglichkeiten aber problemlos erreicht werden. Für Traffic wurde daher eine (abstrakte) Sicht spezifiziert, die nur zum Editieren des Verkehrsnetzes dient. Darin werden Kreuzungen und Ein-/Ausgänge als Quadrate dargestellt und Straßen als Pfeile, welche die Quadrate miteinander verbinden. Abb. 2.26 zeigt das Verkehrsnetz aus Abb. 2.25 in abstrakter Form. Letztere Abbildung stellt die zweite Sicht dar und dient zur Animationsdarstellung, die in Abb. 2.24 noch einmal vergrößert gezeigt wird. Darin sind u. a. unterschiedliche Fahrzeuge zu erkennen, ein Abbiegevorgang und eine Ampelanlage, die im Begriff ist, umzuschalten und entsprechende Achtungssignale anzeigt.

ToonProg

Der Editor für eine Sprache namens ToonProg war ein frühes Experiment, um die Realisierbarkeit von animierten Sprachen im Kontext einer virtuellen Welt zu analysieren. Ähnlich wie in ToonTalk (vgl. Abschnitt 2.2, S. 18) soll der Editornutzer eine Figur (den Avatar) innerhalb der Welt steuern, mit ihr interagieren und so ein ablauffähiges Programm erstellen können.

Der entstandene experimentelle Editor bietet solche Möglichkeiten allerdings nur ansatzweise und ist nur bis zu einem Punkt entwickelt worden, an dem der Editornutzer seinen Avatar per Tastatur steuern und in der virtuellen Welt einige Effekte erzielen kann. So kann der Avatar bspw. springen, Plattformen erzeugen, auf denen er sich dann auch fortbewegen kann, oder Schalter platzieren und bedienen. Per Maus können auch weitere Elemente erstellt und editiert werden, die zum Teil auch unsichtbar geschaltet werden können. Bei Betätigung eines Schalters durch den Avatar kann u. a. der Zustand von Plattformen verändert werden, z. B. kann eine Plattform durchlässig werden, was dazu führt, dass der Avatar darauf nicht mehr stehen kann. Ein Screenshot des Editors ist in Abb. 2.27 zu sehen.

Kapitel 3

Graphen als Modell für Diagramme und Editorgenerierung

Diagrammeditoren können unter Einsatz unterschiedlicher Ansätze realisiert werden. Neben der Möglichkeit, Editoren manuell zu implementieren, gibt es bspw. Editor-Generatoren, die den Großteil des Editor-Codes auf der Grundlage einer Sprachspezifikation erzeugen können. Zusammen mit einem Editor-Framework wird der Programmieraufwand dadurch enorm verringert.

Eine wichtige Frage bei der Generierung von Diagrammeditoren ist, wie eine (Diagramm-)Sprache spezifiziert wird. Die Antwort ist oftmals damit verbunden, wie der generierte Editor Diagrammdaten verarbeitet, wozu auch Aspekte wie Validierung oder Simulation gehören. Eine etablierte Methode ist die Verwendung von Graphen als internes Modell für Diagramme. Diese Methode bildet auch das Fundament dieser Arbeit.

Daher werden in diesem Kapitel zunächst Graphen als Modell für Diagramme begründet. Dabei wird detailliert auf die in dieser Arbeit relevante Klasse der Hypergraphen eingegangen und erläutert, wie sie als Datenstruktur für Diagramme genutzt werden können (Abschnitt 3.1). Ein weiterer Abschnitt beschreibt Graphtransformationen als mögliches Mittel zur Diagramm-Manipulation (Abschnitt 3.2). Ein Überblick über das konkret eingesetzte DiaMeta-System zur Erzeugung von Diagramm-Editoren folgt im Anschluss (Abschnitt 3.3). Schließlich werden im letzten Abschnitt noch einmal andere Ansätze und Editor-Generatoren gelistet und kurz beschrieben (Abschnitt 3.4).

3.1 Hypergraphen als Diagramm-Modell

Wie eingangs erwähnt, können Graphen als internes Modell für Diagramme verwendet werden. Genauer gesagt, es werden im präsentierten Ansatz soge-

nannte Hypergraphen eingesetzt. Abschnitt 3.1.1 erläutert einige Gründe, die für die Verwendung von Hypergraphen sprechen. Es folgen formale Definitionen und Beschreibungen von Hypergraphen in Abschnitt 3.1.2. Abschließend wird in Abschnitt 3.1.3 erläutert, wie Hypergraphen für die Repräsentation von Diagrammen genutzt werden können.

3.1.1 Anforderungen und Motivation

Diagramme bestehen aus einer endlichen Menge von Diagrammkomponenten. Diese lassen sich meist frei positionieren. Aus diesem Grund unterstützen viele Editoren das sogenannte *freie Editieren*. In diesem Modus darf der Editornutzer die Komponenten frei platzieren oder löschen. Editoren ähneln in diesem Modus daher oft Zeichenprogrammen. Teilweise können solche Editoren sogar die Erkennung handgezeichneter Bilder (in einem Zeichenprogramm oder auf Papier) unterstützen (siehe [Bri10]).

Die Syntax der Diagrammsprache gibt bzgl. der Positionierung aber meist einen gewissen Rahmen vor. In diesem Fall ist der Benutzer oftmals auf eine Syntaxhilfe angewiesen, v. a. bei komplexen Diagrammen bzw. Diagrammsprachen. Daher gibt es Editoren, die das sogenannte *strukturierte/syntaxgesteuerte Editieren* unterstützen. In diesem Modus wird dem Benutzer eine Auswahl an Editieroperationen zur Verfügung gestellt. Durch deren Anwendung wird das Diagramm in vordefinierter Weise modifiziert, wobei die Möglichkeiten von einfachen bis hin zu komplexen Änderungen reichen. Dieser Modus stellt eine Art Gegensatz zum freien Editieren dar und nur wenige Editoren bzw. Editoren-Frameworks stellen beide Modi bereit (vgl. [Min01, Min02]).

Die technische Umsetzung beider Modi ist meist aufwendig. Beim freien Editieren sollte für eine anwenderfreundliche Bedienung zusätzliche Unterstützung angeboten werden. So kann bspw. nach jeder Diagrammänderung überprüft werden, ob das entstandene Diagramm syntaktisch und semantisch korrekt ist.[1] Es existieren auch Syntaxhilfen, die dem Benutzer Hinweise zur Erstellung eines korrekten Diagramms geben (z. B. [Maz10]). Auch die Implementierung der Operationen für das strukturierte Editieren kann sich schwierig gestalten, da verschiedene Aspekte wie Anwendbarkeit der Operationen etc. berücksichtigt werden müssen. Unabhängig vom Modus sind außerdem Layoutmechanismen wünschenswert, die Diagrammkomponenten zielgerichtet und mit gültigem Layout anordnen können (z. B. [MMM08]).

Die erwähnten Punkte stellen mögliche Anforderungen an Editoren dar, und es ist schwierig, Techniken zu finden, mit denen derartige Editoren effektiv umgesetzt werden können. Man stößt bei einer Suche aber unweigerlich auf Graphen, die als interne Datenstruktur für Diagramme genutzt werden können. Durch die Verwendung von Graphen wird eine formale Basis für die Diagramme geschaffen, die bspw. für deren Manipulation, Syntaxanalyse, Modelltransformationen oder – wie später präsentiert – Animation eingesetzt werden kann. Durch die Möglichkeit,

[1] Im Gegensatz dazu geht man beim strukturierten Editieren meist davon aus, dass Diagramm-Manipulationen, die durch den Editor zur Verfügung gestellt werden, nicht zu ungültigen Diagrammen führen.

Beziehungen zwischen den Diagrammkomponenten einfach darzustellen, bieten Graphen im Vergleich zu vielen anderen Techniken (z. B. „attribut-basierte“ Repräsentationen wie Constraint Multiset Grammars, siehe [MM96]) außerdem viele Vorteile.

Oft werden typisierte, attributierte Graphen eingesetzt (vgl. Abschnitt 3.4, S. 95), wobei jeder Knoten eine Diagrammkomponente repräsentiert und Kanten genutzt werden, um Beziehungen zwischen den Diagrammkomponenten darzustellen. Der Typ eines Knotens wird verwendet, um die Art der Diagrammkomponenten festzulegen. Die Attribute können bspw. zur Speicherung der x- und y-Koordinaten der Diagrammkomponente dienen. Auf diese Weise kann die konkrete Syntax eines Diagramms vollständig in Form eines Graphen gespeichert werden, der im Folgenden daher auch konkreter Syntaxgraph (KSG) genannt wird.

Die beschriebene Verwendung von Graphen ist aber nicht die einzige Möglichkeit. Minas propagiert einen Ansatz, in dem Hypergraphen verwendet werden [Min01, Min06a]. Darin repräsentieren typisierte, attributierte Hyperkanten sowohl Diagrammkomponenten als auch Verbindungen. Die Knoten stellen hingegen (mögliche) Verbindungspunkte einzelner Diagrammkomponenten dar. Auch einige der zuvor referenzierten Arbeiten ([MMM08, Bri10, Maz10]) beschreiben Ansätze, die Hypergraphen in beschriebener Weise nutzen.

Schließlich bieten Hypergraphen auch eine solide Grundlage für animierte Editoren und Sprachen, da *Graphtransformationen (GTs)* genutzt werden können, um einfache oder komplexe Modifikationen am Diagramm während der Animation vorzunehmen. Mögliche Modifikationen können *regelbasiert* mittels *Graphtransformationsregeln (GTRs)* festgelegt werden. Dies ähnelt der Vorgehensweise für strukturiertes Editieren, weshalb die Spezifikation von animierten Sprachen auch mit ähnlichen Mitteln erfolgen kann. Details diesbezüglich werden in Kapitel 4 behandelt.

3.1.2 Verwendete Klasse von Hypergraphen

Hypergraphen sind – in der in dieser Arbeit präsentierten Form – eine Verallgemeinerung *gewöhnlicher gerichteter Graphen*, die aus einer endlichen Menge von *Knoten* und *Kanten* bestehen. Während die Kanten gewöhnlicher Graphen aber immer zwei *Knoten* miteinander verbinden (oder auch einen Knoten mit sich selbst verbindet), können *Hyperkanten*, wie die Kanten von Hypergraphen genannt werden, beliebig viele Knoten miteinander verbinden. In diesem Zusammenhang spricht man auch davon, dass die *Tentakel* einer Hyperkante die Knoten *besuchen*. Zusätzlich sollen die verwendeten Hyperkanten auch *Markierungen* tragen, die im weiteren Verlauf der Arbeit auch als *Typ* der Hyperkante bezeichnet werden. Dieser Typ legt u. a. die *Stelligkeit* der Hyperkante fest, d. h., wie viele Tentakel diese hat.

Es folgen die formalen Definitionen für Hypergraphen und weitere Konzepte, die für das Verständnis dieser Arbeit und die notwendigen Strukturen wichtig

sind. Die Definitionen wurden im Wesentlichen aus [Min01] entnommen. Dort können auch weiterführende Einzelheiten gefunden werden.

Hilfsdefinitionen
Im Allgemeinen werden folgende Notationen und Symbole verwendet:

- $\mathbb{B} = \{wahr, falsch\}$ bezeichne die Menge der Booleschen Werte.
- $\mathbf{P}(X)$ bezeichne die *Potenzmenge* für eine gegebene Menge X.
- $|X|$ bezeichne die Kardinalität einer gegebenen Menge X.
- ε bezeichne das *leere Wort*.
- X^n bezeichne für eine gegebene Menge X und eine nichtnegative ganze Zahl $n \in \mathbb{N}_0$ die Menge aller Wörter über X der Länge n.
- $X^* = \bigcup_{n \in \mathbb{N}_0} X^n$ ist die Menge aller Wörter über X einschließlich ε.
- $|w|$ bezeichne die Länge n des Worts $w \in X^n$.
- $[w]$ bezeichne die Menge aller in Wort w vorkommenden Symbole.
- $f|_X$ schränkt die Funktion f auf den Definitionsbereich X ein.
- $f^* : X^* \to Y^*$ sei die auf Wörter definierte, symbolweise Erweiterung einer Funktion $f : X \to Y$ mit $f^*(x_1 \dots x_k) = f(x_1) \dots f(x_k)$ für alle $k \in \mathbb{N}_0$ und $x_i \in X$ für alle $i \in \mathbb{N}^+ \wedge i \leq k$.

Speziell in Bezug auf Hypergraphen wird definiert:

- $\mathcal{L}$ heiße *Markierungsalphabet* und ist eine endliche Menge sogenannter *Markierungen*, wobei
- $arity : \mathcal{L} \to \mathbb{N}_0$ jeder Markierung $lbl \in \mathcal{L}$ ihre *Stelligkeit* zuordnet. △

Definition 3.1 (Hypergraph)
Ein *(kantenmarkierter) Hypergraph* über ein Markierungsalphabet $\mathcal{L}$ ist ein Quadrupel $H = (V_H, E_H, vis_H, lab_H)$, wobei

- V_H eine endliche Menge von *Knoten* ist,
- E_H eine endliche Menge von *Hyperkanten* ist,
- $vis_H : E_H \to V_H^*$ jeder Hyperkante $e \in E_H$ die sortierte Folge der *besuchten* Knoten $vis_H(e)$ zuordnet und
- $lab_H : E_H \to \mathcal{L}$ jeder Hyperkante $e \in E_H$ eine *Markierung* $lab_H(e)$ zuordnet, so dass gilt $arity(lab_H(e)) = |vis_H(e)|$.

$\mathcal{H}_\mathcal{L}$ bezeichne die Menge aller kantenmarkierter Hypergraphen über $\mathcal{L}$. △

Im Gegensatz zu [DKH97] wird durch diese Definition nicht gefordert, dass die von einer Hyperkante besuchten Knoten paarweise verschieden sein müssen, d. h., ein Knoten kann von einer Hyperkante mehrfach besucht werden.

In Bezug auf Hypergraphen werden außerdem folgende Begriffe bzw. Operationen verwendet (vgl. [Min01]):

Definition 3.2 (Teilhypergraph)
Ein Hypergraph H' heiße *Teilhypergraph* von H, geschrieben $H' \subseteq H$, wenn gilt $E_{H'} \subseteq E_H$, $V_{H'} \subseteq V_H$, $vis_{H'} = vis_H|_{E_{H'}}$ und $lab_{H'} = lab_H|_{E_{H'}}$. △

Definition 3.3 (Vereinigungshypergraph)
Gilt für zwei Hypergraphen $H_1, H_2 \in \mathcal{H}_\mathcal{L}$, dass $lab_{H_1}(e) = lab_{H_2}(e)$ und $vis_{H_1}(e) = vis_{H_2}(e)$ für alle Hyperkanten $e \in E_{H_1} \cap E_{H_2}$, dann ist der *Vereinigungshypergraph* $H = H_1 \cup H_2$ definiert durch $E_H = E_{H_1} \cup E_{H_2}$, $V_H = V_{H_1} \cup V_{H_2}$,

$$vis_H(e) = \begin{cases} vis_{H_1}(e) & \text{für alle } e \in E_{H_1} \\ vis_{H_2}(e) & \text{sonst} \end{cases} \quad \text{und}$$

$$lab_H(e) = \begin{cases} lab_{H_1}(e) & \text{für alle } e \in E_{H_1} \\ lab_{H_2}(e) & \text{sonst} \end{cases} \quad . \qquad \triangle$$

In einigen Fällen wird auch der *Schnitthypergraph* benötigt. Ohne eine vergleichbare Definition aus einer anderen Arbeit sei dieser folgendermaßen gegeben:

Definition 3.4 (Schnitthypergraph)
Gilt für zwei Hypergraphen $H_1, H_2 \in \mathcal{H}_\mathcal{L}$, dass $lab_{H_1}(e) = lab_{H_2}(e)$ und $vis_{H_1}(e) = vis_{H_2}(e)$ für alle Hyperkanten $e \in E_{H_1} \cap E_{H_2}$, dann ist der *Schnitthypergraph* $H = H_1 \cap H_2$ definiert durch $E_H = E_{H_1} \cap E_{H_2}$, $V_H = V_{H_1} \cap V_{H_2}$, $vis_H = vis_{H_1}|_{E_{H_1} \cap E_{H_2}}$ und $lab_H = lab_{H_1}|_{E_{H_1} \cap E_{H_2}}$. △

Der Schnitthypergraph H ist der (größte) gemeinsame Teilhypergraph von H_1 und H_2, da aus offensichtlichen Gründen gilt $H \subseteq H_1$ und $H \subseteq H_2$.

Attributierte Hypergraphen

Für viele Anwendungsgebiete sind Hypergraphen interessant, in denen Knoten und/oder Hyperkanten mit Attributen bzw. Attributwerten versehen werden können. Dabei werden meist Vorschriften verwendet, die festlegen, wie Elemente mit bestimmten Markierungen attributiert werden können. So könnte bspw. festgelegt werden, dass jeder Hyperkante mit Markierung M ein (ganzzahliger) Wert für ein Attribut namens x zugeordnet werden muss. Details und formaler Aufbau einer solchen Vorschrift sollen im Rahmen dieser Arbeit aber nicht genau definiert werden. Wie später in Beispielen ersichtlich, entspricht die Vorstellung von Attributen und Attributwerten gängigen Vorstellungen, weshalb auf eigentlich notwendige – im Falle von Attributen meist komplexe – Definitionen verzichtet wurde.

Wie aus der objektorientierten Programmierung bekannt, werden die für eine Hyperkante[2] verfügbaren Attribute durch ihre Markierung (die „Klasse“) bestimmt. In einem attributierten Hypergraphen muss somit jedem verfügbaren Attribut von jeder Hyperkante (das „Objekt“) ein Attributwert zugeordnet werden. Dies ist jeweils ein Wert einer Wertemenge, die durch den *Attributtyp* bestimmt wird.

Um anzudeuten, dass ein Markierungsalphabet mit der Spezifikation von verfügbaren Attributen notwendig ist, werden im Folgenden sogenannte *Markierungsalphabete mit Attributen* $\mathcal{LV}$ verwendet. Sie enthalten neben dem regulären Markierungsalphabet $\mathcal{L}$ ein weiteres (an dieser Stelle nicht genauer definiertes) Element $\mathcal{V}$, das festlegt, wie Attributwerte den Hyperkanten zugeordnet werden müssen, d. h., welche Attribute eine Markierung besitzt.

Definition 3.5 (Attributierter Hypergraph)
Gegeben sei ein Markierungsalphabet mit Attributen $\mathcal{LV} = (\mathcal{L}, \mathcal{V})$. Ein *attributierter Hypergraph* über $\mathcal{LV}$ ist ein Paar (H, A), geschrieben H_A, mit dem Hypergraphen $H \in \mathcal{H}_\mathcal{L}$ und der Zuordnung von Attributwerten A, welche die Vorgaben durch $\mathcal{V}$ *erfüllt*.

$\mathcal{H}_{\mathcal{LV}}$ bezeichne die Menge aller kantenmarkierter, attributierter Hypergraphen über $\mathcal{LV}$. △

Typisierung

Auch die Typisierung der Graphen durch die Kantenmarkierungen wird für den später präsentierten Ansatz nicht genau genug definiert. Das formale Modell mit einem Markierungsalphabet $\mathcal{L}$ als Typen ist stark vereinfacht. Tatsächlich wird nämlich davon ausgegangen, dass ein Typsystem verwendet wird, welches bspw. Typhierarchien oder Vererbung (von Attributen) unterstützt. Als grundlegende Bedingung soll lediglich gefordert werden, dass $arity(B) \geq arity(A)$, falls gilt $B \sqsubset A$, d. h., B ist ein abgeleiteter Typ von A. Damit wird das Liskovsche Substitutionsprinzip [LW94] bzgl. der Knotenanzahl eingehalten. Ansonsten sollen die benötigten Konzepte auf bekannte Vorstellungen der objektorientierten Programmierung basieren. Beispiele werden zeigen, welche Bedeutung ein solches Typsystem im Rahmen von GTs hat (z. B. Beispiel 3.6, S. 69).

Weiterführende Arbeiten

An dieser Stelle soll auf Arbeiten hingewiesen werden, die Attributierung und Typisierung in Bezug auf Graphen auch formal definieren. In [Min01] werden strukturiert markierte Hypergraphen eingeführt, d. h. doppelt markierte Hypergraphen, deren zweite Markierung die Attributwerte auf der formalen Grundlage von strukturierten Alphabeten repräsentiert. Ein weiterer Ansatz wird in [EPT04] beschrieben. Die darin präsentierten E-Graphen enthalten Datenknoten. Durch Zuordnung zu einem sogenannten Typgraphen ist die Interpretation dieser Datenknoten als Attributwerte der im Typgraphen spezifizierten

[2] Auf die Attributierung von Knoten wird verzichtet, da für Knoten keine Markierungen festgelegt werden bzw. diese nicht typisiert sind.

Attribute möglich. In [BEL+03] werden Typgraphen beschrieben, in der die Vererbung von Knoten inkl. abstrakter Knoten spezifiziert werden können. Aufbauend auf [EPT04] und [BEL+03] behandelt [LBE+07] sowohl Attributierung als auch Vererbung von Knotentypen. Interessant sind die genannten Arbeiten v. a. deshalb, weil die darin dargestellten GTs auf demselben Ansatz basieren wie die GTs, die in dieser Arbeit verwendeten werden (DPO-Ansatz, siehe Abschnitt 3.2.1, S. 60). Allerdings gab es bereits zuvor wichtige Arbeiten in diesem Bereich, wie bspw. [LKW93], [BK00] oder [HKT02].

3.1.3 Arten verwendeter Hyperkanten

Hyperkanten werden im präsentierten Ansatz sowohl für Diagrammkomponenten als auch für Beziehungen zwischen diesen verwendet. Dieser Abschnitt stellt verschiedene Arten von Hyperkanten und deren Besonderheiten vor (vgl. [Min00, Min01]).

Neben ihrem eigentlichen Typ bzw. ihrer Markierung werden Hyperkanten unterschieden in *Komponentenhyperkanten* und *Verbindungshyperkanten*. Dabei werden Verbindungshyperkanten noch weiter untergliedert in *Relationshyperkanten* und *Mehrzweck-Hyperkanten*. Die jeweilige Art wird von der Markierung der Hyperkante widergespiegelt, weshalb sich das Markierungsalphabet zusammensetzt aus: $\mathcal{L} = \mathcal{CMP} \cup \mathcal{LNK}$ mit $\mathcal{LNK} = \mathcal{REL} \cup \mathcal{GEN}$, wobei

- $\mathcal{CMP}$ die Menge der Markierungen für *Komponentenhyperkanten*,
- $\mathcal{LNK}$ die Menge der Markierungen für *Verbindungshyperkanten*,
- $\mathcal{REL}$ die Menge der Markierungen für *Relationshyperkanten* und
- $\mathcal{GEN}$ die Menge der Markierungen für *Mehrzweck-Hyperkanten* ist.

Komponentenhyperkanten

Wie der Name bereits andeutet, repräsentieren Komponentenhyperkanten die Diagrammkomponenten. Jeder Art von Diagrammkomponente ist dabei eine Markierung für Komponentenhyperkanten $lbl \in \mathcal{CMP}$ zugeordnet. Für Alligator Eggs gibt es z. B. eine Markierung für Alligator-Diagrammkomponenten, für Avalanche eine Markierung für Murmel-Diagrammkomponenten usw.

Jede Komponentenhyperkante muss mindestens einen Knoten besuchen, d. h., es gilt $arity(lbl) \geq 1$. Die Knoten, die dabei von den Tentakeln jeder Komponentenhyperkante besucht werden, sind untrennbar mit dieser verbunden, d. h., Komponentenhyperkante und Knoten gehören zusammen und können nur gemeinsam erzeugt oder entfernt werden. Außerdem gilt für jede Komponentenhyperkante $e \in E_H$ mit $lab_H(e) \in \mathcal{CMP}$, dass $arity(lab_H(e)) = |[vis_H(e)]|$, d. h., eine Komponentenhyperkante besucht keinen Knoten mehrfach.

Jeder Knoten repräsentiert einen sogenannten *Konnektor* (engl. attachment area) der Diagrammkomponente, d. h., abhängig von der Markierung der Komponentenhyperkante wird jedem zugehörigen Knoten ein Konnektor zugeordnet. Als

Konnektoren werden dabei bestimmte Punkte und Bereiche der Diagrammkomponente bezeichnet, die mit anderen Diagrammkomponenten in einer räumlichen Beziehung stehen können. Mehr dazu wird im folgenden Unterabschnitt über Relationshyperkanten und im abschließenden Beispiel beschrieben.

Zudem besitzen Komponentenhyperkanten meist Attribute. Sie werden benutzt, um visuelle Merkmale, wie bspw. die Position der Diagrammkomponente, und andere Eigenschaften zu speichern. Andere Arten von Hyperkanten benötigen in der Regel keine Attribute, weshalb die Verwendung von Attributen bei anderen Hyperkantenarten nachfolgend nicht berücksichtigt wird.

Mehrzweck-Hyperkanten

Mehrzweck-Hyperkanten werden verwendet, um Beziehungen zwischen Diagrammkomponenten zu repräsentieren. Eine Mehrzweck-Hyperkante $lbl \in \mathcal{GEN}$ verbindet dabei meist die Knoten von Komponentenhyperkanten miteinander. Falls gilt $arity(lbl) = 1$, d. h., die Mehrzweck-Hyperkante ist unär, kann sie auch benutzt werden, um Knoten zu markieren. Der genaue Zweck einer solchen Hyperkante kann beliebig festgelegt werden.

In den Beispielen dieser Arbeit und auch im präsentierten Ansatz werden Mehrzweck-Hyperkanten vielfältig genutzt. Für Avalanche existieren bspw. Mehrzweck-Hyperkanten mit den Markierungen *startedAt* oder *switchedTo*. Diese Hyperkanten bestimmen den aktuellen (Teil-)Zustand einer Murmel oder einer Weiche (vgl. Beispiel 3.2, S. 58). Da Mehrzweck-Hyperkanten auch verwendet werden können, um Animationen zu repräsentieren (siehe Kapitel 4), können sie im entsprechenden Kontext auch als *Animationshyperkanten* bezeichnet werden.

Relationshyperkanten

Eine Relationshyperkante repräsentiert immer die räumliche Beziehung zwischen zwei Diagrammkomponenten. Im Gegensatz zu einer Mehrzweck-Hyperkante ist eine Relationshyperkante $lbl \in \mathcal{REL}$ daher immer binär, d. h., es gilt $arity(lbl) = 2$.

Eine existierende Relationshyperkante sagt aus, dass der Konnektor einer Diagrammkomponente in einer räumlichen Relation mit einem Konnektor einer anderen Diagrammkomponente steht. Häufig bedeutet dies, dass sich die Bereiche von Konnektoren überlagern. In Avalanche können die Bauteile bspw. untereinander angeordnet werden, wodurch sich bestimmte Konnektoren überlappen. Die zugehörigen Knoten der Komponentenhyperkanten werden dann mittels Relationshyperkanten miteinander verbunden (vgl. Beispiel 3.1, S. 58). Hierfür müssen mögliche Relationen für bestimmte Konntektorenpaare spezifiziert werden. Zur Laufzeit eines Editors kann die Lage der Diagrammkomponenten und deren Konnektoren dann entsprechend analysiert werden, um Relationshyperkanten *automatisch* zu erzeugen bzw. zu entfernen, falls eine Relation gemäß Spezifikation vorhanden ist bzw. nicht mehr vorhanden ist.

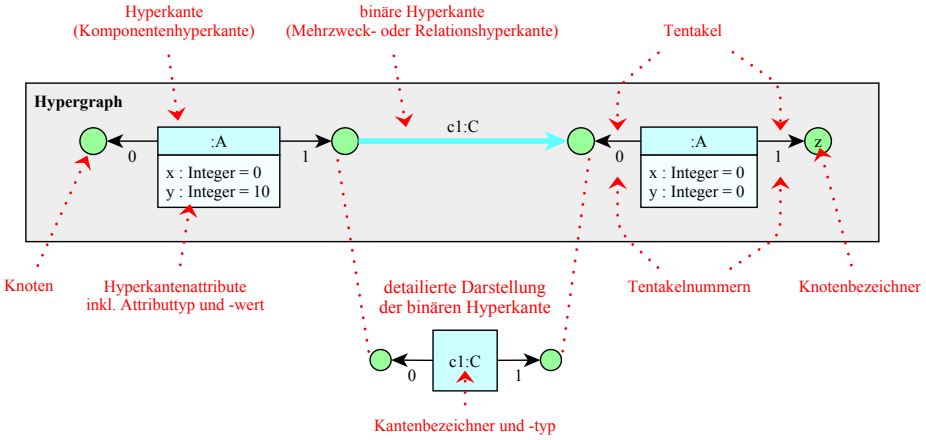


Abbildung 3.1: Zeichnerische Darstellung eines Hypergraphen

Grafische Darstellung der Hypergraphen

Abb. 3.1 zeigt, wie Hypergraphen in allen Beispielen grafisch abgebildet werden. Knoten werden durch Kreise dargestellt, Hyperkanten hingegen als Rechtecke. Attribute von Komponentenhyperkanten werden in einem separaten Rechteck unterhalb notiert. Dünne Pfeile von der Hyperkante zu den Knoten stellen die Tentakel der Hyperkante dar, d. h., sie zeigen, welche Knoten durch die Kante miteinander verbunden werden. Da eine solche Verbindung durch eine geordnete Menge festgelegt wird, muss neben den Pfeilen auch die jeweilige *Tentakelnummer* notiert werden. Binäre Verbindungshyperkanten werden außerdem durch dicke Pfeile veranschaulicht, die vom Quellknoten (Tentakel 0) zum Zielknoten (Tentakel 1) gerichtet sind.

Der Bezeichner und die Markierung (der Typ) der Hyperkante werden textuell neben dem Pfeil oder innerhalb der Form notiert[3], wobei Bezeichner und Markierung durch ein Doppelpunkt voneinander getrennt werden. Der Bezeichner darf weggelassen werden, falls dieser im Kontext nicht relevant ist. Dies gilt auch im Falle von Knoten, wobei diese aber keine Markierung haben können.

Einige Abbildungen sollen außerdem ein spezifiziertes Markierungsalphabet (mit Attributen) $\mathcal{LV}$ darstellen. Abb. 3.2 zeigt ein zum Hypergraphen in Abb. 3.1 passendes Beispiel. Darin werden Markierungen inkl. Knotenanzahl in Form von Hyperkanten ohne konkreten Bezeichner oder „:“ gezeigt. Die Knoten werden im Falle von Komponentenhyperkanten mit durchgezogener Linie gezeichnet. Bei Verbindungshyperkanten (z. B. bei Markierung C) wird ein gestrichelter Rahmen verwendet, um anzudeuten, dass die Knoten nicht zur Hyperkante gehören. Ob es sich dabei um eine Relationshyperkante oder Mehrzweck-Hyperkante handelt, ist daraus nicht ersichtlich, da es für die Darstellung der Hypergraphen unwesentlich

[3] Der Bezeichner ist ein Hilfsmittel bei der Erstellung von GTRs. Die Bezeichner werden v. a. dann verwendet, um auszudrücken, dass Knoten aus unterschiedlichen Graphen (bzw. Graphmustern) der GTR einander entsprechen müssen.

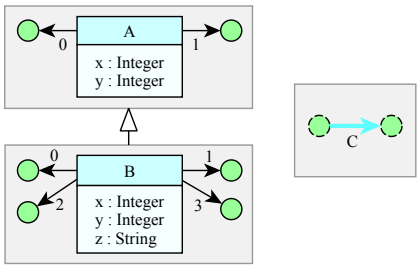


Abbildung 3.2: Spezifizierte Hyperkanten bzw. Markierungen

ist. Außerdem stellt die Abbildung dar, welche Attribute (inkl. Attributtypen) die Komponentenhyperkanten besitzen und wie die Typhierarchie aufgebaut ist. Beispielsweise ist Markierung B von Markierung A abgeleitet, d. h., jede B-Hyperkante *ist* eine A-Hyperkante.

Beispiel 3.1 (Avalanche-Komponenten mit Konnektoren und Relationen)
In Abb. 3.3 werden die Diagrammkomponenten von Avalanche dargestellt, wie sie auch im Editor grafisch abgebildet werden. An den Diagrammkomponenten werden zusätzlich die Bereiche angedeutet, welche den Konnektoren der Komponenten entsprechen. Die meisten dieser Bereiche sind rechteckig und werden mit Hilfe von Koordinaten spezifiziert. Eine Ausnahme gibt es bei Komponentenhyperkante *Marble*, denn für sie wird spezifiziert, dass ihre (runde) Form einem Konnektor entspricht.[4]

Für eine Gerade wird bspw. oben und unten jeweils ein rechteckiger Bereich als Konnektor festgelegt, beschriftet mit 0 und 1. Diese Konnektoren entsprechen den Knoten an Tentakel 0 und Tentakel 1 der Komponentenhyperkante *Straight*. Weiterhin wurde für Avalanche spezifiziert, dass die Bereiche an den oberen Enden von Bauteilen (*Straight-0*, *End-0*, *Switch-0* und *Switch-1*) mit Bereichen an den unteren Enden in Relation stehen können (*Straight-1*, *Start-0*, *Switch-2* und *Switch-3*). Schneiden sich zwei solche Bereiche, wie bei a und b in Abb. 3.4 angedeutet, soll daher automatisch die für diese Relationen spezifizierte Relationshyperkante *topBottom* erstellt werden, wie rechts in der Abbildung gezeigt wird. △

Beispiel 3.2 (Avalanche-Spielbrett und internes Graphmodell)
In Abb. 3.5 wird gezeigt, wie ein komplettes Avalanche-Spielbrett (links, schematische Darstellung) durch einen Hypergraphen (rechts) repräsentiert wird. Die Positionen der einzelnen Bauteile des Spielbretts werden durch Attributwerte der Komponentenhyperkanten (*xPos, yPos*) festgelegt.

Das Beispiel veranschaulicht ebenfalls die Verwendung der Mehrzweck-Hyperkanten *startedAt* und *switchedTo*. Während erstere den Startpunkt einer fallenden Murmel markiert, hält die zweite im Hypergraphen fest, ob der Hebel

[4] Für eine effiziente Verarbeitung berechnet und verwendet das später vorgestellte DiaMeta-System allerdings das minimale Rechteck, das diese Form umschließt.

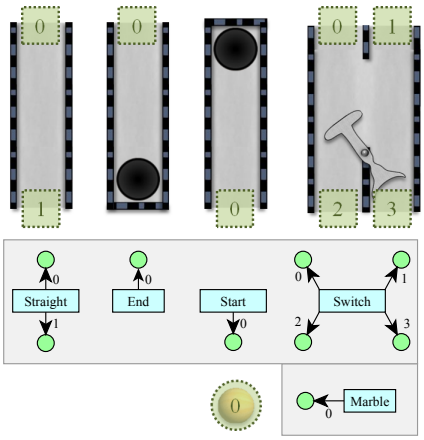


Abbildung 3.3: Avalanche-Bauteile mit Konnektoren und Hyperkanten

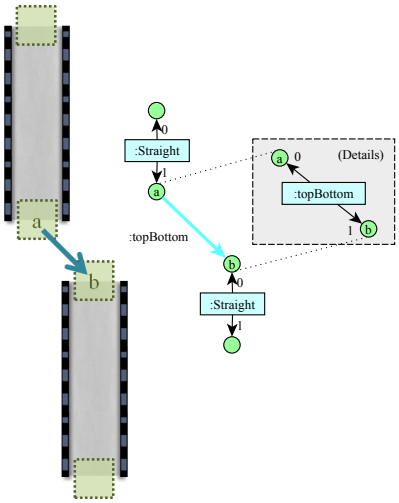


Abbildung 3.4: Automatische Verbindung bei Überlappung

einer Weiche auf die linke oder rechte Seite gekippt ist. Der Hebel im Beispiel ist auf die linke Seite gekippt, wodurch die linke Bahn, d. h. die Bahn der Murmel, geblockt wird. △

3.2 Diagrammmodifikation durch Graphtransformationen

Falls Diagramme durch Graphen repräsentiert werden, sind *Graphtransformationen (GTs)* ein geeignetes Mittel, um Diagramme zu manipulieren. Voraussetzung hierfür ist die Verwendung eines Graphtransformationssystems (GTS). Mögliche Diagrammmanipulationen bzw. Graphtransformationen werden im Rahmen des GTS oftmals in Form von *Graphtransformationsregeln (GTRs)* spezifiziert. Diese GTRs können auch als Operationen für strukturiertes Editieren verwendet werden (vgl. Abschnitt 3.1.1, S. 50).

In diesem Abschnitt werden zunächst einige Grundlagen für GTs und GTRs (Abschnitt 3.2.1 und Abschnitt 3.2.2) erläutert.[5] Anschließend werden spezielle Arten von GTRs beschrieben (Abschnitt 3.2.3, Abschnitt 3.2.4 und Abschnitt 3.2.5), die in späteren Kapiteln benötigt werden. Am Ende des Abschnitts wird auf einige beispielhafte GTSe verwiesen (Abschnitt 3.2.6).

Hilfsdefinitionen
Ist ein Hypergraph $H \in \mathcal{H}_{\mathcal{L}}$ gegeben, werden nachfolgende Konstrukte verwendet.

[5] Einen anderen, kurz und einfach gehaltenen Einstieg in diese Thematik bietet [Hec06].

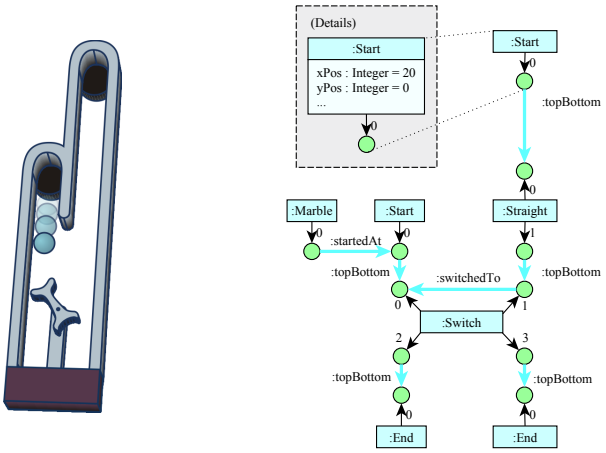


Abbildung 3.5: AVALANCHE-Spielbrett und zugehöriger Hypergraph

- $\overline{vis}_H : V_H \to \mathbf{P}(E_H)$ ist die Funktion, die jedem Knoten $v \in V_H$ die Menge der besuchenden Hyperkanten zuordnet:

$$\overline{vis}_H(v) = \{e \in E_H \mid v \in [vis_H(e)]\} \ .$$

- $E_H|_{\mathcal{X}}$ mit einer gegebener Menge von Markierungen $\mathcal{X} \subseteq \mathcal{L}$ bezeichne die Teilmenge von E_H, welche genau die Hyperkanten mit den Markierungen in $\mathcal{X}$ enthält:

$$E_H|_{\mathcal{X}} = \{e \in E_H \mid lab(e) \in \mathcal{X}\} \ .$$

Spezielle Hypergraphen sind:

- $H^{\emptyset} \in \mathcal{H}_{\mathcal{L}}$ sei der *leere Hypergraph*, der weder Hyperkanten noch Knoten enthält, d. h. $E_{H^{\emptyset}} = \emptyset$ und $V_{H^{\emptyset}} = \emptyset$.
- $H \in \mathcal{H}_{\mathcal{L}}$ heiße *elementarer Hypergraph*, falls gilt $|E_H| = 1$ und $|\overline{vis}_H(v)| = 1$ für alle $v \in V_H$.
- $H_{lbl} \in \mathcal{H}_{\mathcal{L}}$ mit einer Markierung für Komponentenhyperkanten $lbl \in \mathcal{CMP}$ bezeichne den elementaren Hypergraphen, der lediglich aus einer Komponentenhyperkante mit Markierung lbl und ihren besuchten Knoten besteht, d. h. $E_{H_{lbl}} = \{e\}$, $lab_{H_{lbl}}(e) = lbl$, $V_{H_{lbl}} = \{v_1, v_2, \ldots, v_{arity(lbl)}\}$ und $vis_{H_{lbl}}(e) = v_1 v_2 \ldots v_{arity(lbl)}$. Um zu verdeutlichen, dass $e \in E_{H_{lbl}}$, wird auch die Schreibweise $H_{e:lbl}$ verwendet. △

3.2.1 Grundlagen und DPO-Ansatz

Das wichtigste Instrument zur formalen Beschreibung von GTRs und GTs sind sogenannte *Graph- bzw. Hypergraphmorphismen*. Sie können als Abbildungen

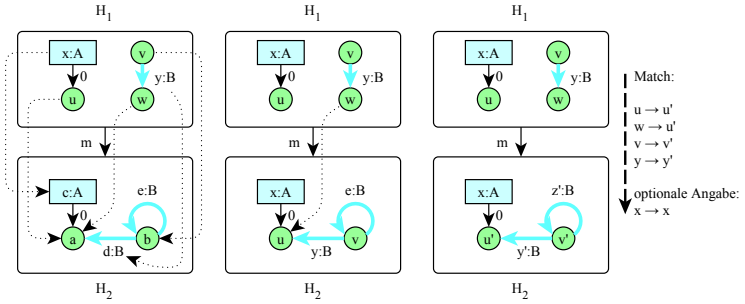


Abbildung 3.6: Grafische Darstellung von Hypergraphmorphismen

zwischen Hypergraphen betrachtet werden. Die folgenden Definitionen in Bezug auf Hypergraphmorphismen (mit Ausnahme auf deren Vereinigung und Hypergraphinklusionen) wurden sinngemäß aus [Min01] übernommen.

Definition 3.6 (Hypergraphmorphismus)
Gegeben seien zwei Hypergraphen $H_1, H_2 \in \mathcal{H}_{\mathcal{L}}$. Ein *Hypergraphmorphismus* $H_1 \xrightarrow{m} H_2$ (oder $m : H_1 \to H_2$) ist ein Paar $m = (m_V, m_E)$ mit den Funktionen $m_V : V_{H_1} \to V_{H_2}$ und $m_E : E_{H_1} \to E_{H_2}$, wobei $vis_{H_2}(m_E(e)) = m_V^*(vis_{H_1}(e))$ und $lab_{H_1}(e) = lab_{H_2}(m_E(e))$ für alle Hyperkanten $e \in E_{H_1}$ gilt. △

Wie in der Definition zu erkennen ist, bleiben die Markierungen der Hyperkanten im Rahmen des Hypergraphmorphismus erhalten, wodurch auch die Anzahl der zugeordneten Knoten identisch bleibt.

Beispiel 3.3 (Grafische Darstellung von Hypergraphmorphismen)
In Abb. 3.6 (links) werden zwei Hypergraphen H_1, H_2 und ein Hypergraphmorphismus m grafisch abgebildet. Die gestrichelten Linien stellen die einzelnen Elemente von m dar. Der Hypergraphmorphismus wird also festgelegt durch $m_V = \{(u, a), (v, b), (w, a)\}$ und $m_E = \{(x, c), (y, d)\}$.

Da gestrichelte Linien die Grafiken sehr überladen, werden auch alternativen Darstellungsformen verwendet. Tragen in H_1, H_2 bspw. mehrere Elemente den gleichen Bezeichner, wie in Abb. 3.6 (mittig) gezeigt, können gestrichelte Pfeile für entsprechende Paare einfach weggelassen werden. In Abb. 3.6 (rechts) werden Informationen über zusammengehörige Paare, die Teil des Hypergraphmorphismus sind, textuell dargestellt. Ob das jeweilige Paar Teil von m_V oder m_E ist, kann aus der zugehörigen Grafik abgeleitet werden. △

Falls nicht anders erwähnt, sind verwendete Hypergraphmorphismen total, d. h., in einem Hypergraphmorphismus m sind sowohl m_V als auch m_E totale Funktionen. In einigen Fällen werden allerdings *partielle Hypergraphmorphismen* benötigt:

Definition 3.7 (Partieller Hypergraphmorphismus)
Gegeben seien drei Hypergraphen $H_1, H_1', H_2 \in \mathcal{H}_\mathcal{L}$, wobei gilt $H_1' \subseteq H_1$. Ein *partieller Hypergraphmorphismus* $H_1 \overset{p}{\rightharpoonup} H_2$ ist ein Hypergraphmorphismus $H_1' \overset{p}{\to} H_2$, wobei Hypergraph H_1' *Definitionsbereich* von p, geschrieben $dom(p) = H_1'$, heißt.

Gegeben sei zusätzlich ein Hypergraphmorphismus $H_1 \overset{m}{\to} H_2$. Es wird geschrieben $p \subseteq m$, falls gilt $p_V = m_V|_{V_{H_1'}}$ und $p_E = m_E|_{E_{H_1'}}$. △

Außerdem müssen folgende Begriffe und Operationen in Bezug auf Hypergraphmorphismen bestimmt werden:

Definition 3.8 (Komposition von Hypergraphmorphismen)
Gegeben seien drei Hypergraphen $H_1, H_2, H_3 \in \mathcal{H}_\mathcal{L}$ und zwei Hypergraphmorphismen $H_1 \overset{f}{\to} H_2$ und $H_2 \overset{g}{\to} H_3$. Die *Komposition* der Hypergraphmorphismen g und f, geschrieben $g \circ f$, ist der Hypergraphmorphismus $H_1 \overset{g \circ f}{\to} H_3$, der komponentenweise definiert ist durch $(g_V \circ f_V, g_E \circ f_E)$. △

Die Komposition $g \circ f$ ergibt dabei einen Hypergraphmorphismus, da für alle $e \in E_{H_1}$ gilt:

$$\begin{aligned} lab_{H_1}(e) = lab_{H_2}(f_E(e)) = lab_{H_3}(g_E(f_E(e))) = lab_{H_3}((g \circ f)_E(e)) \quad &\text{und} \\ vis_{H_3}((g \circ f)_E(e)) = vis_{H_3}(g_E(f_E(e))) = g_V^*(vis_{H_2}(f_E(e))) & \\ = g_V^*(f_V^*(vis_{H_1}(e))) = (g \circ f)_V^*(vis_{H_1}(e)) \quad &. \end{aligned}$$

Im Folgenden wird auch die Komposition von partiellen Hypergraphmorphismen $H_2 \overset{g}{\rightharpoonup} H_3$ und $H_1 \overset{f}{\rightharpoonup} H_2$ benötigt. In diesem Fall ist der Definitionsbereich eingeschränkt, d. h. $dom(g \circ f) = H_1'$. Dabei ist H_1' definiert durch

- $E_{H_1'} = \{e \in E_{dom(f)} \mid f_E(e) \in E_{dom(g)}\}$,
- $V_{H_1'} = \{v \in V_{dom(f)} \mid f_V(v) \in V_{dom(g)}\}$,
- $vis_{H_1'} = vis_{H_1}|_{E_{H_1'}}$ und
- $lab_{H_1'} = lab_{H_1}|_{E_{H_1'}}$.

Definition 3.9 (Vereinigung von Hypergraphmorphismen)
Gegeben seien drei Hypergraphen $H_1, H_2, H_3 \in \mathcal{H}_\mathcal{L}$ und zwei Hypergraphmorphismen $H_1 \overset{f}{\to} H_3$ und $H_2 \overset{g}{\to} H_3$, wobei $f_E(e) = g_E(e)$ für alle Hyperkanten $e \in E_{H_1} \cap E_{H_2}$ und $f_V(v) = g_V(v)$ für alle Hyperkanten $v \in V_{H_1} \cap V_{H_2}$ gilt. Die *Vereinigung* der Hypergraphmorphismen f und g, geschrieben $f \cup g$, ist der Hypergraphmorphismus $H_1 \cup H_2 \overset{h}{\to} H_3$, der definiert ist durch

$$h_V(v) = \begin{cases} f_V(v) & \text{für alle } v \in V_{H_1} \\ g_V(v) & \text{sonst} \end{cases} \quad \text{und}$$

$$h_E(e) = \begin{cases} f_E(e) & \text{für alle } e \in E_{H_1} \\ g_E(e) & \text{sonst} \end{cases} \quad . \qquad \triangle$$

Definition 3.10 (Mono-/Epi-/Isomorphismus)
Gegeben seien zwei Hypergraphen $H_1, H_2 \in \mathcal{H}_\mathcal{L}$. Ein Hypergraphmorphismus $m = (m_V, m_E)$ heißt

- *Hypergraphmonomorphismus*, geschrieben $H_1 \overset{m}{\rightarrowtail} H_2$ bzw. $H_1 \overset{m}{\nrightarrow} H_2$ bei einem partiellen Hypergraphmorphismus, wenn sowohl m_V als auch m_E *injektiv* sind,
- *Hypergraphepimorphismus*, wenn sowohl m_V als auch m_E *surjektiv* sind,
- *Hypergraphisomorphismus*, wenn sowohl m_V als auch m_E *bijektiv* sind. △

Vor allem Hypergraphmonomorphismen werden im weiteren Verlauf der Arbeit häufiger benötigt. Vereinfacht ausgedrückt dürfen bei einem Hypergraphmonomorphismus nicht zwei oder mehrere Knoten bzw. Hyperkanten von H_1 auf nur einen Knoten bzw. eine Hyperkante von H_2 abgebildet werden.

Definition 3.11 (Hypergraphinklusion)
Gegeben seien zwei Hypergraphen $H, H' \in \mathcal{H}_\mathcal{L}$, wobei gilt $H' \subseteq H$. Die *Hypergraphinklusion* $H' \overset{m}{\hookrightarrow} H$ ist der Hypergraphmonomorphismus $H' \overset{m}{\rightarrowtail} H$ mit $m_V(v) = v$ für alle $v \in V_{H'}$ und $m_E(e) = e$ für alle $e \in E_{H'}$. △

Definition 3.12 (Partielle Hypergraphinklusion)
Gegeben seien zwei Hypergraphen $H_1, H_2 \in \mathcal{H}_\mathcal{L}$, wobei $lab_{H_1}(e) = lab_{H_2}(e)$ und $vis_{H_1}(e) = vis_{H_2}(e)$ für alle Hyperkanten $e \in E_{H_1} \cap E_{H_2}$ gilt. Die *partielle Hypergraphinklusion* $H_1 \overset{m}{\hookrightarrow} H_2$ ist der partielle Hypergraphmonomorphismus $H_1 \overset{m}{\nrightarrow} H_2$ mit $dom(m) = H_1 \cap H_2$, $m_V(v) = v$ für alle $v \in (V_{H_1} \cap V_{H_2})$ und $m_E(e) = e$ für alle $e \in (E_{H_1} \cap E_{H_2})$. △

Der Begriff und die Definition der Hypergraphinklusion leiten sich von der Graphinklusion (vgl. [LEO05]) ab. Bei diesem speziellen Graphmorphismus wird ein (ursprünglicher) Teilgraph H_1 in einen (erweiterten) Graphen $H_2 \supseteq H_1$ abgebildet, wobei alle Knoten und Kanten jeweils auf sich selbst abgebildet werden. Im Falle einer partiellen Hypergraphinklusion wird $H_1 \subseteq H_2$ nicht gefordert, d. h., in H_2 können auch Knoten und Kanten von H_1 „fehlen“. Lediglich Knoten und Kanten, die sowohl Teil von H_1 als auch H_2 sind, werden jeweils auf sich selbst abgebildet. Benötigt wird die partielle Hypergraphinklusion für die Definition zur Anwendung von Graphtransformationsprogrammen (siehe Definition 3.30, S. 81).

Im Folgenden werden auch sogenannte *Pushouts* verwendet. Dieser Begriff stammt aus der *Kategorientheorie*, auf die aber nicht im Detail eingegangen wird. Ein Ziel der Kategorientheorie ist es, mathematische Strukturen auf eine abstraktere Ebene zu übertragen und dort zu betrachten, um Beweise in allgemeinerem Rahmen führen zu können. In diesem Zweig der Mathematik spielen sogenannte *Kategorien* eine wichtige Rolle. Beispiele für solche Kategorien sind die Kategorien der Mengen, der Graphen oder der Hypergraphen. Eine Kategorie besteht dabei aus *Objekten* und *Morphismen* (Pfeile). Im Falle der Kategorie der Hypergraphen sind Hypergraphen die Objekte und Hypergraphmorphismen

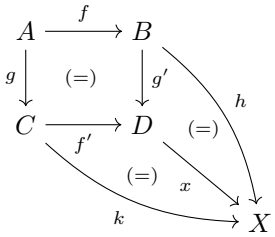
die Morphismen. Weiterführende Definitionen und Beispiele diesbezüglich finden sich in [HS73, RB88]. Speziell auf die Kategorie der Graphen und Hypergraphen konzentriert sich [DW80]. Folgende Definition kann auch in [Erm06] gefunden werden:

Definition 3.13 (Pushout)
Gegeben seien zwei Morphismen $f : A \to B$ und $g : A \to C$ in einer Kategorie $\mathcal{C}$. Ein *Pushout* über f und g ist ein Tripel (D, f', g') mit

- einem (Pushout-)Objekt D und
- den Morphismen $f' : C \to D$ und $g' : B \to D$ mit $f' \circ g = g' \circ f$,

so dass die *universelle Eigenschaft* erfüllt wird. Diese wird erfüllt, falls es für alle Objekte X und Morphismen $h : B \to X$ und $k : C \to X$ mit $k \circ g = h \circ f$ einen eindeutigen Morphismus $x : D \to X$ gibt, so dass gilt $x \circ g' = h$ und $x \circ f' = k$.△

Das folgende (kommutative) Diagramm veranschaulicht noch einmal visuell die in der Definition verwendeten Symbole:



Ausgehend von solchen Pushouts kann ein Formalismus für GTs aufgestellt werden, der dem sogenannten *algebraischen Ansatz* folgt (siehe [EPS73]). Unter Verwendung dieses Ansatzes können GTs durch die Konstruktion von Pushouts beschrieben werden. Es existieren zwei unterschiedliche, häufig verwendete Vorgehensweisen: den Single-Pushout-Ansatz (SPO-Ansatz) und den Double-Pushout-Ansatz (DPO-Ansatz), der im Gegensatz zum SPO-Ansatz zwei anstatt einem Pushout zur Konstruktion des transformierten Graphen nutzt. Vertieft wird dieses Thema in [CMR$^+$97, EHK$^+$97].

Einfache Graphtransformationsregeln

In beiden Ansätzen werden zur Spezifikation einer GTR eine linke Seite (LHS) und eine rechte Seite (RHS) benötigt. Sowohl LHS als auch RHS werden durch Graphen dargestellt. Dabei beschreibt die LHS eine Situation, die vor der GT zutreffen muss, und die RHS beschreibt die Situation danach. Unter Verwendung des DPO-Ansatz muss zusätzlich ein *Klebegraph* spezifiziert werden, wodurch Konflikte vermieden werden, die bei Verwendung des SPO-Ansatzes möglich sind (vgl. [EHK$^+$97]). Daher kommt in dieser Arbeit ausschließlich der DPO-Ansatz zur formalen Darstellung zum Einsatz. Auf treffende Bezeichnungen wie DPO-GTR oder DPO-GT, die auf die Verwendung des DPO-Ansatzes hindeuten, wird zugunsten einer besseren Lesbarkeit verzichtet.

Die folgenden beiden Definitionen zeigen, wie *einfache GTRs* formal spezifiziert werden können und wie darauf basierende GTs (auch *Anwendung der GTR*) durch Konstruktion von Pushouts durchgeführt werden.

Definition 3.14 (Einfache GTR)
Gegeben sei ein Markierungsalphabet $\mathcal{L}$. Eine *einfache Graphtransformationsregel (GTR)* über $\mathcal{L}$ ist ein Tripel $egtr = (L, K, R)$ mit den Hypergraphen $L, K, R \in \mathcal{H}_\mathcal{L}$, genannt *linke Seite (LHS)*, *Klebegraph* und *rechte Seite (RHS)*, wobei gilt $L \supseteq K \subseteq R$.

$\mathcal{EG}_\mathcal{L}$ bezeichne die Menge aller einfachen GTRs über $\mathcal{L}$.

$egtr_L^\emptyset \in \mathcal{EG}_\mathcal{L}$ bezeichne für einen gegebenen Hypergraphen $L \in \mathcal{H}_\mathcal{L}$ die *einfache GTR ohne Änderung* mit LHS L. Für die darin enthaltenen Hypergraphen L, K und R gilt $L = K = R$. △

Diese und folgende Definition basieren auf den Definitionen aus [Erm06], auch wenn diese auf gewöhnlichen Graphen aufbauen. Ein wesentlicher Unterschied ist allerdings, dass die Hypergraphmonomorphismen $K \overset{l}{\rightarrowtail} L$ und $K \overset{r}{\rightarrowtail} R$ im Rahmen der einfachen GTR nicht spezifiziert werden müssen. Dies ist möglich, da die Einschränkung $K \subseteq L$ und $K \subseteq R$ gilt, wodurch festgelegt werden kann, dass l und r den Hypergraphinklusionen $K \overset{l}{\hookrightarrow} L$ und $K \overset{l}{\hookrightarrow} R$ entsprechen sollen.

Definition 3.15 (Anwendung einer einfachen GTR)
Gegeben seien eine einfache GTR $egtr \in \mathcal{EG}_\mathcal{L}$, ein *Ausgangshypergraph* $H \in \mathcal{H}_\mathcal{L}$ und ein Hypergraphmorphismus $L \overset{m}{\rightarrow} H$, genannt *Match*. Die einfache GTR $egtr$ heißt *anwendbar* auf (m, H), geschrieben $(m, H) \models egtr$, falls ein *Kontexthypergraph* $D \subseteq H$ existiert, so dass (1) in folgendem Diagramm ein Pushout ist:

$$\begin{array}{rccc} egtr: & L & \overset{l}{\hookleftarrow} & K \overset{r}{\hookrightarrow} R \\ & {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle d} \\ & H & \underset{h}{\hookleftarrow} & D \end{array}$$

Falls $(m, H) \models egtr$, können der Kontexthypergraph D und der *abgeleitete Hypergraph* $H^\dagger$ unter Verwendung von $egtr$ durch die folgenden beiden Pushouts (1) und (2), auch genannt Double-Pushout (DPO), konstruiert werden:

$$\begin{array}{rccccc} egtr: & L & \overset{l}{\hookleftarrow} & K & \overset{r}{\hookrightarrow} & R \\ & {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle d} & (2) & \downarrow{\scriptstyle m^\dagger} \\ & H & \underset{h}{\hookleftarrow} & D & \underset{h^\dagger}{\hookrightarrow} & H^\dagger \end{array}$$

Eine derartige Ableitung wird auch *Anwendung* einer einfachen GTR oder *einfache GT* genannt. Sollen alle Hypergraphmorphismen explizit genannt werden, wird geschrieben $H \overset{egtr, m, d, m^\dagger, h, h^\dagger}{\Longrightarrow} H^\dagger$, verkürzt auch $H \overset{egtr, m}{\Longrightarrow} H^\dagger$. △

Grob vereinfachend dargestellt, bezieht sich H auf den Graphen, der transformiert werden soll (daher Ausganghypergraph). Der Match m legt fest, an

welcher Stelle die GT durchgeführt wird. Der abgeleitete (bzw. transformierte) Graph $H^\dagger$ wird gebildet, indem aus dem Ausgangshypergraph H zunächst die Elemente von L ohne die Elemente von K entfernt und danach die Elemente von R ohne die Elemente von K hinzugefügt werden.

Neben der in der Definition gezeigten Schreibweise und deren Verwendung werden auch alternative Formen eingesetzt. Ist ein partieller Hypergraphmorphismus $p \subseteq m$, auch genannt *Teilmatch*, anstelle eines Matches m vorgegeben, so heißt $egtr$ anwendbar auf (p, H), geschrieben $(p, H) \approx egtr$, falls ein Match $m \supseteq p$ existiert, so dass gilt $(m, H) \models egtr$. Es wird geschrieben $H \overset{egtr,p}{\rightsquigarrow} H^\dagger$, falls ein Match $m \supseteq p$ für die Anwendung einer einfachen GTR beliebig gewählt werden kann.[6]

Vereinfachend soll angenommen werden, dass durch die Anwendung einfacher GTRs auch attributierte Hypergraphen transformiert werden können. Die Schreibweise unterscheidet sich dann nur geringfügig. Je nach Variante wird geschrieben: $H_A \xRightarrow{egtr,m,d,m^\dagger,h,h^\dagger} H^\dagger_{A'}$ (kurz $H_A \xRightarrow{egtr,m} H^\dagger_{A'}$), oder $H_A \overset{egtr,p}{\rightsquigarrow} H^\dagger_{A'}$.

Der abgeleitete Hypergraph $H^\dagger$ wird dabei gebildet, wie zuvor beschrieben, d. h., die Transformation wird zunächst unabhängig von den Attributen durchgeführt. Erst danach wird die Zuordnung von Attributwerten A' abgeleitet. Da derartige Zuordnungen allerdings nicht formal definiert sind, wird die Konstruktion von A' lediglich vereinfacht dargestellt: durch eine einfache GT werden grundsätzlich keine Attributwerte verändert, d. h., A' entspricht A zum größten Teil. Es werden lediglich Attributwerte von entfernten Hyperkanten aus A' gelöscht und Attributwerte für neue Hyperkanten in A' hinzugefügt. In letzterem Fall werden sie mit den Standardwerten des zugehörigen Attributtyps initialisiert, z. B. Null für Zahlenwerte oder die leere Zeichenkette für Zeichenketten.

In Definition 3.15 wird davon ausgegangen, dass $H^\dagger$ immer konstruiert werden kann, falls D existiert. Dies ist sichergestellt, da die Kategorie der Hypergraphen mit Hypergraphmorphismen Pushouts besitzt (vgl. [HS73]). Ob die GTR angewendet werden kann oder nicht, hängt also allein davon ab, ob ein geeigneter Hypergraph D gefunden werden kann. Da eine solche Suche kein einfaches Kriterium für die Entscheidung darstellt, kann auch folgender Ansatz verwendet werden (vgl. [CMR+97, Min01]). Demnach ist eine einfache GTR anwendbar, wenn die sogenannte *Klebebedingung* erfüllt wird.

Definition 3.16 (Klebebedingung)
Gegeben seien zwei Hypergraphmorphismen $K \overset{l}{\hookrightarrow} L$ und $L \overset{m}{\rightarrow} H$. Das Paar (l, m) *erfüllt* die *Klebebedingung*, wenn folgende drei Bedingungen erfüllt werden:

- $m_V(V_L \setminus l_V(V_K)) \cap [vis_H(e)] = \emptyset$ für jede Hyperkante $e \in E_H \setminus m_E(E_L)$ („Dangling Condition“),
- $a, b \in l_V(V_K)$ für alle Knoten $a, b \in V_L$ mit $a \neq b$ und $m_V(a) = m_V(b)$ („Identification Condition (1)“) und
- $e, f \in l_E(E_K)$ für alle Hyperkanten $e, f \in E_L$ mit $e \neq f$ und $m_E(e) = m_E(f)$ („Identification Condition (2)“). △

[6] Alle genannten Schreibweisen und Prinzipien gelten neben einfachen GTs analog für andere GT-Varianten, die in diesem und folgenden Abschnitten noch definiert werden.

Die „Dangling Condition“ stellt sicher, dass kein Knoten entfernt werden darf, falls dieser von einer Hyperkante besucht wird, die nicht entfernt wird. Eine Hyperkante, die einen entfernten Knoten besucht, wird auch *hängende Hyperkante* genannt. Würde eine solche Kante resultieren, so wäre D kein Hypergraph. Die „Identification Condition (1)“ fordert, dass ein Knoten in H, der zwei Urbilder in L hat, nicht entfernt werden darf. Die „Identification Condition (2)“ fordert dies für Hyperkanten.

Damit die Klebebedingung nicht verletzt wird, seien im Rahmen dieser Arbeit für alle einfachen GTRs folgende Bedingungen festgesetzt:

- $V_K = V_L$, d. h., es dürfen keine Knoten entfernt werden, wodurch auch das Entfernen von Komponentenhyperkanten verboten ist (vgl. Abschnitt 3.1.3, S. 55).
- m muss zwar kein Hypergraphmonomorphismus sein, allerdings muss $m_E|_{E_L|_{\mathcal{LNK}}}$ injektiv sein, d. h., jede Verbindungshyperkante in H darf nur ein Urbild in L haben.

Durch den ersten Punkt können weder die „Dangling Condition“ noch die „Identification Condition (1)“ verletzt werden. Zusammen mit dem zweiten Punkt ist es zudem nicht mehr möglich, die „Identification Condition (2)“ zu verletzen, da Verbindungshyperkanten nur ein Urbild in L haben und Komponentenhyperkanten aufgrund des ersten Punkts nicht gelöscht werden dürfen.

Natürlich muss es trotzdem möglich sein, Knoten oder Komponentenhyperkanten durch GTs zu entfernen. Da dies aufgrund der formulierten Einschränkungen nicht mittels einfacher GTRs erlaubt ist, wird in einem späteren Abschnitt eine spezielle GTR eingeführt (siehe Abschnitt 3.2.3, S. 73).

Beispiel 3.4 (Einfache GT)
Abb. 3.7 zeigt die einfache GTR *egtr_ex* inkl. GT. Die GTR ist im grau hinterlegten Block abgebildet. Die LHS legt fest, dass zwei Hyperkanten vom Typ A benötigt werden. Ihre beiden Knoten müssen durch eine binäre Hyperkante vom Typ B miteinander verbunden sein. Diese binäre Kante ist im Klebegraphen K nicht mehr vorhanden, d. h., sie soll entfernt werden. In der RHS wird jedoch eine neue binäre Hyperkante vom Typ B verwendet, d. h., eine solche Hyperkante muss erzeugt werden. Sie ist entgegengesetzt zur entfernten Kante gerichtet.

Unten links wird ein Beispielhypergraph H dargestellt, auf den diese Regel angewendet werden soll. Bei der Anwendung wird in H zunächst das Graphmuster gesucht, das durch die LHS definiert ist. Tatsächlich gibt es zwei mögliche Matches, wobei aber nur einer davon für die Anwendung der GTR gewählt werden kann. Auf der rechten Seite von Abb. 3.7 wird konkret angegeben, welcher Match m im Beispiel genutzt wird. Entsprechend der Regel wird aus H zunächst die Hyperkante w' entfernt und im Anschluss Hyperkante z' erzeugt.△

Graphtransformationen ohne Hypergraphmonomorphismus

Wie zuvor erwähnt, muss der Match für eine GT kein Hypergraphmonomorphismus sein. Allerdings soll bei der Angabe einer einfachen GTR kontrollierbar

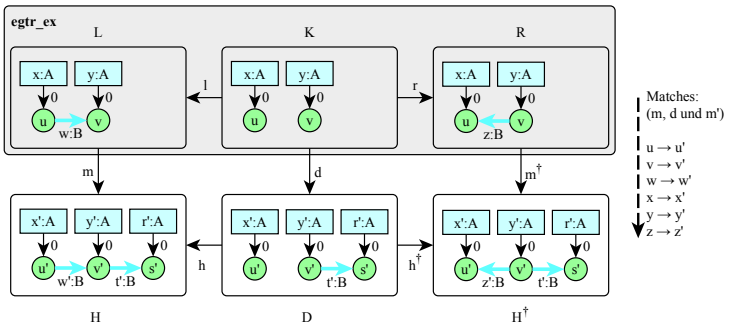


Abbildung 3.7: Einfache GTR und GT

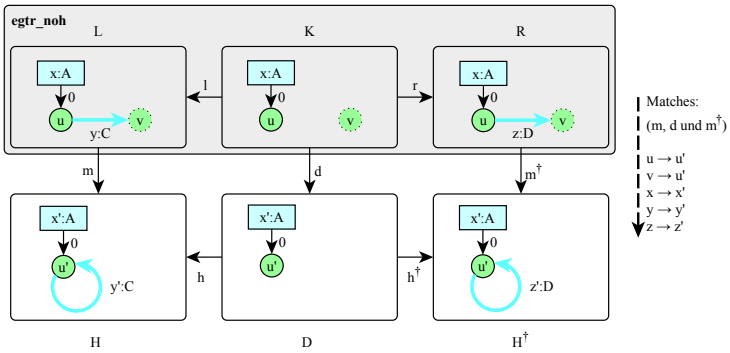


Abbildung 3.8: GT ohne Hypergraphmonomorphismus als Match

sein, ob bei der Suche eines Matches – falls dieser nicht vorgegeben ist – auch Hypergraphmorphismen berücksichtigt werden sollen, bei denen es sich nicht um Hypergraphmonomorphismen handelt. Ist dies der Fall, so soll auch angegeben werden können, welche Knoten bzw. Komponentenhyperkanten mit anderen Knoten bzw. Komponentenhyperkanten im Rahmen des Matches *zusammenfallen* dürfen. Derartige Spezifikationsmöglichkeiten sollen verwendet werden, obwohl diese in Definition 3.15, S. 65, nicht berücksichtigt werden. Die Darstellung in der GTR erfolgt durch gestrichelte Knoten und Komponentenhyperkanten. Diese Elemente dürfen dann bei der Matchsuche mit beliebigen anderen Knoten bzw. Komponentenhyperkanten zusammenfallen, d. h., auf dasselbe Element der Zielmenge abgebildet werden. Auf der anderen Seite dürfen zwei Knoten oder Komponentenhyperkanten nicht zusammenfallen, falls sie mit einer durchgezogene Linie umrahmt werden. Ist kein Element gestrichelt dargestellt, so muss der Match also ein Hypergraphmonomorphismus sein. Das folgende Beispiel zeigt eine einfache GT, die nach den beschriebenen Regeln arbeitet.

Beispiel 3.5 (Einfache GT ohne Hypergraphmonomorphismus)
Abb. 3.8 zeigt in L den Knoten v mit gestricheltem Rahmen. Dies bedeutet, dass der Knoten v bei der Suche nach einem Match m mit einem beliebigen

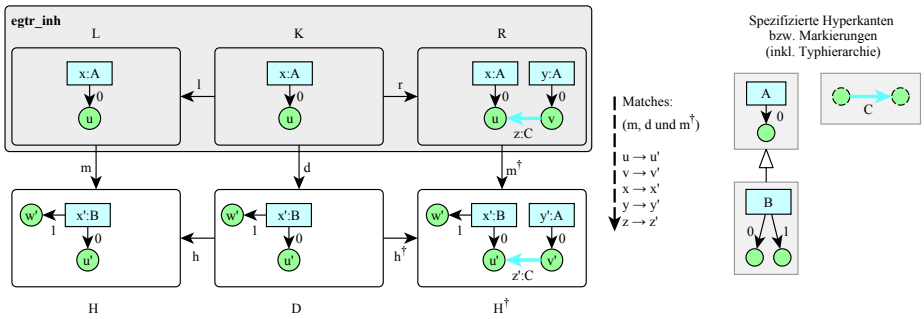


Abbildung 3.9: GT bei Typhierarchie

anderen Knoten aus L zusammenfallen darf. Gleiches gilt entsprechend für die Hypergraphmorphismen d und $m^\dagger$. Tatsächlich wird bzgl. H nur dann ein Match m gefunden, falls dies erlaubt ist. Durch m werden also sowohl u als auch v aus L auf den Knoten u' in H abgebildet. Aus diesem Grund kann auch Kante y auf Kante y' abgebildet werden. Letztendlich wird bei der dargestellten GT die Kante y' entfernt und Kante z' erzeugt. △

Graphtransformationen mit Typhierarchie

Ein abschließendes Beispiel soll zeigen, wie Typhierarchien für Hyperkanten bei GTs berücksichtigt werden können, was ebenfalls nicht formal definiert worden ist (vgl. Abschnitt 3.1.2, S. 54). Im Beispiel bildet der Match die Hyperkante eines Basistyps auf die Hyperkante eines abgeleiteten Typs ab.

Beispiel 3.6 (Einfache GT bei Typhierarchie)
Im Beispiel gelte für zwei Markierungen A und B: $B \sqsubset A$. Entsprechend typischer objektorientierter Systeme wird damit ausgesagt, dass jede Hyperkante des Typs B auch eine Hyperkante des Typs A ist. Die Grundvoraussetzung, nämlich $arity(A) \leq arity(B)$, werde dabei erfüllt. Wird also eine GTR definiert, in der Hyperkanten des Typs A verwendet werden, so kann diese auf naheliegende Weise auf Hyperkanten des Typs B angewendet werden.

Abb. 3.9 zeigt ein einfaches Beispiel. Durch die LHS wird eine Hyperkante des Typs A vorausgesetzt. Hyperkante x' in H erfüllt diese Bedingung, obwohl ihr konkreter Typ B ist. Der Knoten u wird dem Knoten u' zugeordnet.

Typhierarchien spielen allerdings nur für die Suche von „Matches“ eine Rolle. Soll durch die GTR bspw. eine Hyperkante erzeugt werden, so muss bei der GT eine Hyperkante des exakt gleichen Typs erzeugt werden, wie in der RHS spezifiziert. Im Beispiel ist dies eine Hyperkante des Typs A. △

3.2.2 Änderung von Attributwerten und Bedingungen

Neben der Transformation des Hypergraphen selbst müssen im Rahmen einer GT oftmals auch zusätzliche Modifikationen durchgeführt werden, z. B. eine

Änderung der Attributwerte. Es ist ebenfalls denkbar, dass eine GT an zusätzliche Bedingungen geknüpft werden soll. In diesen Fällen ist die Spezifikation von einfachen GTRs nicht ausreichend. Daher werden weitere Konzepte benötigt, die in diesem Abschnitt zunächst vorgestellt werden, um anschließend innerhalb sogenannter *erweiterter GTRs* genutzt zu werden.

Definition 3.17 (Attributberechnungsregel)
Gegeben sei die RHS $R \in \mathcal{H}_{\mathcal{L}}$ einer einfachen GTR und das Markierungsalphabet mit Attributen $\mathcal{LV}$. Eine *Attributberechnungsregel* über R und $\mathcal{LV}$ ist eine Funktion acr, die jedem Paar $(m^\dagger, H^\dagger_{A'})$ mit dem Match $R \xrightarrow{m^\dagger} H^\dagger$ und dem attributierten Ausgangshypergraphen $H^\dagger_{A'} \in \mathcal{H}_{\mathcal{LV}}$ einen abgeleiteten attributierten Hypergraphen $H^\dagger_{A^\dagger} \in \mathcal{H}_{\mathcal{LV}}$ zuordnet.

Eine derartige Ableitung wird auch Anwendung einer Attributberechnungsregel genannt, geschrieben $H^\dagger_{A'} \xRightarrow{acr, m^\dagger} H^\dagger_{A^\dagger}$.

$\mathcal{ACR}_{R,\mathcal{LV}}$ bezeichne die Menge aller Attributberechnungsregeln über R und $\mathcal{LV}$.

$acr^\emptyset \in \mathcal{ACR}_{R,\mathcal{LV}}$ bezeichne die *Attributberechnungsregel ohne Änderung*. Für alle attributierten Hypergraphen $H^\dagger_{A'} \in \mathcal{H}_{\mathcal{LV}}$ und Matches $R \xrightarrow{m^\dagger} H^\dagger$ gilt $acr^\emptyset(m^\dagger, H^\dagger_{A'}) = H^\dagger_{A'}$. △

Wie aus der Definition ersichtlich, wird eine Attributberechnungsregel lediglich verwendet, um die aktuellen Attributwerte eines attributierten Hypergraphen zu verändern. Der Hypergraph selbst wird in seiner Struktur nicht verändert. Es dürfen jedoch Strukturinformationen im Rahmen der Attributberechnungsregel genutzt werden, um die *abgeleiteten Attributwerte* $A^\dagger$ zu bestimmen.

Desweiteren sollen mittels sogenannter *Anwendungsbedingungen* (vgl. [EH86, HHT96, EHK+97, Min01, Erm06]) Möglichkeiten geschaffen werden, um GTs nur unter bestimmten Voraussetzungen durchzuführen, auch wenn ein Match m gefunden werden kann bzw. festgesetzt wurde. In dieser Arbeit werden zwei Arten von Anwendungsbedingungen genutzt: funktionale und negative Anwendungsbedingungen.

Im Rahmen einer *funktionalen Anwendungsbedingung* wird eine Funktion festgelegt, die in erster Linie die Attributwerte der Hyperkanten (meist auch in Zusammenhang mit dem gefundenen Match) überprüfen soll. Die Auswertung kann dafür verwendet werden, um zu entscheiden, ob eine GTR anwendbar ist oder nicht. Dabei sind funktionale Anwendungsbedingungen den allgemeinen Anwendungsbedingungen, die in [Min01] als Prädikate über Hypergraphmorphismen definiert werden, sehr ähnlich.

Definition 3.18 (Funktionale Anwendungsbedingung)
Gegeben seien die LHS $L \in \mathcal{H}_{\mathcal{L}}$ einer einfachen GTR und das Markierungsalphabet mit Attributen $\mathcal{LV}$. Eine *funktionale Anwendungsbedingung* über L und $\mathcal{LV}$ ist eine Funktion con, die jedem Paar (m, H_A) mit dem Match $L \xrightarrow{m} H$ und dem attributierten Hypergraphen $H_A \in \mathcal{H}_{\mathcal{LV}}$ eine Anwendbarkeit $a \in \mathbb{B}$ zuordnet. Das Paar (m, H_A) *erfüllt* die funktionale Anwendungsbedingung con, geschrieben $(m, H_A) \models con$, falls gilt $con(m, H_A) = wahr$.

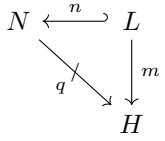
$\mathcal{CON}_{L,\mathcal{LV}}$ bezeichne die Menge aller funktionalen Anwendungsbedingungen über L und $\mathcal{LV}$. △

Eine spezielle Ausprägung von funktionalen Anwendungsbedingungen, die nach Pfaden im Hypergraphen sucht, wird in Abschnitt 3.2.4, S. 75, beschrieben.

Eine weitere Art von Anwendungsbedingungen sind die sogenannten *negativen Anwendungsbedingungen*. Diese werden ähnlich wie in [Erm06] definiert. Allerdings wird in dieser Arbeit ein Hypergraphinklusionsmorphismus n verwendet, der eindeutig aus den gegebenen Hypergraphen hervorgeht und daher nicht spezifiziert werden muss. Zusätzlich können funktionale Anwendungsbedingungen einbezogen werden.

Definition 3.19 (Negative Anwendungsbedingung)
Gegeben seien die LHS $L \in \mathcal{H}_{\mathcal{L}}$ einer einfachen GTR und das Markierungsalphabet mit Attributen $\mathcal{LV}$. Eine *negative Anwendungsbedingung* über L und $\mathcal{LV}$ ist ein Paar $nac = (N, CONS)$ mit einem Hypergraphen $N \in \mathcal{H}_{\mathcal{L}}$, wobei gelten muss $L \subseteq N$, und einer endlichen Menge funktionaler Anwendungsbedingungen $CONS \subseteq \mathcal{CON}_{L,\mathcal{LV}}$.

Ein attributierter Hypergraph $H_A \in \mathcal{H}_{\mathcal{LV}}$ und ein Hypergraphmorphismus $L \xrightarrow{m} H$ *erfüllen* eine negative Anwendungsbedingung *nac*, geschrieben $(m, H_A) \models nac$, falls es keinen Hypergraphmorphismus $N \xrightarrow{q} H$ gibt, so dass gilt $(q \circ n = m) \wedge (\forall con \in CONS : (q, H_A) \models con)$ mit $L \xhookrightarrow{n} N$:



$\mathcal{NAC}_{L,\mathcal{LV}}$ bezeichne die Menge aller negativen Anwendungsbedingungen über L und $\mathcal{LV}$. △

Durch eine negative Anwendungsbedingung wird zunächst ein Graph N festgelegt. Die negative Anwendungsbedingung wird erfüllt, falls kein von N ausgehender Match q existiert, wobei Bezug auf den verwendeten Match m genommen wird. Dadurch sind Formulierungen möglich wie etwa: „An einem Knoten, welcher der LHS einer GTR zugeordnet wurde, darf keine Hyperkante des Typs A hängen“. Zusätzlich darf eine negative Anwendungsbedingung auch funktionale Anwendungsbedingungen einbeziehen, d. h., dass eine negative Anwendungsbedingung nur dann nicht erfüllt wird, wenn neben einem gefundenen Match q alle zugehörigen funktionale Anwendungsbedingungen, die sich bspw. auf Attributwerte beziehen können, erfüllt werden.

Die in diesem Abschnitt formulierten Konzepte können in die Definition von erweiterten GTRs wie folgt eingebaut werden:

Definition 3.20 (Erweiterte GTR)
Gegeben sei ein Markierungsalphabet mit Attributen $\mathcal{LV} = (\mathcal{L}, \mathcal{V})$. Eine *erweiterte Graphtransformationsregel* über $\mathcal{LV}$ ist ein Quadrupel $fgtr = (egtr, acr, CONS, NACS) = ((L, K, R), acr, CONS, NACS)$ mit

- einer einfachen GTR $egtr \in \mathcal{EG}_{\mathcal{L}}$,
- einer Attributberechnungsregel $acr \in \mathcal{ACR}_{R,\mathcal{LV}}$,

- einer endlichen Menge funktionaler Anwendungsbedingungen $CONS \in \mathbf{P}(\mathcal{CON}_{L,\mathcal{LV}})$ und
- einer endlichen Menge negativer Anwendungsbedingungen $NACS \in \mathbf{P}(\mathcal{NAC}_{L,\mathcal{LV}})$.

$\mathcal{FG}_{\mathcal{LV}}$ bezeichne die Menge aller erweiterten GTRs über $\mathcal{LV}$. △

Definition 3.21 (Anwendung einer erweiterten GTR)
Gegeben seien eine erweiterte GTR $fgtr \in \mathcal{FG}_{\mathcal{LV}}$, ein attributierter Ausgangshypergraph $H_A \in \mathcal{H}_{\mathcal{LV}}$ und ein Match $L \xrightarrow{m} H$. Die erweiterte GTR $fgtr$ heißt *anwendbar* auf (m, H_A), geschrieben $(m, H_A) \models fgtr$, falls

- $(m, H) \models egtr$,
- $\forall con \in CONS : (m, H_A) \models con$ und
- $\forall nac \in NACS : (m, H_A) \models nac$.

Falls $(m, H_A) \models fgtr$, kann der abgeleitete, attributierte Hypergraph $H^{\dagger}_{A^{\dagger}}$ unter Verwendung von $fgtr$ durch die unmittelbar aufeinanderfolgenden Anwendungen von einfacher GTR und Attributberechnungsregel konstruiert werden:

$$H_A \xRightarrow{egtr,m,d,m^{\dagger},h,h^{\dagger}} H^{\dagger}_{A'} \xRightarrow{acr,m^{\dagger}} H^{\dagger}_{A^{\dagger}} \quad .$$

Eine derartige Ableitung wird auch Anwendung einer erweiterten GTR oder *erweiterte Graphtransformation* genannt. Sollen alle Hypergraphmorphismen explizit genannt werden, wird geschrieben $H_A \xRightarrow{fgtr,m,d,m^{\dagger},h,h^{\dagger}} H^{\dagger}_{A^{\dagger}}$, verkürzt auch $H_A \xRightarrow{fgtr,m} H^{\dagger}_{A^{\dagger}}$. △

Beispiel 3.7 (Erweiterte GT)
Abb. 3.10 (oben) zeigt, wie erweiterte GTRs dargestellt werden sollen. Zusätzlich zur einfachen GTR werden darin negative und funktionale Anwendungsbedingungen und Aktionen notiert. Im Falle von *fgtr_ex* gibt es zwei negative Anwendungsbedingungen N_1 und N_2. Die negative Anwendungsbedingung N_1 fordert, dass keine Kante vom Typ F zu Knoten u führt, wobei Knoten u und Hyperkante x jeweils zuvor im Rahmen eines Matches (vgl. L) festgelegt werden. Die negative Anwendungsbedingung N_2 legt außerdem fest, dass neben x keine Hyperkante vom Typ E existieren darf, deren Attribut *var2* kleiner als 20 ist. Letztere Zusatzbedingung wird textuell in der Darstellung festgehalten: „Condition (N_2)“. Die funktionale Anwendungsbedingung wird ausschließlich textuell unter „Condition“ erfasst. Der Ausdruck darin legt fest, dass die Attribute *var1* und *var2* von x beide jeweils größer als 10 sein müssen. Die Attributberechnungsregel *acr* wird mittels Aktionen („Actions“) festgelegt, welche ebenfalls rein textuell dargestellt werden. Verwendet werden ausschließlich Zuweisungen, d. h., Modifikationen von Hyperkantenattributen. So soll *var1* von x bspw. verdreifacht und die Attribute *var1* und *var2* der neuen Hyperkante y auf 0 und 20 gesetzt werden.

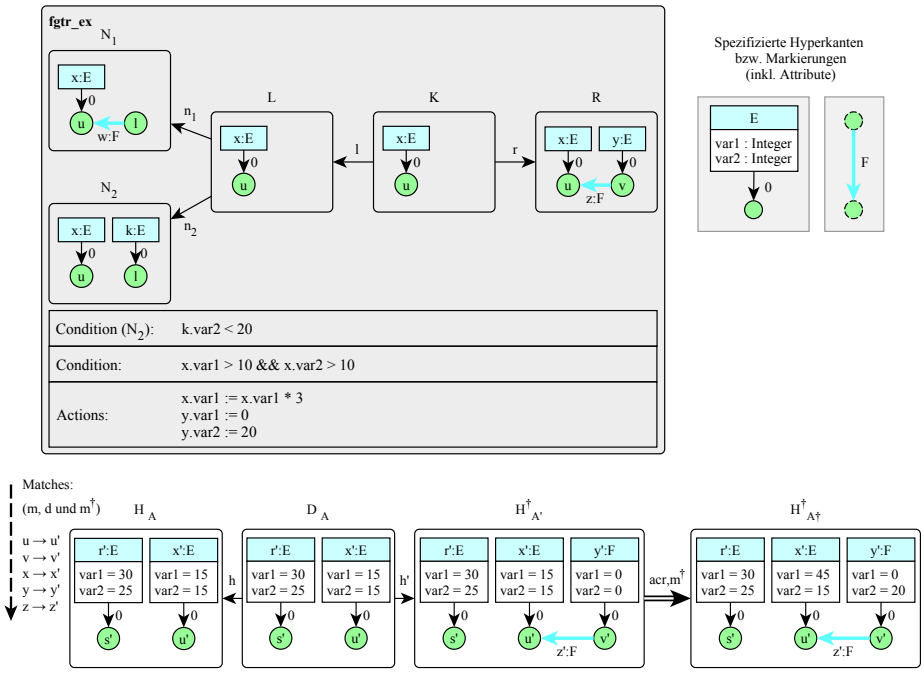


Abbildung 3.10: Erweiterte GTR und GT

Die verwendete textuelle Syntax ist dabei typischen Programmiersprachen nachempfunden. Von dem verwendeten Zuweisungsoperator „:=“ abgesehen, soll in dieser Arbeit eine an Java angelehnte Syntax angenommen werden. Um auf Attribute der Hyperkanten zugreifen zu können, werden die Bezeichner der Hyperkanten verwendet, als wären diese Java-Objekte.

Abb. 3.10 (unten) zeigt die Vorgänge, wenn die beschriebene GTR auf einen konkreten Hypergraphen H_A angewendet wird. Zunächst wird ein passender Match gefunden. Im Beispiel gibt es aufgrund formulierter Bedingungen nur eine Möglichkeit. Danach folgt die einfache GT zu $H^\dagger_{A'}$ (die Hypergraphen H_A, D_A und $H^\dagger_{A'}$ mit den Morphismen h und h' stellen erneut den unteren Teil des DPO-Diagramms dar). Abschließend wird die Attributberechnungsregel *acr* angewendet, wodurch die Attributwerte von A' auf $A^\dagger$ geändert werden. Durch die erweiterte GT resultiert daher der attributierte Hypergraph $H^\dagger_{A^\dagger}$. △

3.2.3 Entfernen von Komponentenhyperkanten

Wie im vorangegangenen Abschnitt erläutert, wurde aufgrund der Klebebedingung festgelegt, dass durch einfache GTRs keine Knoten oder Komponentenhyperkanten entfernt werden sollen (vgl. Definition 3.16, S. 66). Um Komponentenhyperkanten und Knoten trotzdem löschen zu können, wird daher eine spezielle Art von GTR eingeführt, die in Kurzform auch *CHR-GTR* genannt wird. Diese

soll neben der Komponentenhyperkante selbst auch die von der Komponentenhyperkante besuchten Knoten entfernen. Da die Entfernung dieser Knoten zu „hängenden“ Hyperkanten führen kann, müssen auch diese gelöscht werden. Dies können aber ausschließlich Verbindungshyperkanten sein, da jeder dieser Knoten neben der Komponentenhyperkante selbst nur durch Verbindungshyperkanten mit anderen Knoten verbunden werden dürfen (vgl. Abschnitt 3.1.3, S. 55).

Definition 3.22 (CHR-GTR)
Gegeben sei ein Markierungsalphabet $\mathcal{L}$ und die Markierung einer Komponentenhyperkante $del \in \mathcal{CMP}$. Eine GTR $remove_{del}$ über $\mathcal{L}$ heißt *Graphtransformationsregel zum Entfernen einer Komponentenhyperkante (CHR-GTR)* mit Markierung $del \in \mathcal{CMP}$.

$\mathcal{DG}_{\mathcal{L}} = \{remove_{del} \mid del \in \mathcal{CMP}\}$ bezeichne die Menge aller CHR-GTRs über $\mathcal{L}$. △

Zur Spezifikation einer CHR-GTR genügt die Angabe einer Markierung für Komponentenhyperkanten $del \in \mathcal{CMP}$. Durch diese Markierung kann ein elementarer Hypergraph $H_{ehd:del}$ abgeleitet werden, der lediglich die Komponentenhyperkante ehd des Typs del und die von ihr besuchten Knoten enthält. Die folgende Definition macht deutlich, wie die Komponentenhyperkante (mit deren Knoten und ggf. zusätzlichen Verbindungshyperkanten) auf Basis einer CHR-GTR entfernt wird.

Definition 3.23 (Anwendung einer CHR-GTR)
Gegeben seien eine CHR-GTR $remove_{del} \in \mathcal{DG}_{\mathcal{L}}$, ein Ausgangshypergraph $H \in \mathcal{H}_{\mathcal{L}}$ und ein Match $H_{ehd:del} \overset{m}{>\!\!\rightarrow} H$. Der abgeleitete Hypergraph $H^\dagger \subset H$ unter Verwendung von $remove_{del}$ kann konstruiert werden durch

- $E_{H^\dagger} = E_H \setminus \bigcup_{v \in V'_H} \overline{vis}_H(v)$ und
- $V_{H^\dagger} = V_H \setminus V'_H$ mit $V'_H = [vis_H(m_E(ehd))]$,
- $vis_{H^\dagger} = vis_H|_{E_{H^\dagger}}$ und
- $lab_{H^\dagger} = lab_H|_{E_{H^\dagger}}$.

Eine derartige Ableitung wird auch Anwendung einer CHR-GTR oder *Entfernung einer Komponentenhyperkante* genannt, geschrieben

$$H \overset{remove_{del},m}{\dashrightarrow} H^\dagger \quad . \qquad \triangle$$

In Anlehnung an den Darstellungsweisen für die bisher vorgestellten Arten von GTRs wird geschrieben $(p, H) \approx\!\!> remove_{del}$, falls ein Match $H_{del} \overset{m}{>\!\!\rightarrow} H$ existiert, so dass gilt $m \supseteq p$. Es wird geschrieben $H \overset{remove_{del},p}{\rightsquigarrow} H^\dagger$, falls ein solcher Match m für die Anwendung einer CHR-GTR beliebig gewählt werden kann.

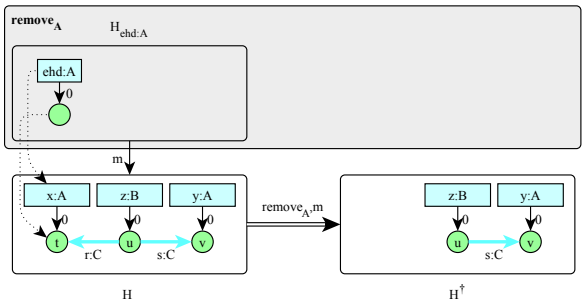


Abbildung 3.11: Löschung einer Komponentenhyperkante

Beispiel 3.8 (Entfernung einer Komponentenhyperkante)
Abb. 3.11 zeigt eine CHR-GTR $remove_A$. Der zugehörige Hypergraph $H_{ehd:A}$ besteht lediglich aus der zu entfernenden Komponentenhyperkante vom Typ A und dem zugehörigen Knoten. Unter der Regel ist ein GT-Beispiel abgebildet. Darin wurde Hyperkante x zusammen mit Knoten t für die Entfernung ausgewählt. Beide Elemente fehlen daher nach der GT. Zusätzlich muss Verbindungshyperkante r entfernt werden, da sie ohne t „hängen“ würde. △

Prinzipiell können CHR-GTRs auch als spezielle GTRs nach SPO-Ansatz betrachtet werden, da bei GTs nach SPO-Ansatz „hängende“ Kanten ebenfalls entfernt werden würden (vgl. [EHK+97]). Um die Definitionen und die Verwendung solcher Regeln schlicht zu halten, wurde auf die Einführung von GTRs nach dem SPO-Ansatz allerdings verzichtet.

3.2.4 Pfad-Anwendungsbedingungen

Als spezielle funktionale Anwendungsbedingung werden in diesem Abschnitt sogenannte Pfad-Anwendungsbedingungen eingeführt. Mit ihrer Hilfe kann vor der Anwendung einer erweiterten GTR geprüft werden, ob sich eine bestimmte, zusammenhängende Folge von Hyperkanten, genannt *Pfad*, im Ausgangshypergraphen befindet, wobei auch Bezug auf den gefundenen Match genommen wird. Die Definition von Pfad-Anwendungsbedingungen basiert auf regulären Ausdrücken, genannt *Pfadausdrücke*, über *Pfadsymbolen* (vgl. [Min01]).

Hilfsdefinitionen
Reguläre Ausdrücke (vgl. [HMU00]) über einem gegebenen Zeichenvorrat Σ sind:

- $\emptyset$ (die leere Menge) und ε (das leeres Wort),
- x für jedes $x \in \Sigma$,
- $a\ b$ (Verkettung), $a \mid b$ (Alternative), wobei a und b reguläre Ausdrücke sind, und
- a^* (Kleenesche Hülle), wobei a ein regulärer Ausdruck ist. Verwendet wird außerdem die Abkürzung $a^+ = a\ (a^*)$. △

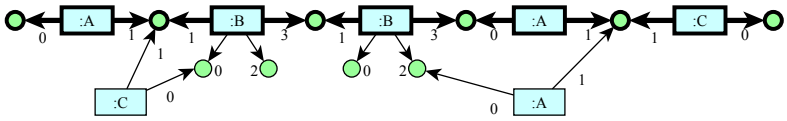


Abbildung 3.12: Beispielpfad in einem Hypergraphen

Definition 3.24 (Pfadsymbol)
Gegeben sei ein Markierungsalphabet $\mathcal{L}$. Ein *Pfadsymbol* über $\mathcal{L}$ ist ein Tripel $psy = (lbl, pt_1, pt_2)$ mit Markierung $lbl \in \mathcal{L}$ und den Tentakeln

$$pt_1, pt_2 \in \{n \in \mathbb{N}_0 \mid n < arity(lbl)\} \ .$$

$\mathcal{PSY}_\mathcal{L}$ bezeichne die endliche Menge aller Pfadsymbole über $\mathcal{L}$. △

Definition 3.25 (Pfadausdruck)
Gegeben sei ein Markierungsalphabet $\mathcal{L}$. Ein *Pfadausdruck pex* über $\mathcal{L}$ ist ein regulärer Ausdruck über Alphabet $\mathcal{PSY}_\mathcal{L}$ ohne $\emptyset$ und ε.
$\mathcal{PEX}_\mathcal{L}$ bezeichne die Menge aller Pfadausdrücke über $\mathcal{L}$. △

Für ein einzelnes Pfadsymbol wird auch die Schreibweise $lbl(pt_1, pt_2)$ verwendet. Beispielsweise bezeichnet in $A(0,1)$ der Buchstabe A die Markierung bzw. den Typ der Hyperkante. Die Zahlen 0 und 1 beziehen sich auf die Tentakel einer entsprechenden Hyperkante. Anschaulich bedeutet dies für einen Pfad, dass er von einem Knoten an Tentakel 0 einer Hyperkante mit Markierung A über die Hyperkante selbst zu den Knoten an Tentakel 1 derselben Hyperkante führen muss.

Mit Hilfe eines Pfadausdrucks kann ein längerer Pfad beschrieben werden, wie z. B. durch $A(0,1)\ B(2,0)$. In diesem Fall muss der Pfad von einem Knoten an Tentakel 0 einer Hyperkante mit Markierung A über die Hyperkante selbst zum Knoten an Tentakel 1 führen. Dieser Knoten muss dem Knoten an Tentakel 2 einer Hyperkante mit Markierung B entsprechen. Der Pfad führt schließlich über diese Hyperkante zum Knoten an Tentakel 0 derselben Hyperkante.

Ein Beispiel für einen Pfadausdruck mit Klammerung und einem Teilpfad, der sich beliebig oft wiederholen kann ist $(A(0,1) \mid B(1,3))^*\ C(1,0)$. In Abb. 3.12 ist ein Pfad abgebildet, der unter Verwendung dieses Ausdrucks in dem dargestellten Hypergraphen gefunden werden kann. Der Pfad selbst wird durch stark umrahmte Hyperkanten und Knoten sowie dicke Tentakelpfeile hervorgehoben.

Definition 3.26 (Pfad-Anwendungsbedingung)
Gegeben seien die LHS $L \in \mathcal{H}_\mathcal{L}$ einer einfachen GTR und das Markierungsalphabet mit Attributen $\mathcal{LV}$. Eine *Pfad-Anwendungsbedingung* über L und $\mathcal{LV}$ ist eine funktionale Anwendungsbedingung $pcon \in \mathcal{CON}_{L,\mathcal{LV}}$, die durch ein Tripel (v_1, pex, v_2) mit einem Startknoten $v_1 \in V_L$, einem Endknoten $v_2 \in V_L \cup \{v^\emptyset\}$, wobei $v^\emptyset \notin V_L$, und einem Pfadausdruck $pex \in \mathcal{PEX}_\mathcal{L}$ festgelegt wird.

Die Funktion *pcon* ordnet einem Paar (m, H_A) den Wert *wahr* zu, falls mittels *pex* ein Pfad in H gefunden werden kann, wobei $m_V(v_1)$ dem Startknoten

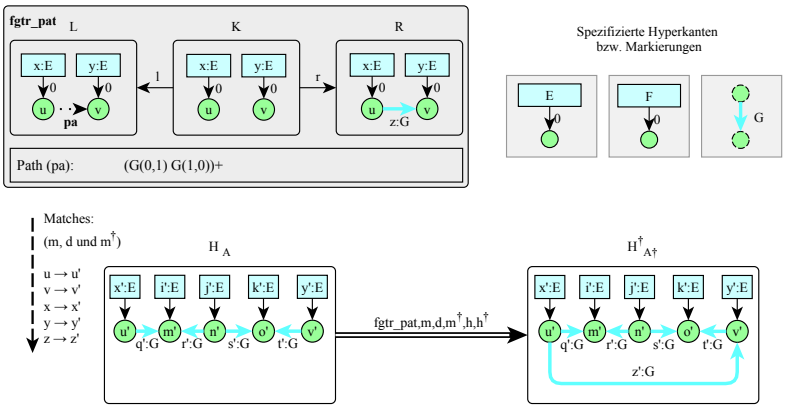


Abbildung 3.13: GTR und GT mit Pfad-Anwendungsbedingung

des Pfads entsprechen muss. Falls $v_2 \neq v^{\emptyset}$, d. h., ein Endknoten vorgegeben wird, muss außerdem $m_V(v_2)$ dem Endknoten des Pfads entsprechen. Ansonsten ordnet *pcon* dem Paar $falsch$ zu. △

Beispiel 3.9 (GT mit Pfad-Anwendungsbedingung)
In Abb. 3.13 (oben) wird eine erweiterte GTR gezeigt, die eine Pfad-Anwendungsbedingung nutzt. Die Pfad-Anwendungsbedingung wird in L der GTR in Form eines dicken Pfeils mit gestrichelter Linie angedeutet. Der Knoten u, bei dem der Pfeil startet, entspricht dem festgelegten Startknoten und v entspricht dem Endknoten der Pfad-Anwendungsbedingung.[7] Der Pfadausdruck trägt den Namen *pa* und wird textuell beschrieben (unterhalb). Er fordert die angegebene Folge von Hyperkanten mit Markierung G, wobei der Pfad abwechselnd von Knoten an Tentakel 0 zu Knoten an Tentakel 1 und danach von dem Knoten an Tentakel 1 zu dem Knoten an Tentakel 0 verlaufen muss. Außerdem muss der Pfad zumindest entlang einem derartigen Paar verlaufen.

Ein solcher Pfad kann im Hypergraphen von Abb. 3.13 (unten) gefunden werden, weshalb die erweiterte GTR anwendbar ist. Der gefundene Pfad verläuft von u' über m', n' und o' zu v'. Es wären allerdings auch andere Matches möglich (bspw. mit $y \to j'$ und $v \to n'$ anstelle von $y \to y'$ und $v \to v'$), so dass der geforderte Pfad gefunden werden kann. △

3.2.5 Graphtransformationsprogramme

Erweiterte GTRs reichen häufig nicht aus, um komplexe GTs durchzuführen. Beispielsweise kann mit den (bisher) vorgestellten GTRs keine Operation zum Schalten einer aktiven Transition in einem B/E-Netz spezifiziert werden (siehe Abschnitt 4.4, S. 117). Sie reichen ebenfalls nicht aus, um alle in den späteren Kapiteln beschriebenen Konzepte umzusetzen.

[7] Falls kein Endknoten spezifiziert wird, zeigt ein solcher Pfeil auf keinen Knoten.

Es werden Kontrollstrukturen benötigt, welche den Ablauf von GTRs steuern können. Es sollte z. B. möglich sein, mehrere GTs sequentiell durchzuführen oder Anwendungsalternativen einzusetzen. In diesem Abschnitt wird daher eine Variante von *Graphtransformationsprogrammen (GTPs)* eingeführt. Andere Ansätze für einen kontrollierten Ablauf von GTs und GTPs werden u. a. in [HMTW95], [PS04] oder [GL07] beschrieben.

Der Vorteil der gewählten GTP-Variante ist, dass sich die später beschriebenen Konzepte gut darauf abbilden lassen. Neben den bereits erwähnten Anwendungssequenzen und -alternativen werden auch Möglichkeiten benötigt, GTs iterativ für alle Hyperkanten einer bestimmten Art durchzuführen und GTs nur durchzuführen, falls eine bestimmte Anzahl an GTRs innerhalb des GTPs mindestens anwendbar ist. Um GTPs mit derartigen Kontrollstrukturen in einem bestimmten Kontext ablaufen lassen zu können, sind sie Bestandteil einer speziellen GTR, genannt *GTP-GTR*. Diese besteht aus einer erweiterten GTR und einem GTP. Bei Anwendung einer GTP-GTR wird zunächst die erweiterte GTR und danach das GTP angewendet. Eine Besonderheit dabei ist, vereinfacht dargestellt, dass sich die Anwendung der GTP auf Hyperkanten und Knoten beziehen kann, die zuvor im Rahmen des Matches für die erweiterte GT verwendet wurden.

In den folgenden Definitionen werden zunächst GTP-GTRs und deren Anwendung formal dargestellt. Da die Definitionen von GTP-GTRs und GTPs allerdings rekursiv aufeinander aufbauen, werden zum vollen Verständnis auch die im Anschluss beschriebenen Definitionen für GTPs benötigt (siehe S. 80ff).

Definition 3.27 (GTP-GTR)
Gegeben sei ein Markierungsalphabet mit Attributen $\mathcal{LV}$. Eine *Graphtransformationsregel mit Graphtransformationsprogramm (GTP-GTR)* über $\mathcal{LV}$ ist ein Paar $pgtr = (fgtr, gtp) = pgtr = (((L, K, R), acr, CONS, NACS), gtp)$ mit

- einer erweiterten GTR $fgtr \in \mathcal{FG}_{\mathcal{LV}}$ und
- einem danach anzuwendenden GTP $gtp \in \mathcal{GTP}_{R,\mathcal{LV}}$.

$\mathcal{PG}_{\mathcal{LV}}$ bezeichne die Menge aller GTP-GTRs über $\mathcal{LV}$. △

Definition 3.28 (Anwendung einer GTP-GTR)
Gegeben seien eine GTP-GTR $pgtr \in \mathcal{PG}_{\mathcal{LV}}$, ein attributierter Ausgangshypergraph $H_A \in \mathcal{H}_{\mathcal{LV}}$ und ein Match $L \xrightarrow{m} H$. Die GTP-GTR $pgtr$ heißt *anwendbar* auf (m, H_A), geschrieben $(m, H_A) \models pgtr$, falls

- $(m, H_A) \models fgtr$ und, falls diese Bedingung erfüllt wird,
- $(m^\dagger, H^\dagger, H^\dagger_{A^\dagger}) \models gtp$, wobei $m^\dagger$ und $H^\dagger_{A^\dagger}$ durch $H_A \xRightarrow{fgtr,m,d,m^\dagger,h,h^\dagger} H^\dagger_{A^\dagger}$ abgeleitet werden.

Falls $(m, H_A) \models pgtr$, kann der abgeleitete, attributierte Hypergraph $H^\ddagger_{A^\ddagger}$ unter Verwendung von $pgtr$ durch die unmittelbar aufeinanderfolgenden Anwendungen von erweiterter GTR und GTP konstruiert werden:

$$H_A \xRightarrow{fgtr,m,d,m^\dagger,h,h^\dagger} H^\dagger_{A^\dagger} \xRightarrow{gtp,m^\dagger,H^\dagger} H^\ddagger_{A^\ddagger} \quad .$$

Eine derartige Ableitung wird auch *Anwendung einer GTP-GTR* genannt, geschrieben $H_A \overset{pgtr,m}{\Longrightarrow} H_A^\ddagger$. △

Bevor GTPs und deren Anwendung formal eingeführt werden, folgt zunächst eine informelle Beschreibung der unterschiedlichen Formen (genannt F1 bis F8), die ein (einzelnes) GTP annehmen kann. Um die Formen verständlicher darstellen zu können, wird ein GTP *gtp* beschrieben, das (direkter) Bestandteil einer GTP-GTR *pgtr* ist und somit nach einer erweiterten GTR *fgtr* angewendet wird.

Die beiden grundlegenden Möglichkeiten von *gtp* sind die Anwendung einer weiteren GTP-GTR (F1) bzw. einer CHR-GTR (F2), in beiden Fällen an dieser Stelle kurz *gtr* genannt. Angewendet wird *gtr* jeweils auf den durch *fgtr* abgeleiteten Hypergraphen. Bei der Anwendung von *gtr* besteht dabei die bereits erwähnte Möglichkeit, Bezug auf den Match zu nehmen, der bei der Anwendung von *fgtr* verwendet wurde. Falls *fgtr* bspw. eine Hyperkante b erzeugt, die einen zuvor gefundenen Knoten a besucht, so können a und b als (Teil-)Match für die Anwendung von *gtr* vorgegeben werden. Um dies zu ermöglichen, muss bei der Spezifikation von *gtp* ein (partieller) Hypergraphmonomorphismus b angegeben werden. Dieser legt bspw. fest (das Beispiel bezieht sich auf F1), welche Elemente der LHS von *gtr* mit welchen Elementen der RHS von *fgtr* im Rahmen des noch zu suchenden und des bereits gefundenen Matches übereinstimmen müssen. Anwendbar ist *gtp* jeweils, wenn auch *gtr* (nur relevant im Fall F1) anwendbar ist und der durch b vorgegebene (Teil-)Match konstruiert werden kann. Letzteres ist für *gtp* im beschriebenen Fall immer möglich, da zwischen der Anwendung von *fgtr* und *gtp* keine Transformation stattfindet. Nachfolgend werden allerdings Möglichkeiten beschrieben, um mehrere GTPs hintereinander auszuführen. Dadurch ist es möglich, dass ein Element entfernt wird, auf das in einer späteren GTP aber Bezug genommen werden soll. Da dieses Element aber nicht mehr Teil des Hypergraphen ist, kann die spätere GTP nicht mehr angewendet werden.

Eine weitere Möglichkeit ist es, durch Anwendung von *gtp* zwei andere GTPs *sequentiell* anzuwenden (F3). Dabei wird vorausgesetzt, dass beide anwendbar sind. Es ist ebenfalls möglich, nur eine von zwei *alternativen* GTPs anzuwenden (F4), wobei die erste anwendbare GTP gewählt wird und mindestens eine der beiden anwendbar sein muss. Außerdem ist es möglich, mittels *gtp* ein anderes GTP *gtp′* anzuwenden. Dies soll allerdings nur geschehen, wenn im Rahmen von *gtp′* eine bestimmte Mindestanzahl von GTRs anwendbar ist (F5). Dabei lässt sich auch die Anzahl 0 festlegen, wodurch *gtp′* zum *optionalen* GTP wird.

Zuletzt kann innerhalb von *gtp* auch ein GTP *gtp′ für alle* Komponentenhyperkanten mit einer bestimmten Markierung ausgeführt werden (F6). In diesem Fall dürfen sich die in *gtp′* enthaltenen GTRs mittels b nicht nur auf die Elemente der RHS von *fgtr* beziehen, sondern auch auf die Komponentenhyperkante, die im jeweiligen Iterationsschritt ausgewählt wurde. Ermöglicht wird dadurch bspw. eine GTP, die an allen Komponentenhyperkanten im Hypergraphen mit einer bestimmten Markierung (einmalig) eine Modifikation durchführt.

Problematisch bei der formalen Darstellung dieses Falls ist jedoch die Tatsache, dass die GTs bei der Iteration sequentiell durchgeführt werden, d. h., es kann

entscheidend sein, in welcher Reihenfolge die Komponentenhyperkanten gewählt werden. In der verwendeten Klasse von Hypergraphen sind die Komponentenhyperkanten allerdings ungeordnet und es gibt auch keinerlei Kriterien, die es erlauben, eine Ordnung bzw. Wahlreihenfolge abzuleiten. Da die Wahlreihenfolge zumeist aber unerheblich ist, wurde auch auf die Einführung eines solchen Kriteriums verzichtet. Um die Anwendung von GTPs dennoch formal beschreiben zu können, sei eine Funktion *msort* definiert, die einer Menge von Matches eine sortierte Folge derselben Matches zuordnet. Wie genau diese Sortierung stattfindet, soll an dieser Stelle aber nicht definiert werden.

Außerdem muss angenommen werden, dass die Matchsuche, die für die Anwendung eines GTPs notwendig sein kann, deterministisch erfolgt.

Definition 3.29 (Graphtransformationsprogramm)
Gegeben seien die RHS $\overline{R} \in \mathcal{H}_{\mathcal{L}}$ einer zuvor anzuwendenden GTP-GTR und das Markierungsalphabet mit Attributen $\mathcal{LV} = (\mathcal{L}, \mathcal{V})$. Ein *Graphtransformationsprogramm (GTP) gtp* über $\overline{R}$ und $\mathcal{LV}$ ist entweder

(F1) ein Paar $(pgtr, b)$, geschrieben $pgtr(b)$ und genannt *Anwendung einer GTP-GTR*, mit einer GTP-GTR $pgtr \in \mathcal{PG}_{\mathcal{LV}}$, wobei $pgtr = (fgtr, gtp')$ und $fgtr = ((L, K, R), \dots)$, und einem partiellen Hypergraphmonomorphismus $L \overset{b}{\rightarrowtail} \overline{R}$,

(F2) ein Paar (del, b), geschrieben $remove_{del}(b)$ und genannt *Anwendung einer CHR-GTR*, mit einer Komponentenhyperkante $del \in \mathcal{CMP}$ und einem Hypergraphmonomorphismus $H_{ehd:del} \overset{b}{\rightarrowtail} \overline{R}$,

(F3) ein Paar (gtp_1, gtp_2), geschrieben $gtp_1 \,;\, gtp_2$ und genannt *GTP-Sequenz*, mit den GTPs gtp_1, gtp_2 über $\overline{R}$ und $\mathcal{LV}$,

(F4) ein Paar (gtp_1, gtp_2), geschrieben $gtp_1 \mid gtp_2$ und genannt *GTP-Alternative*, mit den GTPs gtp_1, gtp_2 über $\overline{R}$ und $\mathcal{LV}$,

(F5) ein Paar (c, gtp'), geschrieben $(gtp')^{\geq c}$ und genannt *GTP mit Mindestanzahl der anwendbaren GTRs*, mit einer Anzahl $c \in \mathbb{N}_0$ der mindestens anwendbaren GTRs und einem GTP gtp' über $\overline{R}$ und $\mathcal{LV}$,

(F6) ein Tripel (e, lbl, gtp'), geschrieben foreach e : lbl do gtp' und genannt *„foreach"-Schleife*, mit einer Komponentenhyperkante e mit der Markierung $lbl \in \mathcal{CMP}$ und einem GTP gtp' über $\overline{R}'$ und $\mathcal{LV}$ mit $\overline{R}' = \overline{R} \cup H_{e:lbl}$ und $e \notin E_{\overline{R}}$,

(F7) das *leere GTP*, geschrieben $gtp^{\emptyset}$, oder

(F8) das *niemals anwendbare GTP*, geschrieben $\overline{gtp}^{\emptyset}$.

$\mathcal{GTP}_{\overline{R},\mathcal{LV}}$ bezeichne die Menge aller GTPs über $\overline{R}$ und $\mathcal{LV}$. △

Definition 3.30 (Anwendung eines GTPs)
Gegeben seien ein GTP $gtp \in \mathcal{GTP}_{\overline{R},\mathcal{LV}}$, ein zuvor verwendeter Match $\overline{R} \overset{\overline{m}^\dagger}{\rightarrowtail} \overline{H}^\dagger$ zusammen mit dem zuvor abgeleiteten Hypergraphen $\overline{H}^\dagger \in \mathcal{H}_\mathcal{L}$ und ein Ausgangshypergraph $H_A \in \mathcal{H}_{\mathcal{LV}}$.

In den Fällen (F1) und (F2) heißt das GTP *gtp anwendbar* auf $(\overline{m}^\dagger, \overline{H}^\dagger, H_A)$, geschrieben $(\overline{m}^\dagger, \overline{H}^\dagger, H_A) \models gtp$, falls im ersten Fall gilt $(p, H_A) \fatsemi pgtr$ bzw. im zweiten Fall gilt $(p, H_A) \fatsemi remove_{del}$. Der partielle Hypergraphmorphismus p mit $dom(p) = dom(b)$ ist dabei gegeben durch $p = c \circ \overline{m}^\dagger \circ b$ mit der Hypergraphinklusion $\overline{H}^\dagger \overset{c}{\hookrightarrow} H$:

$$(F1) \qquad \begin{array}{ccccccc} \overline{R} & \overset{b}{\leftarrowtail} & L & \overset{l}{\hookleftarrow} & K & \overset{r}{\hookrightarrow} & R \\ {\scriptstyle \overline{m}^\dagger}\downarrow & & \downarrow{\scriptstyle p} & & & & \\ \overline{H}^\dagger & \underset{c}{\hookrightarrow} & H & & & & \end{array}$$

$$(F2) \qquad \begin{array}{ccc} \overline{R} & \overset{b}{\leftarrowtail} & H_{ehd:del} \\ {\scriptstyle \overline{m}^\dagger}\downarrow & & \downarrow{\scriptstyle p} \\ \overline{H}^\dagger & \underset{c}{\hookrightarrow} & H \end{array}$$

In den Fällen (F3), (F4), (F5) und (F6) heißt das GTP *gtp* anwendbar, falls

(F3) $(\overline{m}^\dagger, \overline{H}^\dagger, H_A) \models gtp_1$ und, falls diese Bedingung erfüllt wird,
$(\overline{m}^\dagger, \overline{H}^\dagger, H'_{A'}) \models gtp_2$, wobei $H'_{A'}$ abgeleitet wird durch $H_A \overset{gtp_1, \overline{m}^\dagger, \overline{H}^\dagger}{\Longrightarrow} H'_{A'}$,

(F4) $(\overline{m}^\dagger, \overline{H}^\dagger, H_A) \models gtp_1 \vee (\overline{m}^\dagger, \overline{H}^\dagger, H_A) \models gtp_2$,

(F5) $cgts_{gtp'}(\overline{m}^\dagger, \overline{H}^\dagger, H_A) \geq c$, wobei $cgts_{gtp'}$ eine für gtp' definierte Funktion ist, die jedem Tupel $(\overline{m}^\dagger, \overline{H}^\dagger, H_A)$ die *Anzahl* $x \in \mathbb{N}_0$ *der anwendbaren GTRs* zuordnet, und

(F6) $(\overline{m}^\dagger \cup m_1, \overline{H}^\dagger, (H_1, A_1)) \models gtp'$,
wobei $(H_1, A_1) = H_A$,
und, falls diese Bedingung erfüllt wird,
$(\overline{m}^\dagger \cup m_2, \overline{H}^\dagger, (H_2, A_2)) \models gtp'$,
wobei (H_2, A_2) abgeleitet wird durch $(H_1, A_1) \overset{gtp', \overline{m}^\dagger, \overline{H}^\dagger}{\Longrightarrow} (H_2, A_2)$,
und, falls diese Bedingung erfüllt wird,
$\ldots$,
$(\overline{m}^\dagger \cup m_n, \overline{H}^\dagger, (H_n, A_n)) \models gtp'$,
wobei (H_n, A_n) abgeleitet wird durch $(H_{n-1}, A_{n-1}) \overset{gtp', \overline{m}^\dagger, \overline{H}^\dagger}{\Longrightarrow} (H_n, A_n)$.

Verwendet wird dabei eine Folge von Matches $m_1 m_2 \ldots m_n = msort(\mathcal{M})$, wobei $\mathcal{M} = \{(m_{V_1}, m_{E_1}), (m_{V_2}, m_{E_2}), \ldots, (m_{V_n}, m_{E_n})\}$ eine Menge von Matches $H_{e:lbl} \rightarrowtail \overline{H}^\dagger$ ist. Die Elemente der Menge $\mathcal{M}$ sind einerseits gegeben durch $\{m_{E_1}, m_{E_2}, \ldots, m_{E_n}\} = \{\{(e, e')\} \mid e' \in E_{\overline{H}^\dagger} \wedge lab_{\overline{H}^\dagger}(e') = lbl\}$. Andererseits sind die Elemente m_{V_i} für alle $i \in \mathbb{N}^+ \wedge i \leq n$ gegeben durch

$m_{V_i} = \{(v_1, v_1'), (v_2, v_2'), \ldots (v_m, v_m')\}$ mit $v_1 v_2 \ldots v_m = vis_{H_{e:lbl}}(e)$ und $v_1' v_2' \ldots v_m' = vis_{\overline{H}^\dagger}(m_{E_i}(e))$. Falls $n = 0$, so heißt gtp nicht anwendbar.

In den restlichen Fällen (F7) und (F8) heißt das GTP gtp

(F7) immer anwendbar und

(F8) niemals anwendbar.

Falls $(\overline{m}^\dagger, \overline{H}^\dagger, H_A) \models gtp$, kann der abgeleitete, attributierte Hypergraph $H^\dagger_{A^\dagger}$ unter Verwendung des GTPs gtp konstruiert werden. Eine derartige Ableitung wird auch *Anwendung eines GTPs* genannt, geschrieben $H_A \overset{gtp, \overline{m}^\dagger, \overline{H}^\dagger}{\Longrightarrow} H^\dagger_{A^\dagger}$. Der abgeleitete Hypergraph $H^\dagger_{A^\dagger}$ wird dabei je nach Fall konstruiert durch:

(F1) $H_A \overset{pgtr, p}{\rightsquigarrow} H^\dagger_{A^\dagger}$,

(F2) $H_A \overset{removedel, p}{\rightsquigarrow} H^\dagger_{A^\dagger}$,

(F3) $H_A \overset{gtp_1, \overline{m}^\dagger, \overline{H}^\dagger}{\Longrightarrow} H'_{A'} \overset{gtp_2, \overline{m}^\dagger, \overline{H}^\dagger}{\Longrightarrow} H^\dagger_{A^\dagger}$,

(F4) $H_A \overset{gtp_1, \overline{m}^\dagger, \overline{H}^\dagger}{\Longrightarrow} H^\dagger_{A^\dagger}$, falls $(\overline{m}^\dagger, \overline{H}^\dagger, H_A) \models gtp_1$,
$H_A \overset{gtp_2, \overline{m}^\dagger, \overline{H}^\dagger}{\Longrightarrow} H^\dagger_{A^\dagger}$, sonst,

(F5) $H_A \overset{gtp', \overline{m}^\dagger, \overline{H}^\dagger}{\Longrightarrow} H^\dagger_{A^\dagger}$, falls $(\overline{m}^\dagger, \overline{H}^\dagger, H_A) \models gtp'$,
$H_A = H^\dagger_{A^\dagger}$, sonst,

(F6) $H_A \overset{gtp', \overline{m}^\dagger \cup m_1, \overline{H}^\dagger}{\Longrightarrow} (H_2, A_2) \overset{gtp', \overline{m}^\dagger \cup m_2, \overline{H}^\dagger}{\Longrightarrow} \ldots \overset{\ldots}{\Longrightarrow} (H_n, A_n) \overset{gtp', \overline{m}^\dagger \cup m_n, \overline{H}^\dagger}{\Longrightarrow} H^\dagger_{A^\dagger}$

(F7) $H^\dagger_{A^\dagger} = H_A$.

Die Anzahl der im GTP gtp für Tupel $(\overline{m}^\dagger, \overline{H}^\dagger, H_A)$ anwendbaren GTRs $cgts_{gtp}(\overline{m}^\dagger, \overline{H}^\dagger, H_A)$ beträgt 0, falls nicht gilt $(\overline{m}^\dagger, \overline{H}^\dagger, H_A) \models gtp$. Ansonsten ist die Anzahl je nach Fall festgelegt durch

(F1) 1,

(F2) 1,

(F3) $cgts_{gtp_1}(\overline{m}^\dagger, \overline{H}^\dagger, H_A) + cgts_{gtp_2}(\overline{m}^\dagger, \overline{H}^\dagger, H_A)$,

(F4) $cgts_{gtp_1}(\overline{m}^\dagger, \overline{H}^\dagger, H_A)$, falls $(\overline{m}^\dagger, \overline{H}^\dagger, H_A) \models gtp_1$,
$cgts_{gtp_2}(\overline{m}^\dagger, \overline{H}^\dagger, H_A)$, sonst,

(F5) $cgts_{gtp'}(\overline{m}^\dagger, \overline{H}^\dagger, H_A)$,

(F6) $cgts_{gtp'}(\overline{m}^\dagger \cup m_1, \overline{H}^\dagger, (H_1, A_1))$
$+ cgts_{gtp'}(\overline{m}^\dagger \cup m_2, \overline{H}^\dagger, (H_2, A_2))$
$+ \ldots$
$+ cgts_{gtp'}(\overline{m}^\dagger \cup m_n, \overline{H}^\dagger, (H_n, A_n))$,

(F7) 0. △

In GTP-Ausdrücken ist das Setzen von Klammern zur Gruppierung erlaubt. Folgen von GTP-Sequenzen und GTP-Alternativen sind linksassoziativ zu interpretieren. Anstelle von GTP-GTRs (F1) dürfen auch erweiterte oder einfache GTRs verwendet werden. Formal betrachtet müssen diese dann entsprechend zu einer GTP-GTR erweitert werden (unter naheliegender Verwendung von $acr^{\emptyset}$, $gtp^{\emptyset}$ und leeren Mengen für die Anwendungsbedingungen).

Die Fälle (F7) und (F8) werden außerdem angenommen für

(F7) $gtp_1 \ ; \ \ldots \ ; \ gtp_n$, falls $n = 0$, oder

(F8) $gtp_1 \mid \ldots \mid gtp_n$, falls $n = 0$.

Aufgrund des verwendeten Typsystems mit Vererbung wird zur Spezifikation von $L \overset{b}{\rightarrowtail} \overline{R}$ bzw. $H_{ehd:del} \overset{b}{\rightarrowtail} \overline{R}$ nicht immer ein Hypergraphmonomorphismus nach Definition verwendet. Es ist bspw. möglich, dass eine Hyperkante in L gegenüber der zugeordneten Hyperkante in $\overline{R}$ einen abgeleiteten Typen oder Basistypen verwendet (vgl. Abschnitt 3.2.1, S. 69). In diesem Fall ist zur Anwendung der GTR innerhalb eines GTPs zusätzlich wichtig, dass die zugeordneten Hyperkanten in H eine passende Markierung besitzen.

Obwohl die Anwendung eines GTPs oder einer GTP-GTR technisch gesehen aus mehreren Schritten besteht und auch sequentiell durchgeführt wird, eigentlich genau wie bei einer erweiterten GTR, soll die komplette GT als atomare Operation betrachtet werden. Das bedeutet, dass sie „zum selben Zeitpunkt“ entweder komplett oder gar nicht angewendet wird und während ihrer Anwendung keine andere GT parallel abläuft oder eine andere Operation ihren Ablauf beeinflusst.

Beispiel 3.10 (GT mit GTP)
In Abb. 3.14 (oben) wird eine GTP-GTR *pgtr_ex* dargestellt. Der obere Teil von *pgtr_ex* entspricht einer erweiterten GTR, im Beispiel auch *fgtr_ex* bezeichnet, während der untere Teil das zugehörige GTP *gtp_ex* zeigt. Rechts daneben sind die beiden einfachen GTRs *egtr1* und *egtr2* abgebildet, die innerhalb von *gtp_ex* benutzt werden.

Durch *gtp_ex* wird festgelegt, dass nach der Anwendung von *fgtr_ex* sowohl *egtr1* als auch *egtr2* ausgeführt werden sollen. Allerdings ist *egtr1* innerhalb des GTP-Ausdrucks mit $^{\geq 0}$ gekennzeichnet, wodurch die Anwendung von *egtr1* optional ist. Das GTP-GTR *pgtr_ex* ist somit anwendbar, falls *fgtr_ex* und *gtp_ex* anwendbar sind, wobei *gtp_ex* anwendbar ist, falls *egtr2* anwendbar ist (unabhängig von *egtr1*).

Für die Anwendung von *egtr1* und *egtr2* (im Rahmen von *pgtr_ex*) müssen die in *gtp_ex* angegebenen Hypergraphmorphismen berücksichtigt werden. Diese Hypergraphmorphismen sind neben der textuellen Syntax auch in Form von gestrichelten Linien in der Abbildung eingezeichnet, damit die verwendete Syntax besser verstanden werden kann. Der angegebene Hypergraphmorphismus für *egtr1* hat in diesem Beispiel folgende Bedeutung: die Hyperkanten, die im Rahmen der Anwendung von *fgtr_ex* den Hyperkanten x und y (aus *fgtr_ex*) zugeordnet wurden, müssen für die Anwendung von *egtr1* den Hyperkanten c und d (aus *egtr1*) zugeordnet werden. Dadurch ist ein Teilmatch für die Anwen-

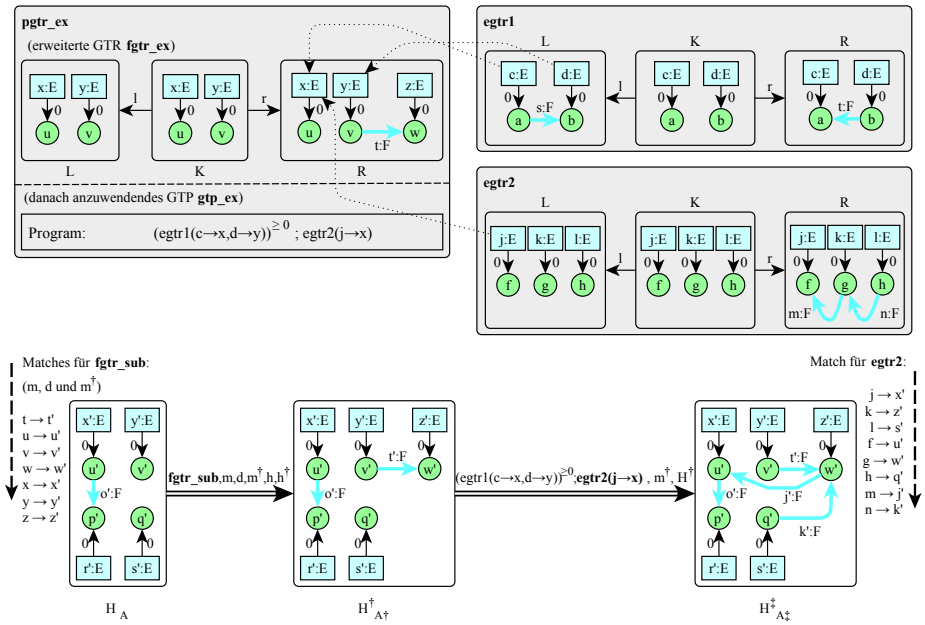


Abbildung 3.14: GTP-GTR und GT

dung vorgegeben. Der restliche Match zur Anwendung von *egtr1* kann beliebig gewählt werden.

Abb. 3.14 (unten) zeigt eine exemplarische Anwendung von *pgtr_ex*. Dabei erfolgt die Anwendung von *fgtr_ex* vor der Anwendung von *gtp_ex*. Bei der Anwendung von *gtp_ex* wird kein passender Match für *egtr1* gefunden, da zwischen den Hyperkanten x' und y' (ausgewählt aufgrund des vorgegebenen Teilmatches, siehe oben) keine Verbindungshyperkante mit Markierung F existiert. Daher wird lediglich *egtr2* angewendet. Bei der Suche nach einem Match zur Anwendung von *egtr2* ist nur der Teilmatch $j \to x'$ vorgegeben. Die restlichen Teilmatches sind beliebig gewählt worden, z. B. $k \to z'$ oder $l \to s'$. △

Beispiel 3.11 (GTP mit „foreach“)
Abb. 3.15 (oben) zeigt das GTP-GTR *pgtr_for*. Die darin enthaltene erweiterte GTR *fgtr_for* ist „leer“, d. h., weder LHS noch RHS enthalten Elemente, wodurch sie zwar anwendbar ist, bei der Anwendung aber keine GT durchgeführt wird. Danach soll jedoch das GTP *gtp_for* angewendet werden.

Durch dieses GTP wird für jede Komponentenhyperkante mit Markierung E eine einfache GTR *egtr3* angewendet. Bei jedem Iterationsschritt soll dabei die spezifizierte Iterations-Hyperkante i einer anderen, passenden Komponentenhyperkante im Ausgangshypergraphen zugeordnet werden, wobei angenommen wird, dass diese Iterations-Hyperkante Teil der RHS R (erweitert auf R') von *fgtr_for*

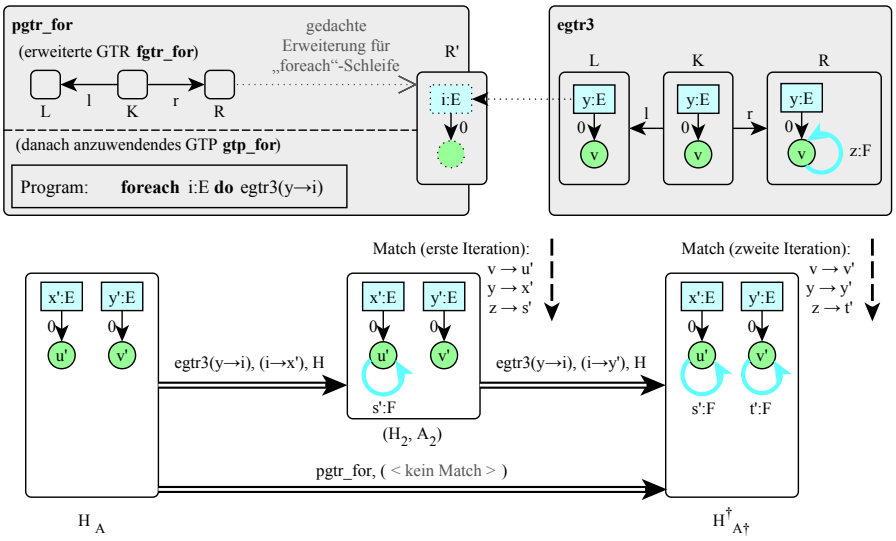


Abbildung 3.15: GTP und GT mit *foreach*

ist.[8] Dadurch kann sich die Anwendung von *egtr3* im Rahmen von *gtp_for* auf diese Iterations-Hyperkante beziehen. Im Beispiel wird Komponentenhyperkante y von *egtr3* der Iterations-Hyperkante i zugeordnet, d. h., die bei einem Iterationsschritt gewählte Komponentenhyperkante ist für die Anwendung von *egtr3* als Teilmatch bzgl. y gesetzt.

In Abb. 3.15 (unten) wird eine beispielhafte GT dargestellt. In H_A existieren zwei Komponentenhyperkanten mit Markierung E. Durch Anwendung von *gtp_for* wird *egtr3* für jede dieser Komponentenhyperkanten angewendet. Im Beispiel wird bei der ersten Iteration der Teilmatch $y \to x'$ vorgegeben, bei der zweiten Iteration $y \to y'$. Die Reihenfolge wurde dabei wirkürlich gewählt, da kein Kriterium zur Sortierung der Matches existiert. △

3.2.6 Graphtransformationssysteme

Die bisherigen Definitionen sind für die Beschreibung des Animationsansatzes wichtig. Für eine Umsetzung entsprechender Editoren wird allerdings auch ein konkretes Graphtransformationssystem (GTS) benötigt. Aus diesem Grund sollen einige wichtige Vertreter genannt und kurz charakterisiert werden. Neben der Einsatzmöglichkeit im Rahmen von Diagrammeditoren werden viele der vorgestellten GTSe primär aber für andere Zwecke eingesetzt, z. B. für Simulation, Modelltransformation oder -verifikation. Übersichten bieten auch Arbeiten wie [FMRS07] und [TEG+05].

[8] Die Iterations-Hyperkante i inkl. Knoten werden deshalb mit gestrichelter Linie gezeichnet, da sie bei Matchsuche mit anderen Hyperkanten bzw. Knoten aus der RHS R von *fgtr_for* (in diesem Fall existieren keine) zusammenfallen dürfen.

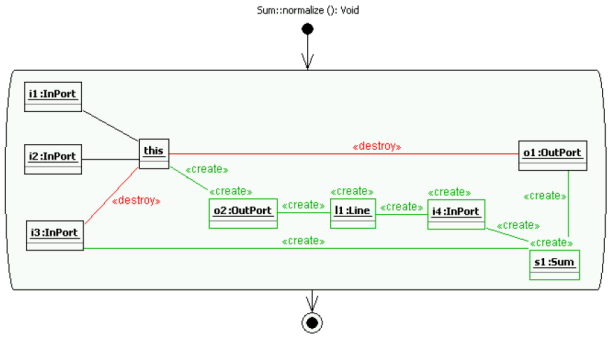


Abbildung 3.16: Story-Diagramm (aus [AKRS06])

An dieser Stelle sollte erwähnt werden, dass auch GTSe verwendet werden können, die keine direkte Unterstützung für Hypergraphen bieten, denn die beschriebenen Konzepte lassen sich auf gewöhnliche, gerichtete Graphen übertragen. Anstatt Komponentenhyperkanten können bspw. speziell markierte Komponentenknoten verwendet werden, denen wiederum Konnektorknoten zugeordnet werden. In dieser Form wird das Konzept auch in [Min06a] beschrieben.

PROGRES

Ein sehr ausdrucksstarkes GTS ist PROGRES [Sch96]. Das System gehört zu den Vertretern der (textuellen) Graphersetzungssprachen und erinnert durch die Unterstützung von Kontrollstrukturen wie Schleifen, Verzweigungen, Funktionen etc. an traditionelle, imperative Programmiersprachen. Das System unterstützt viele objektorientierte Konzepte. Den unterschiedlichen Knotentypen können bspw. Attribute und Methoden zugewiesen werden (ähnlich wie den Klassen in typischen objektorientierten Sprachen). Standardmäßig sucht das GTS nach injektiven Matches. Mit Hilfe von „folding clauses“ können allerdings Elemente bestimmt werden, die im Match zusammenfallen dürfen. Negative Anwendungsbedingungen oder Pfadausdrücke werden von PROGRES ebenfalls unterstützt.

Fujaba

Ähnlich wie PROGRES ist Fujaba [FNTZ00] ein System zur programmierten Graphersetzung. Dabei unterstützt Fujaba prinzipiell alle Möglichkeiten, die auch für PROGRES genannt wurden: Pfadausdrücke, Kontrollstrukturen, Abschalten der injektiven Matchsuche via „maybe“-Konstrukt etc. Ein wesentlicher Unterschied ist allerdings, dass der Ablauf in Fujaba visuell modelliert wird. Eingesetzt werden dabei sogenannte *Story-Diagramme*, die eine Mischung aus UML-Kollaborationsdiagrammen und UML-Aktivitätsdiagrammen darstellen. Sie bestehen zum Teil aus Graphmustern bzw. Story-Patterns, die in den Knoten des Diagramms abgebildet werden. Diese integrieren jeweils LHS und RHS einer GTR. Knoten und Kanten, die hinzugefügt oder entfernt werden sollen, werden

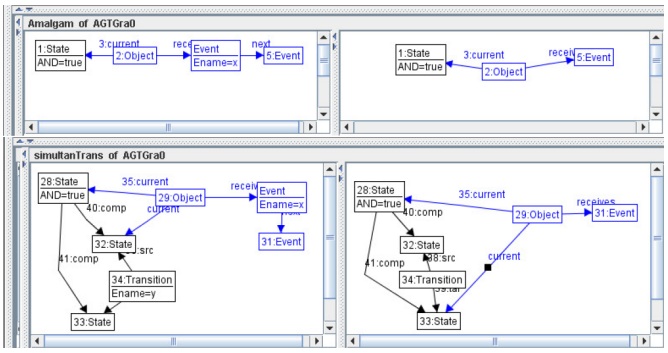


Abbildung 3.17: Screenshot – AGG (aus [Erm06])

mit *«create»* bzw. *«destroy»* gekennzeichnet (siehe Abb. 3.16). Der Graph entspricht dabei einem Netz miteinander verknüpfter Java-Objekte, was auch die ursprüngliche Verwendung von Fujaba widerspiegelt, nämlich die Generierung von Java-Code. Inzwischen existieren bereits sehr viele Varianten und Einsatzgebiete für Fujaba, bspw. Fujaba Real-Time [BG03] oder der Einsatz im MOFLON-Framework zur Transformation von Modellen [AKRS06].

AGG

Im Gegensatz zu GTSen mit programmierter Graphersetzung (wie PROGRES und Fujaba), folgt AGG [ERT99] strikt dem algebraischem Ansatz. Genauer gesagt, müssen GTRs nach dem SPO-Ansatz (inkl. negativer Anwendungsbedingungen) erstellt werden (siehe Abb. 3.17). Dabei werden attributierte Graphen unterstützt, die außerdem durch einen zu spezifizierenden Typgraphen typisiert sind. Für Attributbedingungen oder -modifikationen können beliebige Java-Ausdrücke verwendet werden. Injektives Matching kann nur global an- und abgeschaltet werden. Es können zwar einfache Ausführungsschleifen (die Anwendung einer GTR so lange wie möglich) durch einen „layer“-Mechanismus spezifiziert werden, doch es mangelt AGG an der Möglichkeit typische Kontrollstrukturen einzusetzen [FMRS07]. Auf der anderen Seite wurden dem System theoretisch fundierte Konzepte wie bspw. *amalgamierte GTs* hinzugefügt, mit denen auch GTs umgesetzt werden können, die ansonsten nicht realisierbar wären. Zur Umsetzung von GTRs für die Simulation von Petri-Netzen war dieses Konzept notwendig [Erm06]. Eingesetzt wird AGG u. a. in GenGED und TIGER (vgl. Abschnitt 3.4, S. 96)

Weitere Graphtransformationssysteme

Andere häufig miteinander verglichene GTSe sind VIATRA2, VMTS, GReAT oder GrGen. VIATRA2 [VP04] unterstützt generische GTs, d. h., für GTRs können Typparameter festgelegt werden. Die Steuerung von Regeln (Kontrollfluss) kann mittels abstrakten Zustandsmaschinen (siehe [BCR00]) spezifiziert werden. Ähnlich wie VIATRA2 wird auch VMTS [LLMC05] primär zur Transformation von

Modellen eingesetzt. Der Kontrollfluss kann dabei durch Aktivitätsdiagramme bestimmt werden, die durch Stereotypen erweitert werden (vgl. [TEG+05]). GReAT [BNBK06] ist Bestandteil des GME [LMB+01] (siehe auch Abschnitt 3.4, S. 97). Die darin erstellbaren Regeln basieren auf UML-Klassendiagrammen. Sowohl GTRs als auch Kontrollfluss können dabei visuell erstellt werden. Negative Anwendungsbedingungen werden jedoch nur indirekt unterstützt, indem Verbindungen mit einer Kardinalität von 0 genutzt werden. GrGen [GBG+06] zeichnet sich schließlich durch seine Geschwindigkeit bei der Suche nach Matches aus.

Zusätzlich existieren einige GTSe, die in Frameworks für Editoren eingebaut sind und keine unabhängige Komponente darstellen. So wird bspw. AToM³ [LV02] (siehe Abschnitt 3.4, S. 96) manchmal als GTS aufgeführt. Auch im DiaMeta-Framework ist ein GTS eingebaut.

3.3 Das DiaMeta-System

Der in dieser Arbeit präsentierte Ansatz wurde im Rahmen eines Tools namens DiaMeta implementiert und getestet. DiaMeta ist eine Toolsammlung und gleichzeitig Framework zur Generierung und Implementierung von graphbasierten Editoren für Diagrammsprachen. DiaMeta zählt dabei zu den sogenannten *Meta-Tools*, da es für die Erzeugung von Editoren, die ebenfalls als Tools gelten, genutzt werden kann.

Einige Grundkonzepte von DiaMeta haben einen wichtigen Ausgangspunkt für Analysen und die Entwicklung von Animationsansätzen im Rahmen dieser Arbeit dargestellt. Die Wahl fiel dabei auf DiaMeta, da die von diesem Meta-Tool erzeugten Editoren und deren Diagramme auf der beschriebenen Klasse von Hypergraphen basieren und die damit verbundenen Vorteile bietet (vgl. Abschnitt 3.1.1, S. 50). Diagrammmanipulationen (strukturiertes Editieren) können bspw. mittels GTRs festgelegt werden. Alle im vorherigen Abschnitt beschriebenen Konzepte werden dabei durch das in DiaMeta integrierte GTS in leicht angepasster Form unterstützt. Im Gegensatz zu den in Abschnitt 3.2.6, S. 85, vorgestellten GTSen können Hypergraphen direkt verwendet werden. Der modulare Aufbau von DiaMeta vereinfachte zudem dessen Erweiterung, um experimentelle Animationsansätze zu integrieren. Außerdem sind DiaMeta und damit verwandte Projekte Gegenstand zahlreicher Publikationen, wie z. B. die bereits genannten [Min01], [Min02], [Min04], [Min06a], [MMM08], [Maz10] oder [Bri10].

Daher wurde DiaMeta durch die in Kapitel 4 vorgestellten Ansätze erweitert und zu einem Meta-Tool zur Generierung von Editoren für interaktive animierte Sprachen ausgebaut. Zuvor beschreibt dieser Abschnitt die grundlegenden Konzepte von DiaMeta. Abschnitt 3.3.1 zeigt die Architektur generierter DiaMeta-Editoren und skizziert einige der darin enthaltenen Module und Mechanismen. Abschnitt 3.3.2 legt dar, wie Editoren spezifiziert und generiert werden.

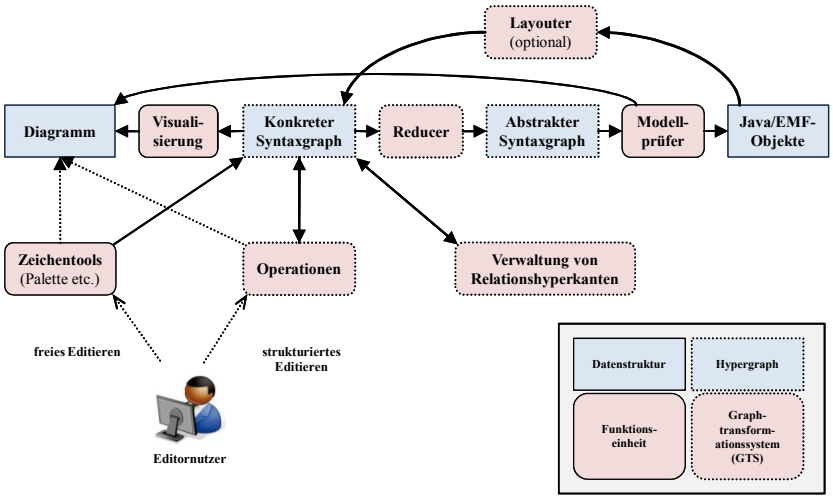


Abbildung 3.18: Architektur eines auf DiaMeta basierenden Editors

3.3.1 Architektur erzeugter Diagrammeditoren

Eine Übersicht des Aufbaus und der Funktionsweise eines auf DiaMeta basierenden Editors wird in Abb. 3.18 gezeigt. Darin werden Datenstrukturen in blauen Kästen und Funktionseinheiten in roten Kästen mit abgerundeten Ecken dargestellt. Mit gestricheltem Rahmen werden Graphen (Datenstrukturen) und GTSe (Funktionseinheiten) gekennzeichnet. Ein Informationsfluss wird mit einem normalen Pfeil dargestellt.

Aktionsmöglichkeiten des Editornutzers

Der Editornutzer kann den Editor auf zwei unterschiedliche Arten bedienen: durch freies Editieren und durch strukturiertes Editieren. Für das freie Editieren kann er die *Zeichentools* des Editors benutzen. Diese bieten bspw. eine Werkzeugpalette, in der verfügbare Diagrammkomponenten gewählt werden, um diese im Diagramm erstellen und beliebig platzieren zu können. Auch das Entfernen von Diagrammkomponenten oder das Ändern von Komponentenattributen ist je nach Diagrammkomponente möglich. Für das strukturierte Editieren stehen dem Editornutzer eine Reihe von *Operationen* zur Verfügung (Knöpfe in der GUI). Für bestimmte Operationen muss der Benutzer zuvor eine Diagrammkomponente wählen, die dann als Kontext für die Operation gilt.

Konkreter Syntaxgraph und Diagramm

Das im Editor gezeichnete *Diagramm* ist lediglich die *Visualisierung* des sogenannten *konkreten Syntaxgraphen (KSG)*[9]. Es soll an dieser Stelle vereinfachend

[9] In einigen Publikationen über DiaMeta auch schlicht Graphmodell oder „Spatial Relationship (Hyper-)Graph“ genannt.

angenommen werden, dass die zuvor beschriebenen Aktionsmöglichkeiten des Editornutzers zur direkten Modifikation des KSGs führen. Derartige Änderungen spiegeln sich danach sofort im Diagramm wider. Ein gestrichelter Pfeil in Abb. 3.18 soll darauf hinweisen, dass der Editornutzer eigentlich mit den Diagrammkomponenten interagiert.

Hinter den vom Editornutzer wählbaren Operationen verbergen sich GTPs, die ähnliche Möglichkeiten bieten, wie in Abschnitt 3.2.5, S. 77, vorgestellt. Wird eine Operation vom Editornutzer gewählt, wendet das im DiaMeta-Framework integrierte GTS ein zugehöriges GTP an, falls möglich. Die GT wird „inplace“ am KSG durchgeführt.

Nach jeder Änderung des KSGs wird außerdem eine Funktionseinheit aktiviert, die für die *Verwaltung der Relationshyperkanten* zuständig ist (vgl. Abschnitt 3.1.3, S. 56). Diese prüft für spezifizierte Paare von Konnektoren, ob diese in Relation stehen. In DiaMeta bedeutet dies, dass sich die Bereiche der Konnektoren überlappen und eine evtl. zusätzlich spezifizierte Bedingung erfüllt wird. Entsprechend werden Relationshyperkanten zum KSG hinzugefügt bzw. entfernt.

Abstrakter Syntaxgraph und Graphreduktion

Der KSG kann unter Umständen selbst für kleinere Diagramme eine beachtliche Größe erreichen. Er enthält mit seinen Relationshyperkanten – aber auch anderen räumlichen Informationen und Attributen – viele Elemente, die für die Bedeutung bzw. die abstrakte Repräsentation des Diagramms redundant sind. Daher wird der KSG nach jeder Änderung durch den sogenannten *Reducer* reduziert. Der dadurch entstehende *abstrakte Syntaxgraph (ASG)* ist normalerweise ein Graph mit anderen Markierungen als der KSG. Stellt man dieses Verfahren dem klassischem Compilerbau gegenüber, so kann man die Graphreduktion mit der lexikalischen Analyse des Codes vergleichen, bei der aus dem Eingabezeichenstrom Tokens erzeugt werden (vgl. [Min01]).

Beispielsweise sind die Pfeile der B/E-Netze, die als Diagrammkomponenten modelliert werden können, für die abstrakte Repräsentation des Netzes unwichtig. Entscheidend ist lediglich, dass Transitionen ihren Vor- und Nachbereich kennen (vgl. Definition 2.2, S. 34). Daher müssen Pfeile als Komponentenhyperkanten im ASG nicht mehr vorkommen. Diese Reduktion wird in [Min04] genauer beschrieben.

Nach der Reduktion wird der entstandene ASG vom *Modellprüfer* gegen die Spezifikation der abstrakten Sprachsyntax geprüft (ein Metamodell in DiaMeta). Dies kann mit der syntaktischen Analyse (Parsing) im klassischen Compilerbau verglichen werden. Die Analyse selbst erfolgt in DiaMeta durch Lösung eines Constraint-Satisfaction-Problems (CSPs). Dabei versucht der Modellprüfer einen maximalen Teilgraph des ASGs zu finden, welcher den durch die Spezifikation gegebenen Bedingungen entspricht. Das Verfahren wird in [Min06a] und [Min06b] beschrieben. Details zur Spezifikation der abstrakten Sprachsyntax sind in Abschnitt 3.3.2, S. 93, zu finden.

Das Ergebnis der Analyse kann als Feedback im Diagramm angezeigt werden. Wird durch die Reduktion also kein zur Spezifikation passender Graph erzeugt, können entsprechende Diagrammkomponenten eingefärbt werden. Besitzt ein Alligator in Alligator Eggs bspw. keine Elemente, die er beschützt, so ist dies gemäß Spezifikation ungültig und der Alligator wird rot umrahmt.

Nach erfolgter Analyse, d. h. Lösung des CSPs, wird eine Objektstruktur aus *Java-Objekten* angelegt, die im Wesentlichen dem ASG entspricht.[10] Teilweise können zur Erstellung der Java-Objekte auch Informationen aus dem KSG wiederverwendet werden (z. B. die darin enthaltenen Attribute). Die Java-Objekte enthalten auch die Informationen, welche Komponentenhyperkanten indirekt zur Entstehung der Java-Objekte geführt haben. Insgesamt stellen die Java-Objekte das abstrakte Modell des Diagramms dar und können für eine weitere Verarbeitung (z. B. Codegenerierung) genutzt werden.

Beispiel 3.12 (Graphreduktion in Alligator Eggs)
Der Reduktionsprozess kann auch für den Alligator Eggs-Editor sinnvoll verwendet werden. Abb. 3.19 zeigt ein Alligator Eggs-Diagramm (ganz links) und den zugehörigen KSG (halb links). Gemäß Spezifikation enthält dieser KSG einige Relationshyperkanten. So sollen *protects*-Relationshyperkanten von einem hungrigen oder alten Alligator zu allen Diagrammkomponenten unterhalb des Alligators erzeugt werden. Daher wurde für den Knoten an Tentakel 1 eines hungrigen Alligators (*Alligator*) ein Konnektor spezifiziert, dessen Bereich den gesamten Bereich unterhalb des hungrigen Alligators umfasst. Den Knoten an Tentakel 0 aller Diagrammkomponenten wurden Konnektoren zugeordnet, die sich jeweils auf den visuellen Bereich der entsprechenden Diagrammkomponente beziehen. Fast analog sollen Elemente rechts von einem hungrigen Alligator mit diesem durch *eats*-Relationshyperkanten verbunden werden.

Zusätzlich existiert im KSG die *Root*-Komponentenhyperkante. Dieses einmalige Element ist eine unsichtbare Diagrammkomponente und kann als „virtueller Alligator“ gesehen werden, der alle anderen Diagrammkomponenten beschützt. Dementsprechend sollen auch in diesem Fall *protects*-Relationshyperkanten von *Root* zu allen anderen Komponentenhyperkanten erstellt werden. Die *Root*-Komponentenhyperkante wurde eingeführt, um Alligator Eggs-Diagramme (bzw. deren abstrakte Repräsentation) wie gerichtete Bäume verarbeiten zu können: die oberste Zeile des Diagramms entspricht dann den Kindern des Wurzelknotens *Root*.

Durch die Graphreduktion entsteht der ASG (Abb. 3.19, halb rechts). Darin sind einige Informationen über (räumliche) Relationen nicht mehr vorhanden und auch die Anzahl der Knoten wurde reduziert. Nach der Analyse des ASGs entstehen
Hierarchie problemlos festgestellt werden kann, welche Elemente von einem Alligator beschützt oder gefressen werden. Letzteres kann über die geordnete Liste *protects* des Eltern-Objekts herausgefunden werden. Eine solche Abfrage kann bspw. auch in Form einer Methode *getEats* für *AAlligator*-Objekte zur Verfügung

[10] Allerdings können bspw. die Klassen der Objektstruktur durch Lösung des CSPs genauer bestimmt werden als durch die Markierungen im ASG. Ein Beispiel ist in [Min06b] zu finden.

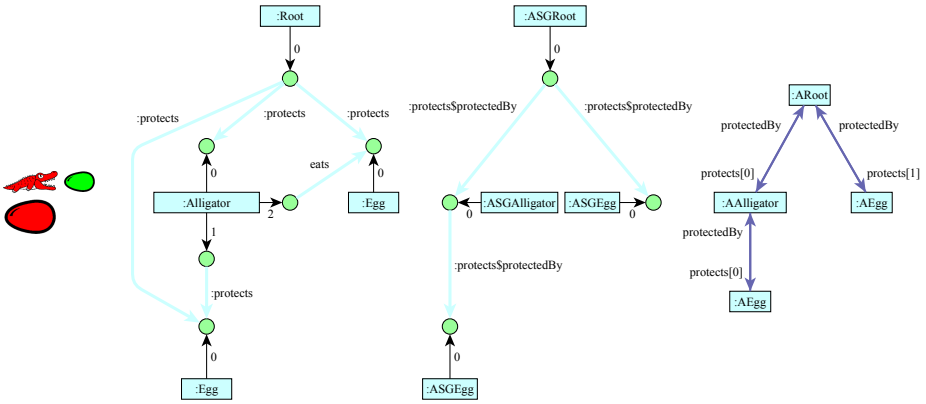


Abbildung 3.19: (a) Diagramm, (b) KSG, (c) ASG, (d) Java/EMF-Objekte

gestellt werden. Die erwähnte Ordnung innerhalb von *protects*-Listen entsteht durch nachträgliche Analyse der x-Koordinaten der Diagrammkomponenten (bzw. Komponentenhyperkanten im KSG). △

Diagrammlayout

Häufig unterstützen Editoren auch das automatische Layout von erstellten Diagrammen. Auch im DiaMeta-Framework kann eine solche Unterstützung optional hinzugefügt werden. Es stehen zwar einige generische Algorithmen und inzwischen auch alternative Ansätze in DiaMeta (siehe [MM10, Mai12]) zur Verfügung, in komplexeren Fällen muss ein Layoutalgorithmus in der Regel aber händisch programmiert werden.

Der (programmierte) *Layouter* ändert normalerweise die Position und Größe der Diagrammkomponenten, indem er die Attribute der Komponentenhyperkanten modifiziert. Er kann dabei automatisch arbeiten, d. h. nach jeder Änderung des KSGs, oder er wird explizit auf Anforderung gestartet. Zur Berechnung des richtigen Layouts kann der Layouter Informationen nutzen, die durch die Java-Objekte zur Verfügung gestellt werden. Auf Basis der abstrakten Struktur soll dann das optimale Layout berechnet und zugewiesen werden. Einige Details verwendbarer Ansätze in Bezug auf DiaMeta können in [MMM08] nachgelesen werden.

Auch für den Editor von Alligator Eggs wird ein *Layouter* eingesetzt. Der notwendige Algorithmus lässt sich vergleichsweise einfach erstellen, da aufgrund des strukturellen Zusammenhangs der Java-Objekte (vgl. Beispiel 3.12) ein typischer Algorithmus für das Layout von Bäumen verwendet werden kann. So kann der Layouter nach Anwendung der Fressregel (eine verfügbare Operation bzw. GTR) benutzt werden, um alle Elemente wieder korrekt anzuordnen. Da der Layouter diese Aufgabe übernimmt, muss die GT selbst nicht die Berechnung eines Layouts übernehmen.[11] Stattdessen muss die GT lediglich Mehrzweck-

[11] Die Berechnung eines richtigen Layouts mittels GTRs wäre schwierig.

Hyperkanten erstellen, so dass durch Reduktion und Analyse ein Baum abgeleitet werden kann, der dann vom Layouter zur Erstellung des passenden Layouts genutzt wird.

Dieses Verfahren wird auch im animierten Alligator Eggs-Editor eingesetzt. Allerdings verändert der dafür programmierte Layouter die Lage der Diagrammkomponenten nicht direkt, sondern berechnet lediglich Zielkoordinaten und -größen, die dann im Rahmen der Animationen verwendet werden. Unter Einsatz des Animationsansatzes, der in dieser Arbeit präsentiert wird, kann diese Vorgehensweise generell genutzt werden, um Layoutänderungen animiert darzustellen.

3.3.2 Sprachspezifikation und Generierungsprozess

Editoren für die beschriebene Architektur können größtenteils auf Basis einer Sprach- und Editorspezifikation generiert werden. Ein weiterer Teil wird durch das DiaMeta-Framework gebildet, welches Teile der GUI, das GTS, die Datenstrukturen, den Modellprüfer und einige weitere Komponenten enthält. Für einen lauffähigen Editor sind danach meist nur noch wenige händisch programmierte Codezeilen notwendig, z. B. für den optionalen Layouter.

Die Sprach- und Editorspezifikation wird im Falle von DiaMeta in zwei unterschiedlichen Schritten durchgeführt. Grob können diese Schritte in die Spezifikation der abstrakten Sprachsyntax und die Spezifikation der konkreten Sprachsyntax (inkl. Editoreigenschaften) eingeteilt werden.

Spezifikation der abstrakten Sprachsyntax

Die abstrakte Sprachsyntax für DiaMeta-Editoren wird mittels Metamodell spezifiziert (vgl. [Min06a]). Die Idee dabei ist, dass gültige Diagramme während der Analysephase (Reduktion des KSGs und anschließende Prüfung, vgl. Abschnitt 3.3.1, S. 90) auf ein Modell dieses Metamodells abgebildet werden. Dieses Modell entspricht somit der abstrakten Repräsentation des Diagramms.

Zur Erstellung des Metamodells können die Meta Object Facility (MOF) der Object Management Group (OMG) (siehe [MOF11]) oder das Eclipse Modeling Framework (EMF) (siehe [EMF12]) eingesetzt werden. Aufgrund der derzeit besseren Toolunterstützung stellt EMF die häufiger verwendete Variante dar. In diesem Fall wird für die Spezifikation der abstrakten Syntax ein Ecore-Modell benötigt. Dabei handelt es sich um ein strukturelles Klassenmodell. Das Modell kann mit einem Ecore-Modellierungstool erstellt werden, wie sie bspw. für Eclipse [Ecl12] verfügbar sind. Aus dem Ecore-Modell müssen im Anschluss durch einen sogenannten *EMF-Compiler* Java-Klassen erzeugt werden. Dies sind die Klassen für die Java-Objekte (auch EMF-Objekte), die in der Analysephase erzeugt werden. Abb. 3.20 zeigt bspw. das Metamodell für Alligator Eggs. Das in Abb. 3.19, S. 92, abgebildete Modell (ganz rechts) ist eine Instanz dieses Metamodells.

Es sei an dieser Stelle angemerkt, dass DiaMeta die Weiterentwicklung einer Toolsammlung namens DiaGen [Min01, Min02] ist. Im Gegensatz zu DiaMeta

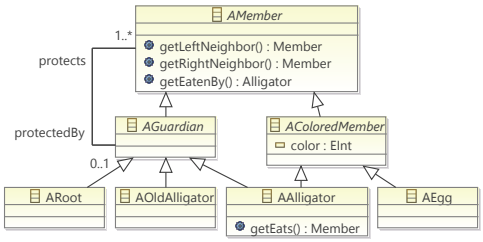


Abbildung 3.20: Alligator Eggs – Metamodell

kommt in DiaGen eine Hypergraphgrammatik für die Spezifikation der abstrakten Syntax zum Einsatz. Analog zur typischen Vorgehensweise für textuelle Sprachen gehört ein ASG zur spezifizierten Sprache, falls dieser einer Ableitung der Hypergraphgrammatik entspricht.

Spezifikation der konkreten Sprachsyntax und des Editors

Für die Spezifikation von konkreter Syntax und weiteren Details zur Editorgenerierung, stellt DiaMeta eine Anwendung namens DiaMeta-*Designer* zur Verfügung. Ein Screenshot ist in Abb. 2.23, S. 47, zu sehen. Basierend auf der daraus hervorgehenden Spezifikation kann der DiaMeta-*Generator* – ebenfalls ein Teil von DiaMeta – große Teile des Editor-Codes generieren.

Die Sprach- und Editorspezifikation, die durch den DiaMeta-Designer erstellt wird, enthält dabei

- die Spezifikation der Diagrammkomponenten mit
  - den Komponentenhyperkanten (entsprechen $\mathcal{CMP}$),
  - den Attributen (entsprechen $\mathcal{V}$),
  - der statischen Visualisierung (Rechtecke, Linien, Text etc.),
  - den Bereichen der Konnektoren und
  - weiteren Editor-spezifischen Merkmalen (z. B. die sogenannten Handles, um Komponenten verschieben und skalieren zu können),
- die Spezifikation der Relations- und Mehrzweck-Hyperkanten (entsprechen $\mathcal{REL}$ und $\mathcal{GEN}$),
- die Reduktionsregeln (spezielle GTRs zur Erzeugung des ASGs) und
- die Operationen (GTRs zur Manipulation des KSGs, die ähnliche Möglichkeiten bieten, wie die vorgestellte Klasse $\mathcal{PG}_{\mathcal{LV}}$).

Je nach Komplexität und individueller Funktionalität eines Editors ist es danach notwendig, dass der Editor-Entwickler zusätzliche Module (z. B. den Layouter) für den Editor programmiert. Zusammen mit dem *Editor-Framework* von DiaMeta und den generierten Teilen, entsteht ein Editor für die spezifizierte Sprache. Eine Übersicht über die beschriebenen Schritte und Tools wird in Abb. 3.21 dargestellt.

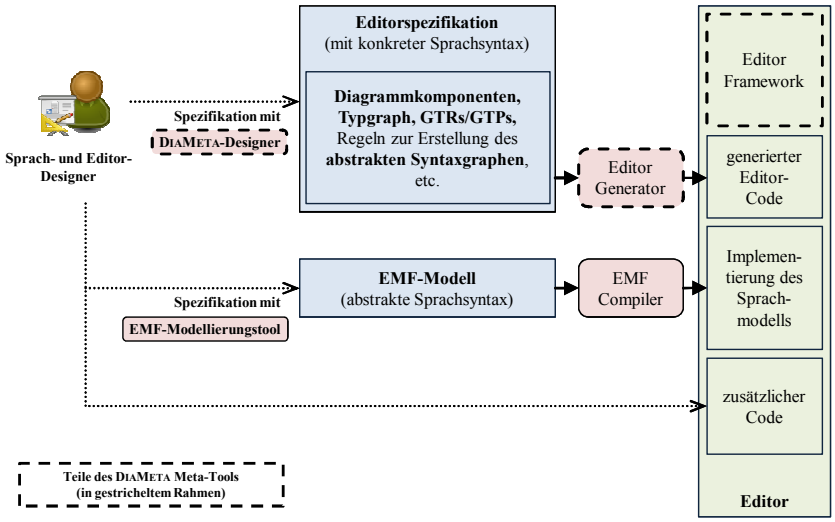


Abbildung 3.21: Spezifikation und Generierung eines DiaMeta-Editors

3.4 Andere Ansätze und Meta-Tools

Neben DiaMeta existieren zahlreiche andere Editor-Frameworks und -Generatoren. Von diesen sollen einige kurz vorgestellt werden. Aufgrund der Vielzahl kann jedoch keine vollständige Auflistung oder detaillierte Beschreibung präsentiert werden. Ähnliche Übersichten und Vergleiche, z. T. auch für ältere Systeme, werden in [Jun00, Min01, LMB+01, LVA04, RDE+08, Cra10] präsentiert. Eine Übersicht über mögliche Formalismen (neben Graphgrammatiken) zur Spezifikation visueller Sprachen bietet [MM96].

Graphical Modeling Project (GMP)

Das Graphical Modeling Project (GMP) ist eine Sammlung von Eclipse-Projekten [Ecl12], die zur Implementierung von grafischen Editor-Plugins für Eclipse genutzt werden können. Es umfasst Projekte wie das Eclipse Modeling Framework (EMF), das Graphical Modeling Framework (GMF) und das Graphical Editing Framework (GEF). EMF wird zur Spezifikation der abstrakten Sprachsyntax (inkl. Operationen) in Form eines Ecore-Metamodells und zur Generierung von Java-Code verwendet (vgl. Abschnitt 3.3.2, S. 93). Inzwischen existieren auch viele Möglichkeiten, Einschränkungen für das Metamodell zu erstellen (z. B. mittels Object Constraint Language (OCL) [CODA+11]), um die abstrakte Sprachsyntax noch genauer festzulegen. Das EMF Validation Framework kann dann zur Validierung eines Modells verwendet werden. Mittels GMF wird zum einen die konkrete Syntax für die einzelnen durch das Metamodell spezifizierten Elemente erstellt. Zum anderen können damit weitere Aspekte des Editors wie Werkzeugpalette, Menüeinträge etc. festgelegt werden. Aus den hervorgehenden

Spezifikationen kann ebenfalls Code erzeugt werden. Zusammen mit dem GEF entsteht so ein visueller Editor für das verwendete Ecore-Metamodell.

Insgesamt werden keine graphbasierten Mechanismen wie GTs genutzt. In [BDUW08] wird dieser Ansatz mit Möglichkeiten verglichen, wie sie durch Fujaba und PROGRES geboten werden. Spezielle Diagrammoperationen oder erweiterte Funktionen des Editors müssen in der Regel händisch programmiert werden, wobei das GEF hilfreiche Grundfunktionalität enthält.

GenGED und TIGER

Ein weiteres Meta-Tool ist GenGED [Bar98, Erm06]. Die abstrakte Sprachsyntax wird darin auf Grundlage von Graphgrammatiken (inkl. Typgraph) spezifiziert. Diese entsprechen auch den GTRs, die zur Modifikation von Diagrammen verwendet werden. Die GenGED-Editoren sind dabei rein syntaxgesteuert. Ein weiterer Unterschied zu DiaMeta (oder DiaGen) ist, dass gewöhnliche Graphen anstatt Hypergraphen zur Repräsentation von Diagrammen verwendet werden. Als GTS wird AGG eingesetzt, das bereits in Abschnitt 3.2.6, S. 87, vorgestellt wurde. Die konkrete Syntax (aufgrund der eingesetzten Graphgrammatik auch visuelles Alphabet genannt) oder die SPO-GTRs können größtenteils visuell mit entsprechenden Editoren spezifiziert werden. Darüber hinaus können Layout-Bedingungen festgelegt werden. Zur Darstellung des Diagramms wird dann eine Konfiguration gesucht, die diese Bedingungen erfüllt. Dies geschieht durch Lösen des CSPs mit Hilfe des Constraintlösers Parcon.

TIGER [EEHT05] ist der Nachfolger von GenGED. Durch TIGER generierte Editoren basieren ebenfalls auf GTs und AGG. Mittels TIGER können allerdings Editor-Plugins für Eclipse erzeugt werden. Hierfür wird außerdem das GEF eingesetzt.

AToM³

In AToM³ [LV02, LVA04] können visuelle Formalismen[12] (als Metamodelle) modelliert werden. Standardmäßig erfolgt dies durch Entity-Relationship-Modelle, die selbst einen mittels AToM³ erzeugbaren Formalismus darstellen. Zur genaueren Spezifikation können diese durch OCL [OCL12] ergänzt werden.

In AToM³ können GTRs (nach SPO- oder DPO-Ansatz) oder Python-Code [Het02] zur Manipulation von Modellen (eines bestimmten Formalismus bzw. Metamodells) eingesetzt werden. Nach Anwendung einer GTR wird geprüft, ob eine weitere GTR angewendet werden kann. Die Reihenfolge der Prüfung wird dabei durch Priorisierung der GTRs festgelegt. Wird eine anwendbare GTR gefunden, so wird diese auch angewendet, und dieser Prozess endet erst, wenn keine anwendbare GTR mehr gefunden wird. Unter Verwendung dieses Systems ist es bspw. möglich, Modelle in andere Formalismen zu transformieren.

[12] Prinzipiell handelt es sich dabei um visuelle Sprachen, allerdings wird der Aspekt betont, dass mit AToM³ Formalismen modelliert werden.

Aus den in der AToM³-Umgebung erstellten Spezifikationen kann Python-Code generiert werden. Im Gegensatz zu vielen anderen Meta-Tools kann dieser Code zur Laufzeit geladen werden, wodurch der generierte Editor (bzw. Formalismus) direkt in der AToM³-Umgebung ausgeführt werden kann.

DEViL

DEViL [KS02, CK09, Cra10] ist der Nachfolger eines Systems namens VLEli [Jun00] und ist somit ein Meta-Tool zur Generierung visueller Strukturreditoren. Die abstrakte Syntax (auch abstrakte Struktur) der Sprache wird mit einer textuellen Sprache namens DSSL spezifiziert. DSSL wird durch das DEViL-Generatorensystem zur Verfügung gestellt und unterstützt Konzepte wie Klassen, Attribute und Vererbung. Dies führt dazu, dass Diagramme intern durch (Struktur-)Bäume repräsentiert werden. Die Bäume bestehen aus Struktur- und Attributknoten, wobei es verschiedene Arten von Attributknoten für primitive Werte, für Unterkomponenten (wodurch die Baumstruktur entsteht) und für Referenzen auf andere Baumknoten gibt.

Um die Bäume grafisch abzubilden, werden Sichten spezifiziert. Hierfür werden sogenannte *visuelle Muster* verwendet. Durch Parameter können die visuellen Muster sprachspezifisch individualisiert werden.

MetaEdit+

MetaEdit+ [TK09] ist unter den präsentierten Meta-Tools eines der wenigen kommerziellen Systeme. Die abstrakte Sprachsyntax wird mittels Metamodell spezifiziert. Die konkrete Sprachsyntax wird in einem grafischen Editor festgelegt.

Für das Metamodell wird die Metamodellierungssprache GOPPRR (Graph, Object, Property, Port, Relationship and Role) verwendet. Ermöglicht wird damit die Spezifikation von graphorientierten Modellierungssprachen. Syntaxgesteuertes Editieren (z. B. mittels GTRs) wird allerdings nicht direkt unterstützt. Es wird aber eine API zur Verfügung gestellt, mit der Modellelemente extern abgefragt und manipuliert werden können.

Sprachdefinitionen werden von der MetaEdit+-Umgebung interpretiert, d. h., es müssen keine Editoren generiert werden. Ansonsten bietet das System zahlreiche weitere Features, wie die Möglichkeit im Team an Modellen zu arbeiten oder eine integrierte Lösung zur Codegenerierung.

Weitere Editor-Frameworks und Generatorensysteme

Weitere Editor-Frameworks und Generatorensysteme sind KOGGE [CE94], VLCC [CLOT97], VisPro [ZZ98], VPE [Gra98] DOME [EK00], GME [LMB+01], Moses [EJ01], Penguins [CM03] oder die DSL Tools [CJKW07].

Darunter fallen Systeme für freies Editieren (Penguins, VLCC, DOME, Moses, GME) sowie für strukturiertes Editieren (VisPro, KOGGE). Zur Sprachspezifikation werden sehr unterschiedliche Ansätze verwendet (vgl. auch [MM96]), z. B. Graphgrammatiken (VisPro), Prädikate (Moses), div. Metamodelle (KOGGE,

DOME, GME, DSL Tools), Constraint Multiset Grammars (Penguins) oder Positional Grammars (VLCC). Einige Systeme generieren keinen Editor-Code, sondern interpretieren die spezifizierte Sprachsyntax direkt (GME, DOME).

Mit den DSL Tools [CJKW07] bietet Microsoft einen Ansatz zur Erstellung visueller DSLs und entsprechender Editoren. Die DSL Tools sind in aktuellen Versionen von Microsofts Visual Studio integriert und werden von Microsoft als wichtiges Konzept zur Umsetzung sogenannter Software-Factories [GS03] aufgefasst. Insgesamt sind die Konzepte der DSL Tools mit denen von GMP vergleichbar.

Kapitel 4

Animation graphbasierter Diagramme

In Kapitel 3 wurde gezeigt, dass Graphen einen geeigneten Formalismus bieten, um Diagramme visueller Sprachen zu repräsentieren. In welcher Form Diagramme interaktiver animierter Sprachen (vgl. Kapitel 2) durch Graphen repräsentiert werden können, wurde jedoch nicht erläutert.

In diesem Kapitel soll daher ein graphbasierter Animationsansatz präsentiert werden, der auch die Grundlage für den später vorgestellten modellgetriebenen Entwicklungsansatz bildet. Um den Animationsansatz auch auf formaler Ebene darstellen zu können, werden zunächst abstrakte Animationssysteme (AASe) eingeführt. Diese können für die Umsetzung interaktiver animierter Systeme eingesetzt werden. Da die Beschreibung animierter Sprachen auf dieser allgemeinen Grundlage allerdings schwierig ist, wird im Anschluss ein zweiter Formalismus namens AAS/GT vorgestellt. Mit ihm kann jedes AAS umgesetzt werden, denn er basiert auf denselben Ideen. Im Gegensatz zu AAS eignet sich AAS/GT jedoch zur Spezifikation interaktiver animierter Sprachen, da im Rahmen von AAS/GT zusätzlich Graphen und GTs eingesetzt werden.

Die folgenden Abschnitte beschreiben AAS und AAS/GT (Abschnitt 4.1 und Abschnitt 4.2). Danach wird darauf eingegangen, wie AAS/GT in einer Umgebung wie DiaMeta eingebettet werden kann (Abschnitt 4.3). Abschließend wird der vorgestellte Animationsansatz mit anderen Ansätzen verglichen (Abschnitt 4.4). Dabei wird ebenfalls die Vorgehensweise anderer Meta-Tools erläutert.

4.1 Abstraktes Animationssystem (AAS)

Für interaktive animierte Sprachen spielen neben sprachspezifischen Konzepten v. a. Zustände, Ereignisse und grafische Animationen eine große Rolle. Es liegt daher nahe, die Diagramme solcher Sprachen als diskretes Ereignis- und

Transitionssystem mit (theoretisch) kontinuierlichem Zustands- und Zeitraum zu betrachten.

Als Ereignisse können dabei Benutzerinteraktionen betrachtet werden, z. B. wenn ein Editornutzer in einem B/E-Netz auf eine Transition klickt, damit diese schaltet. Aber auch durch Interaktion zwischen einzelnen Sprachkomponenten können Ereignisse auftreten, z. B. falls in Avalanche eine Murmel auf einen Hebel fällt, der die Murmel anschließend blockiert. Entsprechend kann man *externe Ereignisse* und *interne Ereignisse* unterscheiden. Während externe Ereignisse von einer externen Quelle (externes System, Editornutzer etc.) zu beliebigen Zeitpunkten ausgelöst werden können, werden interne Ereignisse berechnet. Dabei bestimmt der Systemzustand, welches interne Ereignis ausgelöst wird und zu welchem Zeitpunkt dies geschieht. Die Möglichkeit, dass kein internes Ereignis ausgelöst wird, besteht ebenfalls.

Zwischen den Ereignissen und den damit verbundenen Zustandsübergängen kann das Diagramm animiert dargestellt werden. In einem B/E-Netz können sich bspw. die Tokens auf den Pfeilen bewegen, in Avalanche fallen die Murmeln ihre Bahnen hinunter etc. Die Visualisierung basiert dabei auf dem aktuellen Systemzustand und der Zeit. Durch eine kontinuierlich fortschreitende Zeit kann eine flüssig animierte Visualisierung erfolgen.

Ein derartiges System soll im Folgenden abstraktes Animationssystem (AAS) genannt werden, wofür auch ein gleichnamiger Formalismus eingeführt wird. Dieser wird auch in [SM10] beschrieben. Insgesamt reiht sich der Formalismus in ähnliche Formalismen, wie bspw. DEVS und dessen Variationen (siehe Abschnitt 4.4, S. 113), ein.

Formale Definition

Im Folgenden werden abstrakte Animationssysteme zunächst formal definiert und anschließend erläutert.

Hilfsdefinitionen
Verwendet werden folgende Symbole:

- $\mathbb{T}$ bezeichne die überabzählbare Menge der *absoluten Zeitpunkte*.
- ω bezeichne den *unerreichbaren Zeitpunkt*, der auch Teil von $\mathbb{T}^\omega = \mathbb{T} \cup \{\omega\}$ ist, so dass für jeden Zeitpunkt $t \in \mathbb{T}$ gilt: $\omega > t$ und $\omega + t = \omega$. Für alle $x \in \mathbb{R}^+$ gilt außerdem: $x\omega = \omega$.
- 0 bezeichne den *ersten Zeitpunkt*, so dass für jeden absoluten Zeitpunkt $t \in \mathbb{T}$ gilt: $0 \leq t$. △

In Beispielen werden für $\mathbb{T}$ Zahlen (wenn nicht anders erwähnt zur Repräsentation von Millisekunden) verwendet.

Definition 4.1 (Abstraktes Animationssystem)
Ein *abstraktes Animationssystem (AAS)* ist ein Tupel $\mathcal{AS} = (Z, Q, \tilde{Q}, z_0, \delta, \tilde{\epsilon}, \tilde{\tau})$ mit folgenden Komponenten:

- Z ist die Menge der *Zustände*.
- Q ist die Menge der *Ereignisse*.
- $\tilde{Q} \subseteq Q$ ist die Menge der *internen Ereignisse*.
- $z_0 \in Z$ ist der *Startzustand*.
- $\delta : Z \times Q \times \mathbb{T} \to \mathbf{P}(Z)$ ist die *Zustandsübergangsfunktion*.
- $\tilde{\epsilon} : Z \to \mathbf{P}(\tilde{Q})$ und $\tilde{\tau} : Z \to \mathbb{T}^{\omega}$ beschreiben das Eintreten interner Ereignisse.

Außerdem sei $\hat{Q} = Q \backslash \tilde{Q}$ die Menge der *externen Ereignisse*. Jede der angegebenen Mengen darf überabzählbar sein. △

Die Funktion $\tilde{\tau}$ bestimmt, wann das nächste interne Ereignis in Abhängigkeit vom aktuellen Zustand z_i eintreten kann. Die Menge $\tilde{\epsilon}(z_i)$ gibt an, welche internen Ereignisse dies sein können. Enthält diese Menge mehrere Ereignisse, so kann das nächste interne Ereignis beliebig gewählt werden. Vor dem mittels $\tilde{\tau}(z_i)$ festgelegten Zeitpunkt kann jedoch auch ein externes Ereignis (z. B. durch Benutzerinteraktion) eintreten, das den Systemzustand dann vor dem Eintreten des eingeplanten internen Ereignisses ändert. Nach dem Zeitpunkt $\tilde{\tau}(z_i)$ kann kein externes Ereignis eintreten, da sich das System dann bereits in einem Folgezustand z_j mit $j > i$ befindet. Die Menge $\delta(z_i, q_{i+1}, t_{i+1})$ beschreibt unabhängig davon, ob ein internes oder ein externes Ereignis $q_{i+1} \in Q$ zum Zeitpunkt t_{i+1} eintritt, welche Nachfolgezustände möglich sind. Enthält diese Menge mehrere Zustände, so kann der Folgezustand beliebig gewählt werden.

Das AAS $\mathcal{AS}$ wird zum Zeitpunkt t_0 im Zustand z_0 gestartet. Der Ablauf von $\mathcal{AS}$ äußert sich in der zeitlichen Abfolge der eingenommenen Zustände, wobei Zustandsänderungen durch eintretende Ereignisse (interne sowie externe) zu bestimmten Zeitpunkten ausgelöst werden. Jeder Ablauf kann dargestellt werden durch die *Spur* $z_0 \xrightarrow[t_1]{q_1} z_1 \xrightarrow[t_2]{q_2} \cdots \xrightarrow[t_i]{q_i} z_i \xrightarrow[t_{i+1}]{q_{i+1}} \cdots$.

Dabei ist $z_0, z_1, \ldots \in Z$ die Folge der eingenommenen Zustände. Zum Zeitpunkt t_i, wobei $i > 0$, tritt jeweils das Ereignis q_i ein und löst den Zustandsübergang von z_{i-1} zu z_i aus. Für $i > 0$ gilt jeweils $z_i \in \delta(z_{i-1}, q_i, t_i)$ und

$$t_{i-1} \leq t_i \leq \tilde{\tau}(z_{i-1})$$
falls q_i ein *externes* Ereignis ist, d. h. $q_i \in \hat{Q}$, oder
$$t_i = \tilde{\tau}(z_{i-1}) \text{ und } q_i \in \tilde{\epsilon}(z_{i-1}),$$
falls q_i ein *internes* Ereignis ist, d. h. $q_i \in \tilde{Q}$.

Definition 4.2 (Visualisierung eines AAS)
Eine *Visualisierung* eines AAS $\mathcal{AS}$ wird definiert durch eine Berechnungsvorschrift $D : Z \times \mathbb{T} \to R$, wobei R alle möglichen grafischen Darstellungen repräsentiere. Eine Spur $z_0 \xrightarrow[t_1]{q_1} z_1 \xrightarrow[t_2]{q_2} \cdots \xrightarrow[t_i]{q_i} z_i \xrightarrow[t_{i+1}]{q_{i+1}} \cdots$ hat dann zu einem beliebigen Zeitpunkt $t \geq t_0$ die grafische Darstellung $D(z_i, t)$, wobei $i \geq 0$ derjenige Index sei, so dass gilt $t_i \leq t < t_{i+1}$. △

Die Absicht hinter der Definition ist, dass die grafische Darstellung eines Animationssystems das funktionale Bild von $\mathcal{AS}$ im jeweiligen Zustand ist. Im Zustand z_i befindet sich $\mathcal{AS}$ zwischen den Zeitpunkten t_i und t_{i+1}, falls $t_i \neq t_{i+1}$. Währenddessen hängt die grafische Darstellung vom Zustand z_i und der (kontinuierlich) ablaufenden Zeit $at \in \mathbb{T}$ (auch genannt *Animationszeit*) ab. Die Abhängigkeit von at kann genutzt werden, um Sachverhalte animiert darzustellen.

Falls gilt $t_i = t_{i+1}$, ist $D(z_i, t_i)$ keine zulässige grafische Darstellung. Hat eine Spur also mehr als einen Zustandsübergang zum selben Zeitpunkt t_i, so dass gilt $t_i = t_{i+1} = \ldots = t_{i+k}$ und $t_{i+k} < t_{i+k+1}$, so wird die grafische Darstellung zum Zeitpunkt t_i durch $D(z_{i+k}, t_i)$ abgebildet.

Beispiel 4.1 (Avalanche als AAS)
Dieses Beispiel zeigt die Spezifikation eines AAS. Es handelt sich dabei um die Umsetzung eines einzelnen Aspekts von Avalanche, der für das Beispiel zusätzlich vereinfacht und angepasst wurde. Darin wird lediglich eine Murmel, eine Bahn und ein Hebel in festgelegter Anordnung verwendet. Das AAS wird mit folgenden Komponenten spezifiziert:

$$
\begin{aligned}
Z &= \{(z_M, z_H, y_M, t_z) \mid z_M \in \{\circledcirc, \boxdot\} \wedge \\
&\qquad\qquad z_H \in \{\circledcirc, \circledcirc\} \wedge \\
&\qquad\qquad y_M \in \mathbb{R} \wedge t_z \in \mathbb{T}\} \\
Q &= \{key, hit\} \\
\tilde{Q} &= \{hit\} \\
z_0 &= (\circledcirc, \circledcirc, 0, 0)
\end{aligned}
$$

$$
\tilde{\tau}((z_M, z_H, y_M, t_z)) = \begin{cases} t_z + 30 - y_M & \text{, falls } z_M = \circledcirc \wedge z_H = \circledcirc \wedge y_M < 30 \\ \omega & \text{, sonst} \end{cases}
$$

$$
\tilde{\epsilon}((z_M, z_H, y_M, t_z)) = \begin{cases} \{hit\} & \text{, falls } z_M = \circledcirc \wedge z_H = \circledcirc \wedge y_M < 30 \\ \emptyset & \text{, sonst} \end{cases}
$$

$$
\delta((z_M, z_H, y_M, t_z), q, at) = \begin{cases} \{(\circledcirc, \circledcirc, y_M + at - t_z, at)\} & \text{, falls } q = key \wedge z_M = \circledcirc \wedge z_H = \circledcirc \\ \{(\circledcirc, \circledcirc, y_M + at - t_z, at)\} & \text{, falls } q = key \wedge z_M = \circledcirc \wedge z_H = \circledcirc \\ \{(\circledcirc, \circledcirc, y_M, at)\} & \text{, falls } q = key \wedge z_M = \boxdot \wedge z_H = \circledcirc \\ \{(\boxdot, z_H, 30, at)\} & \text{, falls } q = hit \\ \{(z_M, z_H, y_M, t_z)\} & \text{, sonst} \end{cases}
$$

Abb. 4.1 zeigt, wie die Visualisierung D des AAS den Startzustand z_0 in Abhängigkeit von der Animationszeit at darstellt. Das AAS und seine Visualisierung legen fest, dass die Murmel mit konstanter Geschwindigkeit nach unten fällt, falls der Zustand der Murmel z_M auf $\circledcirc$ („fallen“) gesetzt ist. Der Zustand des Hebels z_H kann jederzeit von einem Systemnutzer per Tastendruck (das externe

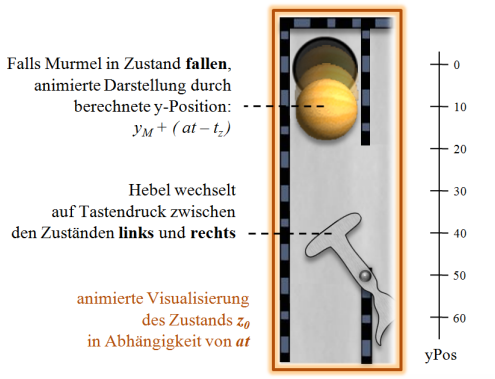


Abbildung 4.1: Visualisierung des AAS

Ereignis *key*) von ⦵ („links“) auf ⦶ („rechts“) und umgekehrt umgeschaltet werden, wobei in diesem Fall keine Animation genutzt wird, d. h., der Positionswechsel des Hebels erfolgt unmittelbar. Falls der Hebel sich in Zustand ⦵ befindet, kann die fallende Murmel auf den Hebel treffen (das interne Ereignis *hit*), woraufhin sie geblockt wird, d. h., der Zustand der Murmel wechselt zu ⊡ („geblockt“). Eine geblockte Murmel setzt ihren Fall allerdings fort, falls der Hebel umgeschaltet wird. △

4.2 Animation mittels Graphen und Graphtransformationen (AAS/GT)

Mittels AAS lassen sich interaktive animierte Sprachen, wie bspw. B/E-Netze, Alligator Eggs oder Avalanche beschreiben. Die angepasste Teilspezifikation von Avalanche aus Beispiel 4.1 legt allerdings die Vermutung nahe, dass es schwierig ist, solche Sprachen auf derartige Weise zu spezifizieren, v. a. wenn darin komplexe Zustandsräume abgebildet werden müssen. Erschwerend kommt hinzu, dass der Formalismus nicht dieselben Möglichkeiten bietet, wie Hypergraphen und zugehörige GTSe (vgl. Kapitel 3). AAS eignet sich daher nicht, um interaktive animierte Sprachen zu spezifizieren.

Aus diesem Grund wurde der Versuch unternommen, den Formalismus für AAS in geeigneter Weise auf Graphtransformationssysteme zu übertragen. Mit anderen Worten sollen Graphspezifikationen und GTRs verwendet werden, um ein AAS zu implementieren. Entstanden ist dabei ein Formalismus, der im weiteren Verlauf als AAS mittels Graphen und GTs (AAS/GT) bezeichnet wird. Der Formalismus ist dabei (nahezu) unabhängig von der genauen Art der verwendeten Graphen oder GTRs. In den Beispielen und Definitionen der folgenden Unterabschnitte werden allerdings typisierte, attributierte Hypergraphen $\mathcal{H}_{\mathcal{LV}}$ und GTP-GTRs $\mathcal{PG}_{\mathcal{LV}}$ verwendet (vgl. Abschnitt 3.1, S. 49ff, und Abschnitt 3.2, S. 59ff).

Nach der Beschreibung einiger Grundlagen von AAS/GT wird die Art und Weise beschrieben, wie externe und interne Ereignisse realisiert werden können. Abschließend werden die beschriebenen Konzepte noch einmal formal definiert, und es wird begründet, warum jedes AAS durch ein AAS/GT (mit attributierten Graphen) umgesetzt werden kann.

Grundlagen

Die Zustandsmenge Z eines AAS entspricht in AAS/GT der Menge der zulässigen Graphen, die sich im Rahmen einer Sprachspezifikation festlegen lässt, und einem zusätzlichen Attribut, das im Folgenden auch als *Zustandszeitpunkt* $zt \in \mathbb{T}$ vorgestellt wird und mit t_z aus Beispiel 4.1, S. 102, verglichen werden kann.

Vernachlässigt man den Zustandszeitpunkt, so entsprechen die Graphen den Zuständen, und Zustandsübergänge können mittels GTs realisiert werden. Die Zustandsübergangsfunktion δ kann daher mit einer Menge von GTRs umgesetzt werden, wobei die Wahl des GTRs durch das Ereignis bestimmt wird (siehe unten). Da δ den Zeitpunkt des Zustandsübergangs übergeben bekommt (der aktuelle Wert der Animationszeit at), muss diese Information auch innerhalb der GTRs abgefragt werden können, z. B. im Rahmen der Attributänderungen. In den Beispielen wird dieser Zeitpunkt kurz mittels at referenziert. Zusätzlich muss die GTR (oder ein damit verbundenes System) den Zustandszeitpunkt zt auf diesen Zeitpunkt at aktualisieren.[1]

Gemäß AAS werden Zustandsübergänge durch Ereignisse ausgelöst. Übertragen auf AAS/GT müssen die GTs bzw. die Anwendungen der GTRs durch Ereignisse ausgelöst werden. Die Ereignismenge Q soll dabei aus Ereignissen gebildet werden, die durch einen *Ereignistyp* und zusätzliche *Ereignisparameter* beschrieben werden. Beispielsweise wäre das Platzieren einer Murmel auf dem Avalanche-Spielbrett ein Ereignistyp, während die genaue Position bzw. das *Start*-Bauteil den Parameter darstellt.

Nach Spezifikation der Ereignismenge wird jedem Ereignistyp genau eine GTR zugeordnet. In diesem Zusammenhang werden die GTRs daher auch *Ereignis-GTRs* genannt. Sobald ein Ereignis des entsprechenden Ereignistyps eintritt, wird versucht, die zugehörige Ereignis-GTR anzuwenden. Als Ereignisparameter wird ein zur GTR passender Teilmatch verwendet, der bei der Anwendung berücksichtigt werden muss.

Externe Ereignisse

Externe Ereignisse $\hat{Q} \subseteq Q$ werden von einem externen System (z. B. dem Editornutzer) verursacht und können zu beliebigen Zeitpunkten eintreten. Wie zuvor beschrieben, ist jeder externe Ereignistyp mit einer GTR verknüpft. Ein externes Ereignis kann somit mit dem gezielten Aufruf einer entsprechenden GTR durch das externe System gleichgesetzt werden. Je nach Ereignis kann das externe System sogar einen Teilmatch als Ereignisparameter angeben. Danach wird ein

[1] Eine entsprechende Zuweisung „$zt := at$“ wird in den Beispielen nicht dargestellt.

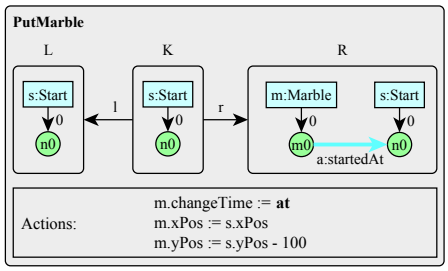


Abbildung 4.2: Externes Ereignis *PutMarble* als GTR

passender Match gesucht. Wird ein Match gefunden, für den die GTR anwendbar ist, führt dies zu einer sofortigen GT und damit zu einer Zustandsänderung im System. Ansonsten führt das externe Ereignis zu keiner GT, wird aber trotzdem „konsumiert“, d. h., es wird auch später nicht mehr verwendet.

Beispiel 4.2 (Externes Ereignis *PutMarble* als GTR)
In Avalanche können Murmeln jederzeit vom Editornutzer auf einem *Start*-Bauteil platziert werden. Hierfür wählt er zunächst ein *Start*-Bauteil aus und sendet danach das externe Ereignis *PutMarble* an das Avalanche-System. Im Editor entspricht dies der Selektion des *Start*-Bauteils durch Mausklick und einem Druck auf einen Knopf *PutMarble*.

Dadurch wird direkt die GTR *PutMarble* aus Abb. 4.2 ausgeführt. Für die Suche des Matches der GT wird das gewählte *Start*-Bauteil bzw. dessen interne Komponentenhyperkante als Teilmatch verwendet. In diesem Fall entspricht dies sogar dem kompletten Match (abgesehen vom zugehörigen Knoten), da lediglich die *Start*-Komponentenhyperkante inkl. Knoten Teil der LHS sind.

Ziel der GTR ist es, dem KSG eine *Marble*-Komponentenhyperkante hinzuzufügen, die per *startedAt*-Verbindungshyperkante mit der *Start*-Komponentenhyperkante verbunden ist. Außerdem werden durch die GTR einige Attribute der Murmel gesetzt. Das für die Berechnung der animierten Darstellung benötigte Attribut *changeTime* wird auf den aktuellen Wert der Animationszeit *at* gesetzt, d. h., auf den Zeitpunkt an dem die GTR angewendet wird. Zusätzlich werden die Attribute *xPos* und *yPos* in Relation zur Position des *Start*-Bauteils gesetzt. Auf dem Avalanche-Spielbrett erscheint dadurch eine Murmel, die sich auf dem oberen Teil des *Start*-Bauteils befindet und zusätzlich die Information trägt, wann ihr Fall von dort beginnt. △

Interne Ereignisse und Zeitberechnungsregel

Im Gegensatz zu externen Ereignissen, treten interne Ereignisse $\tilde{Q} \subseteq Q$ an genau festgelegten Zeitpunkten in Abhängigkeit vom aktuellen Zustand auf. Die Berechnung des nächstgelegenen Zeitpunkts, an dem ein bestimmtes internes Ereignis eintritt, wird in einem AAS durch $\tilde{\epsilon}$ und $\tilde{\tau}$ spezifiziert. Anstatt $\tilde{\epsilon}$ und $\tilde{\tau}$ direkt anzugeben, kann auch folgende Vorgehensweise verwendet werden, um $\tilde{\epsilon}$ und $\tilde{\tau}$ indirekt zu spezifizieren: nachdem alle im System möglichen internen

Ereignistypen festgelegt wurden, muss für jeden internen Ereignistyp eine Berechnungsvorschrift angegeben werden. Diese berechnet jeweils den (nächsten) Zeitpunkt eines internen Ereignisses in Bezug auf seinen Typ und einen bestimmten Systemteil (bzw. einen Diagrammabschnitt). Mit einer solchen Vorschrift können die Zeitpunkte interner Ereignisse für alle relevanten Systemteile berechnet werden. Beispielsweise gibt es in Avalanche einen internen Ereignistyp für den Fall, dass eine Murmel auf einen blockierenden Hebel trifft. Für jede Murmel in Bezug zu jedem Hebel können dabei der nächstgelegene Zeitpunkt berechnet werden, wann dies geschieht oder auch nicht geschieht.

Mit einer Vielzahl solcher Vorschriften kann für jeden Zustand eine vollständige Menge von internen Ereignissen (im Folgenden auch *Ereignisablaufplan*) berechnet werden. Das für den Ablauf relevante interne Ereignis, auch *nächstes internes Ereignis*, ist dann das Ereignis mit dem kleinsten berechneten Zeitpunkt, der größer oder gleich dem Zustandszeitpunkt zt ist. Interne Ereignistypen können außerdem priorisiert werden für den Fall, dass mehrere interne Ereignisse zum selben Zeitpunkt eintreten. Kann dadurch immer noch kein nächstes internes Ereignis bestimmt werden, d. h., mehrere interne Ereignisse haben auch dieselbe Priorität, so kann ein beliebiges als nächstes internes Ereignis gewählt werden.

Sobald der Zeitpunkt des nächsten internen Ereignisses erreicht wird, tritt dieses ein. Deshalb wird unmittelbar danach der Folgezustand entsprechend der Zustandsübergangsfunktion berechnet. Falls der Zustand geändert wird, muss der Ereignisablaufplan auf der Grundlage des neuen Zustands neu berechnet werden. Gleiches gilt, wenn zwischenzeitlich, d. h., vor dem geplanten Ereignis, ein externes Ereignis eintritt und der Zustand geändert wird. Dadurch ist es auch möglich, dass ein zuvor geplantes internes Ereignis ggf. nicht mehr eintritt. Bei einer praktischen Umsetzung (siehe Abschnitt 4.3, S. 111) können allerdings spezielle Eigenschaften genutzt werden, so dass der Ereignisablaufplan nicht vollständig bei jeder Zustandsänderung neu berechnet werden muss, sondern zuvor berechnete Ereignisse wiederverwendet werden können.

Die geschilderte Vorgehensweise ist nicht nur für ein AAS verwendbar, sondern kann gut auf ein AAS/GT übertragen werden. Wie bereits geschildert, wird jedem (internen) Ereignistyp eine GTR zugeordnet. Das Graphmuster der LHS einer zugeordneten GTR beschreibt dabei die Situation, in der ein internes Ereignis eintritt. Für jede GTR, die einem inneren Ereignistyp zugeordnet ist, werden mögliche Matches im KSG gesucht, so dass die GTR anwendbar ist. Daher muss der KSG nach jeder Graph- bzw. Zustandsänderung erneut analysiert werden. Für jeden gefundenen Match wird ein entsprechendes internes Ereignis zusammen mit dem Match im Ereignisablaufplan eingetragen. Zusätzlich wird der Zeitpunkt eingetragen, wann das interne Ereignis eintritt. Dieser Zeitpunkt wird durch eine sogenannte *Zeitberechnungsregel (TCR)* berechnet, die neben der GTR ebenfalls dem internen Ereignistyp zugeordnet wird. Ausgehend vom aktuellen KSG und dem gefundenen Match bestimmt die TCR den Zeitpunkt des entsprechenden internen Ereignisses. Falls das interne Ereignis mehrfach eintritt, z. B. falls sich eine Situation periodisch wiederholt, muss die TCR den Zeitpunkt des ersten Ereignisses berechnen, der größer oder gleich dem

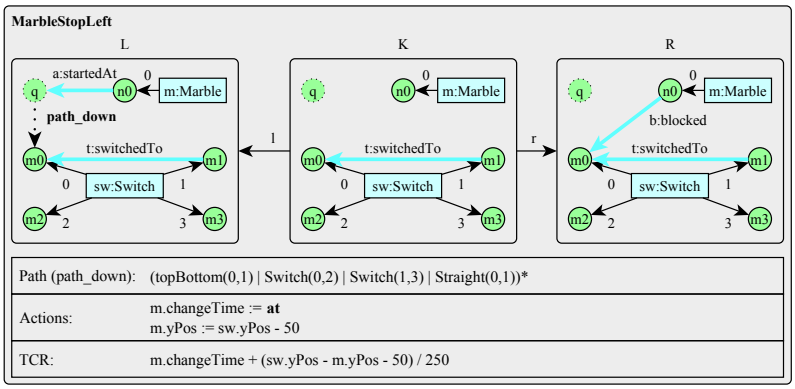


Abbildung 4.3: Internes Ereignis *MarbleStopLeft* als GTR mit TCR

Zustandszeitpunkt zt ist. Wird ein Wert berechnet, der vor zt liegt, wird kein internes Ereignis im Ereignisablaufplan eingetragen.

Falls der nächstgelegene Zeitpunkt eines geplanten internen Ereignisses im Ereignisablaufplan erreicht wird, tritt das interne Ereignis ein. Das bedeutet, dass die dem internen Ereignistyp zugeordnete GTR angewendet wird. Dabei wird der Match verwendet, der zuvor für das interne Ereignis bei der Berechnung des Ereignisablaufplans gefunden wurde. Die GTR ist dabei garantiert anwendbar, da der Ereignisablaufplan (und damit auch das nächste interne Ereignis inkl. Match) bei jeder Veränderung des KSGs neu berechnet wird.

Beispiel 4.3 (Internes Ereignis *MarbleStopLeft* als GTR mit TCR)
In Avalanche können fallende Murmeln auf einen Hebel treffen, welcher die Murmel anschließend blockiert (vgl. Beispiel 4.1, S. 102). Dies entspricht einem internen Ereignistyp. Abb. 4.3 zeigt die Ereignis-GTR *MarbleStopLeft*, die diesem internen Ereignistyp zugeordnet wird.

Die LHS der GTR zeigt eine *Marble*- und eine *Switch*-Komponentenhyperkante. Die *startedAt*-Verbindungshyperkante zeigt auf einen Knoten q, der die Startposition der Murmel im Hypergraphen repräsentiert. Der Pfadausdruck *path_down* prüft, ob ein Pfad von diesem Knoten q „senkrecht nach unten“ bis zum Knoten $m0$ der Weiche existiert. Durch die Mehrzweck-Hyperkante *switchedTo* wird die Lage des Hebels angezeigt. Im Beispiel führt *switchedTo* von Knoten $m1$ zu $m0$, d. h., der Hebel ist nach links gekippt. Anders formuliert prüft der Pfadausdruck, ob die Bahn, auf der die Murmel das Fallen begonnen hat, durch den Hebel der Weiche blockiert wird.

Für die Berechnung des Ereignisablaufplans wird *MarbleStopLeft* verwendet, um Matches der LHS im aktuellen KSG zu suchen. Das Diagramm wird also nach der oben beschriebenen Situation durchsucht, und jeder Fund (Match) führt zu einem entsprechenden Ereignis im Ereignisablaufplan. Darin wird jeweils auch der gefundene Match gespeichert.

Um die Zeitpunkte zu berechnen, wird für jeden Fund die TCR (auch in der Abbildung so genannt) verwendet. Der Ausdruck im Beispiel berechnet einen sol-

chen Zeitpunkt, indem die y-Position der gefundenen Murmel[2] mit der y-Position der gefundenen Weiche (und damit des Hebels) unter Berücksichtigung einer konstanten Geschwindigkeit in Relation gesetzt wird. Da ein solcher Zeitpunkt relativ zum Zeitpunkt ist, an dem die Murmel sich in seiner Position befindet, wird Attribut *changeTime* der Murmel einberechnet, das immer angibt wann der Zustand zuletzt geändert wurde und die gesetzten Attribute (wie das Attribut für die Position) zutreffend sind (vgl. Beispiel 4.2, S. 105).

Die GTR *MarbleStopLeft* wird angewendet, sobald die Animationszeit *at* den Zeitpunkt des nächsten internen Ereignisses erreicht, falls dieses vom entsprechenden Ereignistyp ist. Verwendet wird der vorher für das interne Ereignis gefundene Match. Die GT führt dazu, dass die *startedAt*-Verbindungshyperkante entfernt und eine *blocked*-Verbindungshyperkante in Bezug zur linken Seite der Weiche erzeugt wird. Dies sagt aus, dass die Murmel nun nicht mehr fällt, sondern auf der linken Seite der Weiche blockiert wird. △

Visualisierung

Die Idee der Visualisierung eines AAS/GT entspricht genau der eines AAS. Es soll also möglich sein, das System in Abhängigkeit von seinem Zustand und der Animationszeit zu visualisieren. Übertragen bedeutet dies, dass ein (statischer) Graph unter Verwendung der Animationszeit *at* nicht mehr in Form eines statischen Diagramms, sondern eines animierten Diagramms visualisiert werden kann.

An der grundlegenden Visualisierung eines KSGs, der dem aktuellen Zustand entspricht, soll wenig verändert werden. Für jeden Komponentenhyperkantentyp soll ein Aussehen spezifiziert werden können. Damit ist jede Komponentenhyperkante eines KSGs in der Lage, sich selbst zu zeichnen. Innerhalb des jeweiligen Zeichenalgorithmus muss allerdings die Animationszeit *at* berücksichtigt werden. Während in Avalanche bspw. eine Murmel fällt, soll sich die zugehörige Komponentenhyperkante nicht an ihrer y-Position zeichnen, die durch ihr Attribut *yPos* gegeben ist.[3] Stattdessen soll sie sich an einer berechneten y-Position zeichnen, die von *yPos* und der vergangenen Animationszeit abhängt. Die vergangene Animationszeit kann wiederum durch *at* und Attribut *changeTime* berechnet werden (vgl. Beispiel 4.3).

Formale Definitionen

Um die bisherigen Beschreibungen noch einmal zu präzisieren und um für spätere Kapitel eine formale Grundlage zu schaffen, werden nachfolgend AAS/GT und dessen Konzepte formal dargestellt. Dabei wird auf den (Hilfs-)Definitionen aus Abschnitt 4.1 und Kapitel 3 aufgebaut.

[2] Zur Erinnerung: die Position der Murmel ändert sich während des Falls nur visuell. Der Zustand, und damit auch das Positionsattribut, ist während des Falls bis zum nächsten Ereignis konstant.

[3] Der Zustand bleibt in einem AAS bzw. AAS/GT trotz grafischer Animation gleich. Das Attribut für die y-Position *yPos* verändert sich während des Falls der Murmel also nicht.

Definition 4.3 (Zeitberechnungsregel)
Gegeben sei eine GTP-GTR $pgtr \in \mathcal{PG}_{\mathcal{LV}}$, wobei $pgtr = (fgtr, gtp)$ und $fgtr = ((L, K, R), acr, CONS, NACS)$. Eine *Zeitberechnungsregel (TCR)* über $pgtr$ ist eine Funktion tcr, die jedem Paar (m, H_A) mit dem Match $L \xrightarrow{m} H$ und dem attributierten Hypergraphen $H_A \in \mathcal{H}_{\mathcal{LV}}$ den Zeitpunkt $t \in \mathbb{T}^\omega$ eines internen Ereignisses zuordnet.

$\mathcal{TC}_{\mathcal{G}}$ bezeichne die Menge aller TCRs über alle GTP-GTRs in $\mathcal{G} \subseteq \mathcal{PG}_{\mathcal{LV}}$.△

Eine TCR kann damit für einen aktuellen KSG (gegeben durch H_A) in Bezug auf einen bestimmten Teil des Systems (gegeben durch m) berechnen, wann ein internes Ereignis eintritt.

Definition 4.4 (AAS mittels Graphen und GTs)
Ein *AAS mittels Graphen und GTs (AAS/GT)* ist ein aus folgenden Komponenten bestehendes Tupel $\mathcal{AG} = (\mathcal{LV}, \mathcal{G}, \tilde{\mathcal{Q}}, \hat{\mathcal{Q}}, evtcr, evrule, prio)$:

- $\mathcal{LV}$ ist das Markierungsalphabet mit Attributen,
- $\mathcal{G} \subseteq \mathcal{PG}_{\mathcal{LV}}$ ist die endliche Menge der *Ereignis-(GTP-)GTRs*,
- $\mathcal{Q}$ ist die endliche Menge der *Ereignistypen*,
- $\tilde{\mathcal{Q}}$ ist die endliche Menge der *internen Ereignistypen*,
- $evrule : \mathcal{Q} \to \mathcal{G}$ heißt *Ereigniszuordnung* und ordnet jedem Ereignistyp $qt \in \mathcal{Q}$ eine GTP-GTR $pgtr \in \mathcal{G}$ zu,
- $evtcr : \tilde{\mathcal{Q}} \to \mathcal{TC}_{\mathcal{G}}$ heißt *Zeitberechnungszuordnung* und ordnet jedem internen Ereignistyp $qt \in \tilde{\mathcal{Q}}$ eine TCR $tcr \in \mathcal{TC}_{\{evrule(qt)\}}$ zu,
- $prio : \tilde{\mathcal{Q}} \to \mathbb{N}_0$ heißt *Prioritätszuordnung* und ordnet jedem internen Ereignistyp $qt \in \tilde{\mathcal{Q}}$ eine *Priorität* $n \in \mathbb{N}_0$ zu.

Außerdem sei Menge $\hat{\mathcal{Q}} = \mathcal{Q} \setminus \tilde{\mathcal{Q}}$ die Menge der *externen Ereignistypen.* △

Hilfsdefinitionen
Bei gegebenem AAS/GT $\mathcal{AG}$ werden folgende Konstrukte verwendet:

- $asys = (H_A, zt)$ bezeichne den *aktuellen Zustand des AAS/GTs.* Er enthält den *aktuellen KSG* $H_A \in \mathcal{H}_{\mathcal{LV}}$ und den *Zustandszeitpunkt* $zt \in \mathbb{T}$. $\mathcal{ASYS}$ bezeichne die Menge aller Zustände des AAS/GTs.

  Da kein Startzustand für ein AAS/GT angegeben wird, soll davon ausgegangen werden, dass dieser immer durch $asys_0 = (H^\emptyset, 0)$ gegeben ist.

- $\mathcal{A}_{qt}$ bezeichne die zu $qt \in \mathcal{Q}$ *kompatible Menge der Ereignisparameter*, d. h. die Ereignisparameter, die in Zusammenhang mit Ereignistyp qt verwendet werden können. Im Rahmen von AAS/GTs sind dies die möglichen Teilmatches in Bezug auf die LHS L der Ereignis-GTR $evrule(qt)$:

$$\mathcal{A}_{qt} = \{p \mid L \overset{p}{\rightarrowtail} H \wedge H \in \mathcal{H}_{\mathcal{L}}\} \quad .$$

- $Q = \{(qt, p) \mid qt \in \mathcal{Q} \wedge p \in \mathcal{A}_{qt}\}$ sei die Ereignismenge. Die Menge der internen Ereignisse $\tilde{Q}$ entspricht $\{(qt, p) \mid qt \in \tilde{\mathcal{Q}} \wedge p \in \mathcal{A}_{qt}\}$. Die Menge der externen Ereignisse kann analog gebildet werden bzw. durch $\hat{Q} = Q \setminus \tilde{Q}$.
- Ein Tripel (qt, m, t) mit dem internen Ereignis $(qt, m) \in \tilde{Q}$ und einem Zeitpunkt $t \in \mathbb{T}$ heiße *geplantes Ereignis*. △

Nun lässt sich die in den vorherigen Abschnitten beschriebene Vorgehensweise genauer definieren, bspw. wie Ereignisablaufplan und das nächste interne Ereignis berechnet werden:

Definition 4.5 (Ereignisablaufplan und nächstes internes Ereignis)
Gegeben sei ein AAS/GT $\mathcal{AG}$. Die Funktion PLN ordnet jedem aktuellen Zustand $asys = (H_A, zt) \in \mathcal{ASYS}$, die Menge der geplanten Ereignisse (*Ereignisablaufplan*) zu. Sie ist gegeben durch

$$\begin{aligned} PLN : asys \mapsto \{(qt, m, t) \mid\ & (qt, m) \in \tilde{Q} && \wedge \\ & (m, H_A) \models evrule(qt) && \wedge \\ & t = (evtcr(qt))(m, H_A) && \wedge \\ & t \geq zt && \wedge \\ & t \neq \omega && \} \quad . \end{aligned}$$

Basierend auf dem Ereignisablaufplan ordnet die Funktion NXT die Menge der *nächsten internen Ereignisse* zu. Sie ist gegeben durch

$$\begin{aligned} NXT : asys \mapsto \{&(qt,m,t) \in PLN(asys) \mid \\ & \nexists (qt', m', t') \in PLN(asys) : \\ & (t' < t \vee (t' = t \wedge prio(qt') > prio(qt)))\} \quad . \end{aligned}$$ △

Aus dieser Definition ist u. a. ersichtlich, dass ein internes Ereignis nicht eingeplant wird bzw. eintreten kann, falls die TCR einen Zeitpunkt berechnet, der dem Wert ω entspricht oder vor dem Zustandszeitpunkt zt liegt. Das *nächste interne Ereignis* für einen aktuellen Zustand $asys \in \mathcal{ASYS}$ ist das geplante Ereignis der Menge $NXT(asys)$, falls diese nur ein Element enthält. Enthält die Menge mehr als ein Element, d. h., es gibt mehrere geplante Ereignisse mit gleicher Priorität zum nächstgelegenen Zeitpunkt nach zt, so kann ein beliebiges Ereignis als nächstes internes Ereignis gewählt werden. Im Falle von $NXT(asys) = \emptyset$ gibt es kein nächstes internes Ereignis. Ansonsten tritt das interne Ereignis $(qt, m) \in \tilde{Q}$ zum Zeitpunkt t ein, sobald die Animationszeit at den Zeitpunkt t erreicht und das interne Ereignis als nächstes internes Ereignis eingeplant wurde, d. h. $(qt, m, t) \in NXT(asys)$.

Der Eintritt eines (internen oder externen) Ereignisses führt zu einer GT gemäß folgender Definition. Die darin verwendete Schreibweise $H_A \overset{pgtr,p}{\underset{at}{\rightsquigarrow}} H_{A^\dagger}^\dagger$ soll andeuten, dass die verwendete GTR bzw. dessen Attributberechnungsregel Bezug auf $at \in \mathbb{T}$ nehmen darf. Wie bereits erwähnt, wird dieser Zeitpunkt in den Beispielen durch das Schlüsselwort *at* referenziert (vgl. Beispiel 4.2, S. 105, und Beispiel 4.3, S. 107).

Definition 4.6 (Graphtransformation bei Ereigniseintritt)
Gegeben sei ein AAS/GT $\mathcal{AG}$ und der aktuelle Zustand $asys = (H_A, zt) \in \mathcal{ASYS}$. Der *Eintritt des Ereignisses* $(qt, p) \in Q$ zum Zeitpunkt $at \in \mathbb{T} \wedge at \geq zt$ führt zu einem Folgezustand $asys' \in \mathcal{ASYS}$. Falls $(p, H_A) \models\!\!\approx evrule(qt)$, ist dies ein neuer aktueller Zustand und es gilt $asys' = (H_{A\dagger}^{\dagger}, at)$, wobei $H_{A\dagger}^{\dagger}$ abgeleitet wird durch $H_A \overset{evrule(qt),p}{\underset{at}{\rightsquigarrow}} H_{A\dagger}^{\dagger}$. Ansonsten ändert sich der aktuelle Zustand nicht, d. h. $asys = asys'$. △

Umsetzbarkeit

Es ist einfach zu zeigen, dass jedes AAS durch ein AAS/GT beschrieben werden kann, zumindest falls attributierte Graphen und Ereignis-GTRs verwendet werden, die entsprechende Attribute modifizieren können. Ein beliebiges Knoten- bzw. Kantenattribut (mit überabzählbarem Wertebereich) allein kann genutzt werden, um alle Zustände eines AAS zu kodieren. Die Zustandsübergangsfunktion δ eines AAS kann dann im Rahmen der Attributänderung einer Ereignis-GTR umgesetzt werden. Der Nicht-Determinismus der Zustandsübergänge (sowohl im Rahmen von δ als auch $\tilde{\epsilon}$) kann durch mehrere Ereignis-GTRs mit identischer Prioritätszuordnung abgebildet werden. Der Startzustand des AAS kann durch eine initiale GTR – angewendet durch ein internes Ereignis, das sofort ausgeführt wird, falls $H^{\emptyset}$ dem aktuellen KSG entspricht – hergestellt werden.

4.3 Erweiterungen des DiaMeta-Systems

Das DiaMeta-System wurde so angepasst, dass damit Editoren für interaktive animierte Sprachen erstellt werden können. Hierfür wurden sowohl DiaMeta-Designer, DiaMeta-Generator als auch DiaMeta-Framework angepasst. Die Grundlage aller Änderungen bildet dabei der im vorherigen Abschnitt vorgestellte Formalismus AAS/GT.

Die angepasste Architektur der generierten Editoren wird in Abb. 4.4 dargestellt. Der gezielte Aufruf von Operationen (in typischen DiaMeta-Editoren durch Knöpfe in der Werkzeugleiste) kann weiterhin verwendet werden, um Diagramme strukturiert zu editieren. Im Falle von interaktiven Sprachen können diese Operationen aber auch zum Auslösen *externer Ereignisse* genutzt werden. Unterstützt wird außerdem, dass vorher ein oder mehrere Diagrammkomponenten selektiert werden, welche dann die Parameter des externen Ereignisses darstellen. Derartige Operationen können über eine Schnittstelle auch von externen Prozessen ausgelöst werden. Technisch betrachtet sind strukturiertes Editieren und externe Ereignisse allerdings identisch.

Die größte Änderung stellt das hinzugefügte *Ereignissystem* dar, das bei Eintritt eines *internen Ereignisses* den KSG durch Aufruf einer entsprechenden GTR ändert. Der verwendete *Ereignisablaufplan* ist eine technische Umsetzung des in Definition 4.5, S. 110, beschriebenen Ereignisablaufplans und wird durch Suche von Matches und Verwendung der TCRs nach jeder Änderung des KSGs berechnet. DiaMeta optimiert die Berechnung dabei, indem die Änderung des

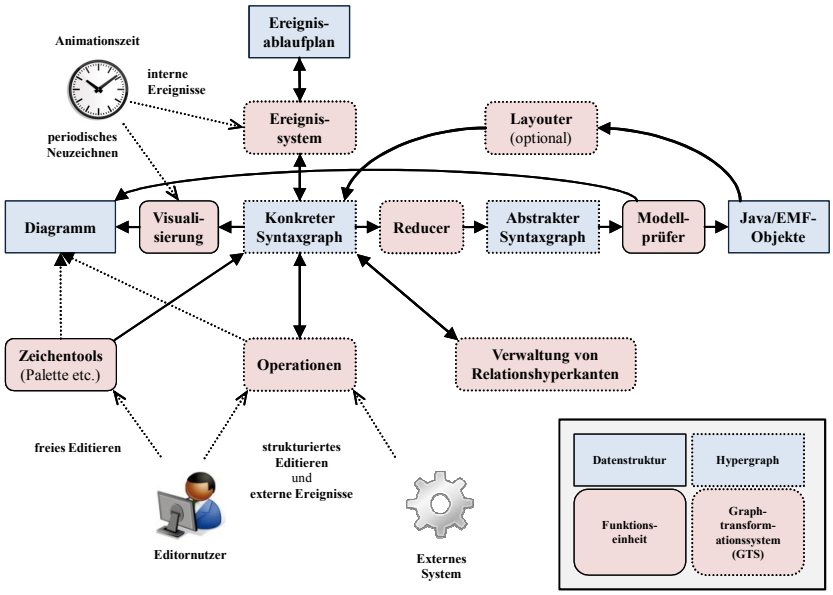


Abbildung 4.4: Architektur eines auf DiaMeta basierenden animierten Editors

KSGs analysiert wird und nur notwendige Teile des Ereignisablaufplans neu berechnet werden. Falls ein Attribut einer Komponentenhyperkante, die im gespeicherten Match eines internen Ereignisses vorkommt, geändert wird, muss das betreffende interne Ereignis gelöscht und anschließend ggf. neu berechnet werden. Dies geschieht ebenfalls, wenn eine entsprechende Komponenten- oder Verbindungshyperkante entfernt wird. Kommt eine Hyperkante hinzu, werden potentiell neue interne Ereignisse berechnet, wobei die hinzugekommene Hyperkante als Teilmatch vorgegeben wird.

Bei der Verwendung von negativen Anwendungsbedingungen gelten zusätzliche Regeln. Einerseits kann das Hinzufügen von Hyperkanten zur Folge haben, dass interne Ereignisse entfernt werden müssen. Andererseits kann das Entfernen von Hyperkanten dazu führen, dass neue interne Ereignisse eingeplant werden müssen. Außerdem kann die beschriebene „intelligente Berechnung“ von internen Ereignissen spezifisch für jede GTR auch deaktiviert werden, da in DiaMeta auch komplexe Bedingungen geprüft werden können (z. B. bei Pfad-Anwendungsbedingungen), die nicht direkt die Elemente im Graphmuster der LHS betreffen. In diesem Fall werden interne Ereignisse bzgl. dieser GTR bei jeder Änderung zunächst vollständig entfernt und danach neu berechnet.

Erstellte Diagramme verfügen in derartigen Editoren außerdem über eine *Animationszeit*. Diese kann an vielen Stellen abgefragt werden, z. B. für die animierte Visualisierung oder im Rahmen von GTRs. Ob und wie schnell diese Animationszeit abläuft, kann über eine zuschaltbare *Animationsleiste* (erkennbar im rechten Teil der Abb. 2.15, S. 36) gesteuert werden. Auch das Überspringen von Zeiträumen ist erlaubt.

Da ein statischer KSG nun auch animierte Szenen beschreiben kann, müssen Diagramme unabhängig von strukturellen Änderungen auch *periodisch neu gezeichnet* werden. Das Zeitintervall kann ebenfalls innerhalb der Animationsleiste eingestellt werden. Es werden allerdings nur die Diagrammabschnitte mit Diagrammkomponenten aktualisiert, die animiert werden. Entsprechende Kriterien, die bestimmen, ob eine Diagrammkomponente aktuell animiert wird bzw. periodisch neu gezeichnet werden muss oder nicht, kann für jede Diagrammkomponente angegeben werden.

Um Editoren für die oben beschriebene Architektur generieren zu können, wurde der DiaMeta-Designer so angepasst, dass Spezifikationen der konkreten Sprachsyntax gemäß AAS/GT möglich sind. Neben den bereits erwähnten Details ist es nun möglich, GTRs für interne Ereignisse zu spezifizieren. Dies geschieht genau wie bei GTRs für strukturiertes Editieren (bzw. externe Ereignisse), allerdings müssen zusätzlich eine TCR und ein Prioritätswert angegeben werden.

Zur Umsetzung der animierten Visualisierung bietet der DiaMeta-Designer keine direkte Unterstützung. Es können lediglich grafische Bestandteile der Diagrammkomponenten (bspw. Rechtecke und Linien) spezifiziert werden, die bestimmte Attribute (z. B. für die Position) zur Visualisierung verwenden. Für derartige Attribute unterstützt das DiaMeta-Framework aber spezielle Abfragen, die nicht den statischen Wert des Attributs zurückliefern, sondern (Java-)Methoden aufrufen, die beliebig ausprogrammiert werden können. Eine solche Methode kann die Animationszeit nutzen, um zu unterschiedlichen Zeitpunkten unterschiedliche Werte (z. B. für die Position einer Komponente) zurückzugeben. Bei entsprechender Verwendung innerhalb der Spezifikation für grafische Primitive führt dies zu Animationen. Trotz dieser Möglichkeit sollte dabei allerdings von händischer Programmierung der Animationen gesprochen werden.

4.4 Verwandte Konzepte und Arbeiten

Dieser Abschnitt behandelt zunächst einen Formalismus namens DEVS, der gut mit AAS verglichen werden kann. Anschließend werden Meta-Tools, Editor-Frameworks und verwandte Konzepte beschrieben, die Animationen unterstützen. Ansätze zur Modellierung von Animationen werden erst im folgenden Kapitel vorgestellt (siehe Abschnitt 5.6, S. 182). Darin werden u. a. auch Petri-Netze zur formalen Modellierung bzw. Beschreibung von Animationen erwähnt.

Discrete Event System Specification (DEVS)

Der beschriebene Formalismus AAS ist lediglich eine Variante zahlreicher diskreter, zeitbasierter Ereignissysteme, die v. a. im Bereich der Simulation eingesetzt werden. Ein bekannter Formalismus, der AAS sehr ähnlich ist, wurde von Zeigler bereits 1976 beschrieben [Zei76]. Die wichtigste Erweiterung von DEVS im Vergleich zu vorherigen Systemen war, dass Zustände mit einer Lebensdauer (aus einem kontinuierlichen Zeitraum) verknüpft werden können. Nach Ablauf

dieser Lebensdauer wird der Zustand gemäß einer Zuständsübergangsfunktion gewechselt.

Später wurde dieser DEVS-Formalismus noch einmal um hierarchische Konzepte erweitert [Zei84]. Neben den ursprünglichen *atomic DEVS-Modellen* können darin *coupled DEVS-Modelle* verwendet werden, um andere DEVS-Modelle (atomic oder coupled) miteinander zu verknüpfen. Als Schnittstelle dienen hierbei Eingabe- und Ausgabeereignisse, die jedes DEVS-Modell verarbeitet bzw. erzeugt. Insgesamt erhöht coupled DEVS die Ausdrucksstärke von atomic DEVS aber nicht: „DEVS is closed under coupling“ ([Zei84]). Dies bedeutet, dass für jedes coupled DEVS-Modell ein äquivalentes atomic DEVS-Modell konstruiert werden kann. Daher wird AAS nachfolgend hauptsächlich mit atomic DEVS verglichen. Auf die inzwischen zahlreichen Varianten von DEVS wird nicht näher eingegangen.

Neben unwesentlichen terminologischen Abweichungen können die Unterschiede zwischen AAS und DEVS wie folgt zusammengefasst werden:

- AAS definiert keine Ausgabeereignisse.
- AAS formuliert die Idee einer kontinuierlichen, animierten Visualisierung auf Grundlage des Systemzustands und der Systemzeit.
- AAS unterstützt keine hierarchischen Konzepte („coupling“).
- AAS verwendet bei Eintritt von internen Ereignissen oder für die Zustandsübergangsfunktion absolute Zeitpunkte, während DEVS eine Lebensdauer und die abgelaufene Zeit seit dem letzten Ereignis verwendet.
- AAS setzt die gleiche Zustandsübergangsfunktion sowohl für externe als auch interne Ereignisse ein, d. h., die Zustandsübergangsfunktion für interne Ereignisse ist ggü. DEVS anders definiert.
- AAS erlaubt unter Verwendung identischer Ausgangsparameter die Spezifikation mehrerer Zustandsübergänge oder interner Ereignisse, die einem Zustand folgen. Zur Systemlaufzeit kann von diesen Mengen jeweils ein beliebiger Zustandsübergang bzw. ein beliebiges internes Ereignis gewählt werden.

Vor allem der letzte Punkt stellt einen wichtigen Unterschied dar. Während in DEVS die internen Zustandsübergänge grundsätzlich deterministisch erfolgen, bietet AAS bspw. die Möglichkeit, dass zwischen unterschiedlichen internen Ereignissen, die zum selben Zeitpunkt stattfinden, beliebig gewählt werden kann. Entsprechend muss bei der Matchsuche im Rahmen von AAS/GT kein Determinismus vorausgesetzt werden. Treten zwei interne Ereignisse des gleichen Typs zur gleichen Zeit an unterschiedlichen Stellen ein, d. h. mit zwei unterschiedlichen Matches, ist das nächste interne Ereignis nicht deterministisch festgelegt.

Animationsunterstützung in Meta-Tools und Editor-Frameworks

Es folgt eine kurze Beschreibung der Möglichkeiten einiger Meta-Tools in Bezug zu Animation und Simulation, die damit meist gekoppelt sind. Die einzelnen Meta-Tools wurden bereits in Abschnitt 3.4, S. 95, vorgestellt. Reine Simulationsumgebungen wie z. B. MATLAB/Simulink [Bod06], die teilweise auch animierte Visualisierung unterstützen, werden nicht aufgeführt. Auch Ansätze und Frameworks zur Erstellung von Umgebungen für Algorithmen-, Programm- und Datenanimation (wie z. B. DANCE [Sta90]) sind meist sehr speziell und werden nicht berücksichtigt.

Zur Vertiefung seien auch Fallstudien erwähnt, die einige der beschriebenen Umgebungen miteinander vergleichen. In einer Fallstudie wird die Umsetzung des Spiels Ludo („Mensch Ärgere Dich Nicht“) untersucht [RDE+08], in einer anderen die sogenannte AntWorld-Simulation [RG08]. In beiden Studien werden Simulationsaspekte hervorgehoben, während Visualisierung oder gar flüssige grafische Animation eher zweitrangig ist. Trotzdem bieten sie eine interessante Übersicht über die Möglichkeiten einzelner Tools.

GenGED

Ermel beschreibt in [Erm06] ein formales Konzept zur Simulation und Animation visueller Sprachen auf der Grundlage von GTs. Es ähnelt AAS/GT dahingehend, dass mehrere GTRs spezifiziert und priorisiert werden können. Diese GTRs arbeiten allerdings auf der abstrakten Sprachsyntax. Für diese Syntax wird ein zusätzliches Simulationsalphabet spezifiziert, durch das simulationsspezifische Informationen im Graphen untergebracht werden können.

Die Visualisierung der Simulation kann in sogenannten *Animationssichten* erfolgen. Durch sogenannte *„Simulation-2-Animation“-Transformationen* entstehen aus Graphen der Simulation die Graphen einer Animationssicht. Letztere enthalten Elemente, die für eine Visualisierung genutzt werden können. Eine wichtige Idee ist dabei, dass je nach Anwendungsbereich verschiedene *Animationssichten* spezifiziert werden können. So wird die Simulation/Animation von speziellen Petri-Netzen bspw. auf eine grafische Visualisierung des Philosophenproblems abgebildet [EE05].

Umgesetzt wurde dieses Konzept für das Meta-Tool GenGED. Durch den „Animation Rule Editor“ können die Grammatiken der Simulationsspezifikation erweitert werden. Dabei werden den bestehenden Simulationsregeln mehrere Animationsoperationen hinzugefügt, welche die betroffenen Komponenten animiert darstellen können. Unterstützt werden eine einfache zeitliche Planung und Animationen wie lineare Bewegung, Änderung der Farbe etc. GenGED-Editoren (bzw. die sogenannten „Animationsumgebungen“) auf der Grundlage solcher Spezifikationen stellen Animationen aber nicht direkt dar, sondern exportieren selbstablaufende Animationssequenzen im *SVG*-Format [SVG11]. Eine Umsetzung von Editoren für interaktive animierte Sprachen wie Avalanche ist daher nicht möglich.

TIGER

In [BEHE08] wird die Umsetzung von Rubic's Clock in TIGER beschrieben. Die darin verwendeten GTRs können neben der Simulation (unter Verwendung der abstrakten Syntax) direkt die konkrete Syntax ändern. Zusätzlich werden Kontrollstrukturen zur Ausführung der GTRs verwendet, d. h., es können automatische Simulationsläufe erstellt werden. Obwohl die Arbeit nicht direkt auf kontinuierliche Animationen eingeht, ist davon auszugehen, dass die Uhren mit geringem Aufwand auch flüssig animiert dargestellt werden können.

DiaGen/DiaMeta (ohne AAS/GT)

Auch für DiaGen/DiaMeta-Editoren existiert ein sehr einfaches Konzept zur Animation von Diagrammänderungen. Durch GTs hervorgerufene Diagrammänderungen werden dabei nicht plötzlich, sondern über einen einstellbaren Zeitraum sanft animiert dargestellt (lineare Interpolation).[4] Für DiaGen existiert außerdem ein anderer (früher) Ansatz zur Beschreibung von komplexeren Animationen, die zur animierten Visualisierung einer GT eingesetzt werden können. Dieser wird in [MG97] beschrieben, wobei auch die Verwendung von Automaten zur Steuerung der Anwendung von GTRs erwähnt wird.

AToM^3

AToM^3 erlaubt die Spezifikation einer zeitlichen Verzögerung für GTRs, wenn die resultierenden Zustandsänderungen animiert visualisiert werden sollen [LVA04, SV07]. Komplexe Animationsabläufe können dabei mittels Python-Code implementiert werden. Außerdem ist die parallele Ausführung mehrerer GTs möglich [LETE04].

In [SV07] wird beschrieben, wie das Spiel PacMan mittels AToM^3 unter Verwendung von GTRs (inkl. deren Prioritäten, vgl. Abschnitt 3.4, S. 96) modelliert werden kann. Alternativ wird auch der DEVS-Formalismus (coupled und atomic) eingesetzt. Mit Hilfe der DEVS-Modelle[5] kann der Kontrollfluss für (Graph-)Transformationen modelliert werden. Diese Vorgehensweise ist dem in diesem Kapitel vorgestellten Ansatz somit in einigen Punkten ähnlich. Allerdings gehen entsprechende Arbeiten [SV07, SV08] nicht auf die Umsetzung grafischer Animationen oder die Spezifikation einer konkreten Sprachsyntax ein. An dieser Stelle sei angemerkt, dass ein ähnlicher Ansatz auch im Rahmen von VMTS existiert [MMC09].

DEViL

Die DEViL-Umgebung ermöglicht die Spezifikation von Simulationen [Cra10]. Zur Anwendung kommt dabei die eigene imperative, textuelle Simulationssprache DSIM, die Kontrollstrukturen und ereignisbasierte Konzepte (inkl. Ereignis-

[4] Dieser Modus kann als Alternative zum präsentierten Ansatz mittels AAS/GT betrachtet werden.

[5] Konkret wird die Modellierungssprache MoTiF eingesetzt [SV08].

warteschlange) unterstützt. Um die Zustandsänderung während der Simulation flüssig animiert zu visualisieren, können die einzelnen Simulationsregeln auch mit Animationen gekoppelt werden, wobei wie in GenGED die Spezifikation unterschiedlicher Animationssichten möglich ist. Ohne Spezifikation wird (lineare) grafische Interpolation zwischen zwei Zuständen verwendet, was dazu führt, dass Animationen automatisch auf der Grundlage der Simulationspezifikation dargestellt werden können [CK09]. Es können aber auch sogenannte *Animationsmuster* (z. B. „Bewegung entlang einer Linie“) eingesetzt werden.

Umsetzung von Petri-Netz-Editoren

Interessant ist aus Sicht dieser Arbeit auch ein Vergleich der Umsetzung von Editoren für B/E-Netze. Ein generelles Problem ist dabei, dass einfache GTRs (nach DPO-Ansatz, siehe Abschnitt 3.2.1, S. 60) zur Abbildung eines Schaltvorgangs nicht ausreichen, da in einem einzigen Simulationsschritt parallele Aktionen unbestimmter Anzahl notwendig sind (z. B. alle Tokens des Vorbereichs löschen). Ein Lösungsansatz ist daher, GTPs oder ähnliche Kontrollstrukturen zu nutzen.

Unter Einsatz eines geeigneten, ereignisgeteuerten Ansatzes (z. B. AAS/GT) genügen theoretisch jedoch einfache GTRs. Möglich ist die Umsetzung eines Schaltvorgangs bspw. durch das Setzen eines Flags für eine Transition, wodurch konsekutive interne Ereignisse für die Stellen des Vor- und Nachbereichs ausgelöst werden. Die GTRs, die diesen internen Ereignissen zugeordnet werden, müssen lediglich das entsprechende Flag überwachen und sollen sofort ausgeführt werden, falls ein solches Flag gesetzt wird. Obwohl dieser Ansatz umsetzbar ist, nutzt das später gezeigte Modell zur Implementierung des B/E-Netz-Editors allerdings Elemente, die zur Verwendung von GTPs führt: das Senden von Nachrichten (siehe Beispiel 6.6, S. 207, und die entsprechende Generierung in Abschnitt 7.3.3, S. 248ff).

Für GenGED (vgl. [Erm06]) müssen zur Umsetzung sogenannte amalgamierte GTRs verwendet werden. Durch diesen Ansatz, der u. a. untergeordnete Regeln nutzt, können Transformationen wiederholt angewendet werden. AToM^3 unterstützt generell die wiederholte Anwendung von GTRs, bis keine weitere Anwendung mehr möglich ist [LVA04]. In DEViL kann die imperative, textuelle Simulationssprache DSIM eingesetzt werden, die Verzweigungen und Schleifen erlaubt [CK09]. In Moses [EJ01] werden abstrakte Zustandsmaschinen (siehe [BCR00]) zur Spezifikation der Sprachsemantik von Petri-Netzen verwendet. Im Rahmen einiger anderer Frameworks werden Schaltvorgänge auch mittels universeller Programmiersprachen ausprogrammiert. Unter Einsatz von GMP wird bspw. das EMF-Modell direkt mittels Java manipuliert [Zim09].

Weitere Meta-Tools

Unter Verwendung der meisten übrigen, noch nicht genannten Meta-Tools und Editoren-Frameworks ist die Unterstützung von Animation noch unausgereift und grafische Animationen müssen teilweise oder vollständig ausprogrammiert werden: GMP [Zim09], DOME [EK00], GME (mittels „Dekorator“ [GME05]),

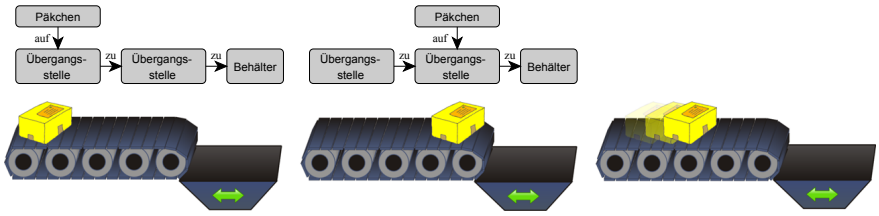


Abbildung 4.5: Verbreiteter Ansatz – Animation einer GT

MetaEdit+ (via API). Auch Moses [EJ01] unterstützt „Dekoratoren“ für Animationen. Inwieweit sich diese zur Umsetzung flüssiger Animationen einsetzen lassen, geht jedoch nicht eindeutig hervor. Häufig wird die schrittweise Visualisierung unterschiedlicher Zustände ohne die Visualisierung von Übergängen bereits als Animation bezeichnet.

Weitere Konzepte und Zusammenfassung

Meta-Tools und Editor-Frameworks unterstützen selten (aufwändige) Animationen. Über Umwege ist es aber oftmals möglich, Animationen durch eigenen Code oder andere Behelfs-Lösungen umzusetzen. Meist werden Animationsmöglichkeiten als sekundär betrachtet und sind kaum dokumentiert, was eine genaue Untersuchung schwierig gestaltet.

Viele der erwähnten Ansätze und Meta-Tools sehen Animation als grafische Erweiterung der Simulation eines Modells. Es wird daher großer Wert auf die Trennung zwischen Sprach- und Simulations-/Animations-Aspekten (zusätzliches Simulationsalphabet, Animationssichten etc.) gelegt. Die Simulation wird dabei oft auf der Grundlage der abstrakten Syntax spezifiziert und ausgeführt. Beim Ansatz in Verbindung mit AAS/GT ist Animation hingegen wichtiger Bestandteil der konkreten Sprachsyntax (und der Sprache selbst). Dies ermöglicht die Umsetzung von animierten Sprachen, die mit anderen Konzepten nicht oder nur schwierig umgesetzt werden können (z. B. Avalanche). Umgekehrt können aber animierte Sprachen realisiert werden, deren Modelle typischerweise auf Basis der abstrakten Syntax simuliert (z. B. B/E-Netze) und anschließend in konkreter Syntax präsentiert werden.

Bei den meisten der näher beschriebenen Meta-Tools steht außerdem die Visualisierung von Zustandsübergängen im Vordergrund. In Bezug zu GTs entspricht dies der Visualisierung der Änderungen durch eine GT. Es muss daher angenommen werden, dass die Visualisierung einer GT Zeit benötigt. Abb. 4.5 zeigt, wie eine Sprache zur Modellierung eines Förderbandsystems in solchen Meta-Tools typischerweise umgesetzt werden würde. Auf der linken Seite wird der Graph und seine Visualisierung vor der GT gezeigt. Die Situation nach der GT ist mittig dargestellt. Dieser Zustandsübergang bzw. die GT kann animiert dargestellt werden, wie rechts angedeutet.

Problematisch ist dabei die Umsetzung von interaktiven Sprachen (vgl. Abschnitt 2.1, S. 14). In Bezug zum Beispiel ist die Frage, ob es während der

Beförderung des Pakets zu Zwischenfällen kommen kann. Zwischenfälle können z. B. durch Interaktion zwischen mehreren animierten Diagrammkomponenten entstehen, oder wenn der Editornutzer Änderungen am Diagramm vornimmt. Dann werden Mechanismen benötigt, um Animationen und damit GTs unterbrechen zu können. In diesem Fall muss allerdings festgelegt werden, in welchen Zustand sich das Modell bei einer Unterbrechung befindet. Sollen simultane und voneinander unabhängige Animationen unterstützt werden, ist ebenfalls fraglich, welche GTs sich zeitlich überschneiden dürfen. Dieses Problem umgeht AAS/GT, indem Zustände animiert visualisiert werden. Daher können verzögerungsfreie GTs eingesetzt werden, die auch niemals parallel angewendet werden.

Andererseits sollte erwähnt werden, dass die Animation von Zuständen im Vergleich zur Animation von Zustandsübergängen weniger intuitiv ist und meist aufwändiger umgesetzt werden muss. In Systemen wie DEViL oder DiaGen können Animationen bspw. durch lineare/grafische Interpolation automatisch abgeleitet werden, wodurch der Spezifikationsaufwand nicht oder nur geringfügig steigt.

Die Kopplung von GTRs mit einer Dauer wird auch in [RDV09] behandelt. Darin wird eine Notation für Transformationsregeln[6] vorgestellt, die auch zeitliches Verhalten unterstützen. Neben der Spezifikation einer Dauer können auch Periodizität oder Ausnahmen/Unterbrechungen angegeben werden.

Formal wird der Zeitbegriff innerhalb von GTs in [GHV02] behandelt. Der darin präsentierte Ansatz verwendet logische Uhren, die durch spezielle Knotenattribute genannt *chronos* repräsentiert werden. Durch GTs (mit Zeit) werden die darin enthaltenen Zeitstempel dann aktualisiert, d. h., alle *chronos*-Werte der in der GT einbezogenen Knoten werden erhöht (oder zumindest nicht verringert) und alle erhalten denselben Wert. Dieses Konzept erinnert an das Attribut *changeTime*, das in Beispiel 4.2, S. 105, und Beispiel 4.3, S. 107, benutzt wurde, um den Zeitpunkt einer GT bzw. der Zustandsänderung zu speichern und Grundlage für Berechnungen (z. B. für eine animierte Visualisierung) bietet. Solche Attribute sind im Rahmen von AAS/GT allerdings weder fest integriert noch formalisiert.

[6] Diese sollen das Verhalten von DSLs spezifizieren und können mit GTRs in einer graphbasierten Umgebung verglichen werden.

Kapitel 5

Animation Modeling Language (AML)

Dieses Kapitel stellt die Animation Modeling Language (AML) vor. Mit dieser Sprache können Komponenten zusammen mit ihrer Struktur, ihrem Verhalten und ihrer grafischen Darstellung einschließlich ihren Animationen modelliert werden. AML bietet damit einige interessante Merkmale, die zur Spezifikation einer konkreten animierten Sprachsyntax eingesetzt werden können. Dabei ist AML zunächst unabhängig von den Ansätzen aus Kapitel 3 und Kapitel 4 zu betrachten. Die Sprache und ihre Modelle sind allerdings die Grundlage des modellgetriebenen Ansatzes, der in den späteren Kapiteln vorgestellt wird. Erstmalig erwähnt und teilweise beschrieben wird die Sprache in [SMPV10, SM12]. Da UML die Basis von AML bildet, sind für das Verständnis dieses Kapitels außerdem grundlegende Erfahrungen mit UML, UML-Werkzeugen und Modellierung im Allgemeinen hilfreich. Einen Einstieg in diese Thematik bieten Bücher wie [BRJ06, RQZ07], v. a. aber auch die Spezifikation von UML selbst [UML11].

Im ersten Abschnitt dieses Kapitels werden zunächst Entstehungsgeschichte und Ziele von AML beschrieben (Abschnitt 5.1). Anschließend gibt ein einführendes Beispiel einen Überblick über die Sprache und zeigt deren Möglichkeiten (Abschnitt 5.2). Erst danach werden Elemente und Notation ausführlicher erläutert, v. a. auch in Bezug auf UML (Abschnitt 5.3 bis Abschnitt 5.5). Abschließend wird ebenfalls auf verwandte Arbeiten, Sprachen und Ansätze eingegangen (Abschnitt 5.6).

5.1 Entstehung und Zielsetzung

Die Idee zur Entwicklung von AML entstand aus der Notwendigkeit einer Modellierungssprache, die auch zum Entwurf von animierten Sprachen effektiv genutzt werden kann. Als weitere Orientierung dienten Spiele oder typische grafische

Applikationen, wie sie v. a. mit Technologien wie Adobe Flash Professional [Fla12] erstellt werden können. Einige wichtige Konzepte, die hilfreich zur Modellierung von derartigen (animierten) Systemen und Anwendungen sind, waren bereits bekannt und sollten durch AML unterstützt werden. AML ist als Sammlung dieser Konzepte zu verstehen. Die Konzepte wurden dabei aus bestehenden Sprachen entnommen, erweitert und in angepasster Form für AML zusammengestellt.

Pleuß [Ple09] und Vitzthum [Vit05] beschreiben zusammen mit den beiden Sprachen Multimedia Modeling Language (MML) und Scene Structure and Integration Modelling Language (SSIML) – eigentlich eine ganze Sprachfamilie – wichtige Sprachelemente, die auch innerhalb von AML verwendet werden. Die Notation und verwendeten Begriffe stammen dabei hauptsächlich aus der MML. Die beiden Autoren waren auch maßgeblich an der Zusammenstellung der Ideen für AML beteiligt. Erst später wurde AML im Rahmen der vorliegenden Arbeit genauer spezifiziert und weiterentwickelt. Trotz gemeinsamer Wurzeln unterscheidet sich AML in einigen Punkten von den Sprachen MML und SSIML. Auch die Schwerpunkte und Ziele der Sprachen wurden bereits zu Beginn unterschiedlich gesetzt. Abschnitt 5.6, S. 182, beschreibt diesbezüglich einige Details.

Vor dem Entwurf von AML stand fest, dass folgende Merkmale modellierbar sein müssen:

- das Aussehen grafischer Komponenten,
- die Struktur bzw. der Aufbau komplexer Komponenten,
- das Verhalten der Komponenten einschließlich
- deren Interaktionsmöglichkeiten inkl. der Möglichkeit auch auf Nachrichten von externen Systemen zu reagieren und
- wie die Komponenten animiert dargestellt werden.

Als allgemeine Ziele für das Sprachdesign wurden ferner formuliert:

- die Sprachkonzepte und -elemente sollen in etablierte UML-Konzepte eingebettet werden können,
- die Sprachkonzepte und -elemente sollen mit UML-Vorkenntnissen intuitiv verstanden werden können,
- die Sprache und Toolunterstützung sollen mit aktuellen Metamodellierungstechnologien einfach realisiert werden können und
- die Sprache soll sowohl für die Planung/Analyse (Grobspezifikation) als auch zur Implementierung und Codegenerierung (Feinspezifikation) eingesetzt werden können.

Die genannten Ziele wurden augenscheinlich zwar erreicht, allerdings wurden keine Metriken zur Qualitätsmessung angewendet. Es wurde ebenfalls keine vollständige Evaluation durchgeführt, v. a. hinsichtlich der Verständlichkeit. Generell

gilt es als schwierig, die Qualität von Sprachen wie UML oder AML zuverlässig zu messen. Mögliche Validierungstechniken werden aber bspw. in [Sha01] präsentiert. Einige darin beschriebene Techniken (z. B. „persuasion“, „experience“ und „implementation“) werden im Rahmen dieser Arbeit eingesetzt. Das bedeutet zum einen, dass die Sprachelemente begründet werden. Zum anderen werden Spezifikationsmöglichkeiten der Sprache in Beispielen verwendet, die auch real umgesetzt worden sind und zur Verbesserung des ursprünglichen Animationsansatzes beigetragen haben. Es wurden jedoch keine empirischen Studien, detaillierten Analysen oder formale Beweise durchgeführt („evaluation“ und „analysis“).

Ähnlich wie UML und viele verwandte Sprachen (siehe Abschnitt 5.6, S. 182) kann AML sowohl zur groben Planung als auch zur verständlichen Dokumentation oder zur detaillierten Spezifikation von Systemen verwendet werden. Die in dieser Arbeit gezeigten Diagramme nutzen in den meisten Fällen einen hohen Detailgrad, wie z. B. die exakte Angabe von Dimensionen oder Dateinamen. Dadurch werden die meisten Aspekte der modellierten Sprachsyntax vollständig spezifiziert. Auf diese Weise können allerdings schnell „aufgeblähte“ und unübersichtliche Diagramme entstehen. In der Praxis kann dieses Problem bspw. gelöst werden, indem Spezifikationsdetails zwar im Modell eingebettet, im Diagramm allerdings versteckt werden. Es ist ebenfalls denkbar, Spezifikationsdetails in zusätzliche Textdateien auszulagern, wie dies manche Modellierungstools bspw. für OCL unterstützen (z. B. TOPCASED [PFG+06]).

5.2 Einführendes Beispiel

Bevor die einzelnen Elemente und Möglichkeiten von AML genauer beschrieben werden, soll dieser Abschnitt einige Sprachkonzepte informell einführen, um einen ersten Eindruck von der Sprache zu vermitteln. Dieser Abschnitt enthält daher die Beschreibung eines kleinen AML-Beispieldiagramms und eine grafische Aufbereitung der modellierten Elemente.

In dem Beispieldiagramm werden Aufbau, Aussehen, Verhalten und Animation einer Pendeluhr modelliert. Die Uhr besteht aus mehreren Teilen wie bspw. Gehäuse, Pendel und Ziffernblatt inkl. Minuten- und Stundenzeiger. Die Zeiger und das Pendel sind außerdem beweglich und können daher animiert werden. Es wird zunächst beschrieben, wie die Struktur und das Aussehen der Uhr in AML festgelegt werden können. Danach zeigt ein weiterer Unterabschnitt, wie das Verhalten und die Animationen der Uhr modelliert werden können.

Struktur und Aussehen

Das AML-spezifisch erweiterte UML-Klassendiagramm in Abb. 5.1 zeigt eine Klasse namens *PendulumClock*. Es handelt sich dabei nicht um eine gewöhnliche UML-Klasse, sondern um eine sogenannte *Medienkomponente*, was durch das Stereotyp-Icon an der rechten oberen Ecke erkennbar ist. Außerdem wird in einem weiteren Compartment (ganz unten) modelliert, aus welchen Unterkomponenten

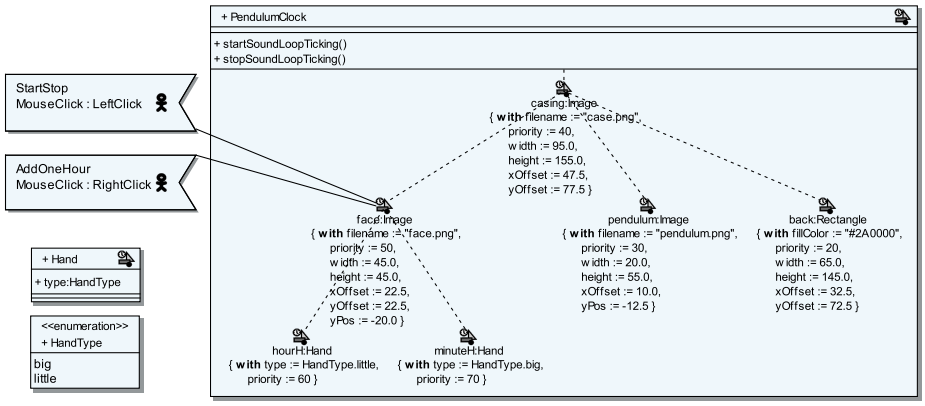


Abbildung 5.1: Pendeluhr – Medienkomponente und statische Struktur

die Medienkomponente zusammengesetzt ist. Im Falle der Uhr sind dies *casing*, *face*, *pendulum* etc. Diese Unterkomponenten stellen die sogenannten *inneren Merkmale* der Medienkomponente dar. Jedes innere Merkmal besitzt, genau wie ein normales Klassenattribut, einen zugewiesenen Typ. Die meisten gezeigten inneren Merkmale verwenden hierbei den Typ *Image*, eine standardmäßig verfügbare – und daher nicht gezeigte – Medienkomponente. Diese Standardkomponente ermöglicht das Laden und Anzeigen von Bilddateien und bietet u. a. das Attribut *filename*. Die Unterkomponente *casing* setzt dieses Attribut standardmäßig auf den Wert *„case.png“*. Dies ist der Name der Bilddatei, die das Gehäuse der Pendeluhr beinhaltet. Neben dem Dateinamen werden aber auch andere Attribute gesetzt, wie z. B. Höhe und Breite der Medienkomponente.

Neben *Image* können auch andere Standardkomponenten oder im Modell gezeigte Medienkomponenten verwendet werden. Grafische Details für die Zeiger werden bspw. in einer eigenen Medienkomponente *Hand* modelliert. Genaue Details werden in der Abbildung allerdings nicht abgebildet. Es wird lediglich angedeutet, dass ein Attribut namens *type* verwendet werden muss, um festzulegen, ob ein Minuten- oder Stundenzeiger angezeigt werden soll. Abb. 5.3 (links) zeigt, wie die Teile der Pendeluhr einzeln gezeichnet werden.

Die Baumstruktur der inneren Merkmale modelliert die Hierarchie der Unterkomponenten innerhalb ihrer übergeordneten Medienkomponente *PendulumClock*. Die Hierarchisierung wird in erster Linie verwendet, um räumliche Attribute von Medienkomponenten in Abhängigkeit zu setzen. Wird bspw. die *casing*-Medienkomponente verschoben, so werden auch Unterkomponenten wie *face* oder *pendulum* entsprechend verschoben. Gleiches gilt für Rotation, Skalierung o. ä.

Beim Zeichnen der einzelnen Teile muss eine bestimmte Reihenfolge eingehalten werden. So muss z. B. die Rückwand der Uhr (*back*) als erstes gezeichnet werden, da sie hinter allen anderen Teilen gezeichnet werden muss. Darüber folgen dann Pendel (*pendulum*), Gehäuse (*casing*), Ziffernblatt (*face*), etc. Diese Reihenfolge wird nicht von der Hierarchie bestimmt, sondern durch eine

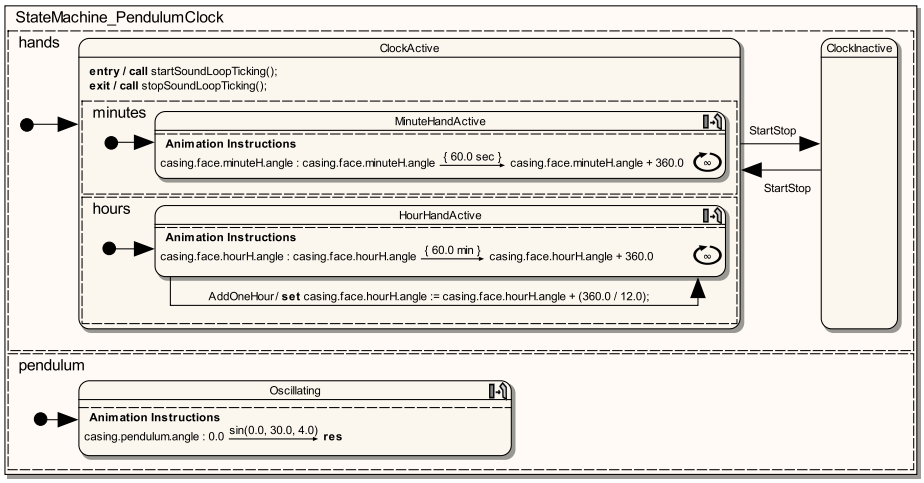


Abbildung 5.2: Pendeluhr – Verhalten und Animationen

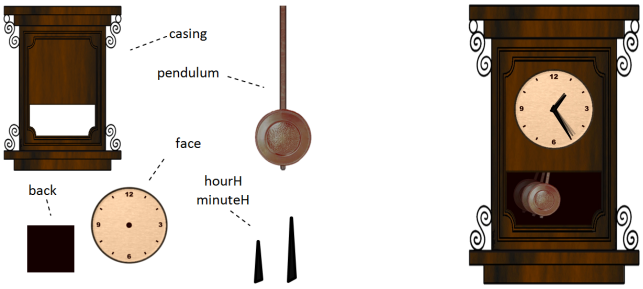


Abbildung 5.3: Grafische Unterkomponenten und animierte Darstellung

Zeichenpriorität festgelegt, die in Form des Attributs *priority* jeder grafischen Medienkomponente zur Verfügung steht.

Neben Medienkomponenten und grafischen Details ermöglicht AML auch die Modellierung von *Sensoren*. Sensoren sind nicht-visuelle Elemente, die jeweils einer Medienkomponente zugeordnet werden müssen, d. h., Medienkomponenten (auch in Form von Unterkomponenten) können Sensoren besitzen. Dabei gibt es unterschiedliche Arten von Sensoren. In dem Beispiel werden zwei sogenannte Benutzersensoren dargestellt (*StartStop* und *AddOneHour*). Beide gehören dem Ziffernblatt der Pendeluhr. Ihre Aufgabe ist es, Mausklicks auf dem Ziffernblatt zu melden. *StartStop* soll dabei Linksklicks überwachen, *AddOneHour* Rechtsklicks. Wird ein solcher Klick gemeldet, kann die Medienkomponente in modellierter Art und Weise reagieren.

Verhalten und Animation

Nachdem die statische Struktur und das äußere Erscheinungsbild festgelegt wurden, können auch Verhalten und Animationen der Pendeluhr modelliert werden. Dies ist mittels UML-Zustandsdiagramm möglich, das ebenfalls durch einige zusätzliche Konstrukte erweitert wird. Das Zustandsdiagramm für das betrachtete Beispiel wird in Abb. 5.2 gezeigt. Darin ist der Zustandsautomat für die Medienkomponente *PendulumClock* dargestellt.

Zum einen werden gewöhnliche Zustände verwendet, wie z. B. *ClockActive*. Es handelt sich dabei um einen zusammengesetzten Zustand mit zwei nebenläufigen Regionen. Für den Zustand wird mit einer AML-spezifischen Syntax festgelegt, dass Methoden der Medienkomponente (*startSoundLoopTicking* und *stopSoundLoopTicking*) aufgerufen werden, falls der Zustand betreten bzw. verlassen wird. In der Implementierung dieser Methoden, die nicht durch das AML-Modell vorgegeben wird, soll später das Geräusch einer tickenden Uhr an- bzw. abgeschaltet werden.

Neben gewöhnlichen Zuständen werden sogenannte *Animationszustände* verwendet. Befindet sich die Uhr in einem solchen Animationszustand, so soll sie „animiert“ dargestellt werden, d. h., ein oder mehrere Attribute, welche die Darstellung der Medienkomponente beeinflussen, werden in Abhängigkeit von der Zeit verändert. Hierfür werden *Animationsanweisungen* (engl. animation instructions) spezifiziert, die innerhalb des jeweiligen Animationszustands notiert werden. Die Animationsanweisung in Zustand *MinuteHandActive* legt bspw. fest, dass sich Attribut *casing.face.minuteH.angle* ändern soll, während der Zustand aktiv ist. Das Attribut bestimmt dabei den Darstellungswinkel des Minutenzeigers. Die Änderung soll über einen Zeitraum von 60 Minuten erfolgen, wobei der Winkel zum Zeitpunkt des Betretens des Zustands als Startwert angenommen wird und der Zielwert genau um 360° höher sein soll. Da keine weiteren Angaben gemacht werden, erfolgt die Änderung kontinuierlich, d. h. linear interpoliert. Das runde Symbol am rechten Rand der Anweisung mit dem Wert ∞ deutet außerdem an, dass dieselbe Wertänderung nach deren Ablauf unendlich oft wiederholt werden soll. Letztendlich sorgt die besprochene Anweisung dafür, dass sich der Minutenzeiger der Uhr nach jeweils 60 Minuten einmal um seine Achse dreht. Analog wird die Animation des Minutenzeigers modelliert.

Die Animation des Pendels (siehe Animationszustand *Oscillating*) nutzt keine lineare Interpolation. Stattdessen wird die Sinusfunktion zur Spezifikation der Änderung des Darstellungswinkel verwendet. Als Argumente werden konstante Werte für Phasenverschiebung, Amplitude und Periodendauer übergeben. Ein Zielwert und damit auch ein Ende der Animation wird nicht angegeben, was durch das Schlüsselwort *res* festgelegt wird.[1] Die angegebene Animationsanweisung führt somit zu einem permanent schwingenden Pendel.

Zusätzlich bietet die Uhr durch den Sensor *AddOneHour* die Möglichkeit, den Stundenzeiger eine Stunde nach vorne zu drehen. Hierfür ist an einer Transition, die vom Zustand *HourHandActive* wieder zurück zum diesem führt, der Name des

[1] Das Schlüsselwort ist eine Kurzform für „result“. Es kann verwendet werden, wenn sich der Wert durch vorherige Angaben automatisch ergibt oder schlicht nicht notwendig ist.

Sensors als Trigger notiert. Wird das Ziffernblatt also mit der rechten Maustaste angeklickt, schaltet die Transition, und die zugehörige Aktion wird ausgeführt, d. h., der Winkel des Stundenzeigers wird dem Ausdruck entsprechend neu gesetzt.

Ein Linksklick auf das Ziffernblatt aktiviert den Sensor *StartStop*. Dies führt zu einem Wechsel vom Zustand *ClockActive* zum Zustand *ClockInactive* und umgekehrt. Beim Verlassen des zusammengesetzten Zustands *ClockActive* werden alle untergeordneten Animationszustände verlassen, was zum abrupten Ende der Animationen führt, d. h., die Zeiger bleiben stehen. Wird wieder zurück zum Zustand *ClockActive* gewechselt, so wird die Animation entsprechend den Animationsanweisungen fortgesetzt.

Abb. 5.3 (rechts) zeigt die zusammengesetzte Pendeluhr. Auch die beschriebenen Animationen werden angedeutet. Die Diagramme in Abb. 5.1 und Abb. 5.2 definieren dabei (fast) alle Details, um eine derartige Darstellung inkl. ihrem Verhalten umzusetzen.

5.3 Erweiterung von UML

AML stellt eine Erweiterung von UML dar. Unter den verschiedenen Möglichkeiten, UML zu erweitern, wurde der „middleweight extension"-Mechanismus (vgl. [BH08]) gewählt. Bei diesem Mechanismus wird das grundlegende UML-Metamodell von einem neuen Metamodell importiert, um die darin enthaltenen Klassen erweitern bzw. spezialisieren zu können. Zusätzlich können OCL-Zusicherungen hinzugefügt werden, um bestimmte UML-Elemente oder auch den Sprachumfang an sich einzuschränken. Neben diesem Kapitel sind einige Details auch in Abschnitt B.1, S. 275, zu finden.

Für eine konkrete Umsetzung von AML im Rahmen dieser Arbeit wurde auf das EMF-basierte UML2-Projekt der Model Development Tools (MDT) für die Eclipse-Plattform aufgesetzt. Das darin enthaltene UML-Metamodell entspricht in den relevanten Punkten der offiziellen UML-Spezifikation. Es enthielt zum Zeitpunkt der Umsetzung aber noch zahlreiche Lücken, z. B. bei der Umsetzung der OCL-Zusicherungen, was u. a. darauf zurückzuführen ist, dass viele Zusicherungen nicht umgesetzt werden können [WD11].

Die Übersicht in Abb. 5.4 stellt die Spracheinheiten dar, die für AML definiert sind. Während die Sprachelemente im Paket *Structure* die Modellierung von strukturellen Elementen ermöglichen, werden die Sprachelemente in *Behavior* für die Verhaltensmodellierung eingesetzt. Die entsprechenden Elemente können jeweils innerhalb von UML-Klassen- bzw. Zustandsdiagrammen verwendet werden. Die Übersicht zeigt außerdem, welche Spracheinheiten der UML von *Structure* und *Behavior* benötigt werden. Grundlage bildet dabei UML in seiner Version 2.4 [UML11]. Diese Version ist auch Ausgangspunkt für alle in dieser Arbeit beschriebenen Sachverhalte bzgl. UML.

Obwohl UML von der OMG ständig weiterentwickelt wird, ist UML bekannt für problematische Spezifikationslücken. Arbeiten wie [SG99, RW99, FSKR05, Cra06] beschreiben einige dieser Probleme, v. a. auch in Bezug auf Verhaltensmo-

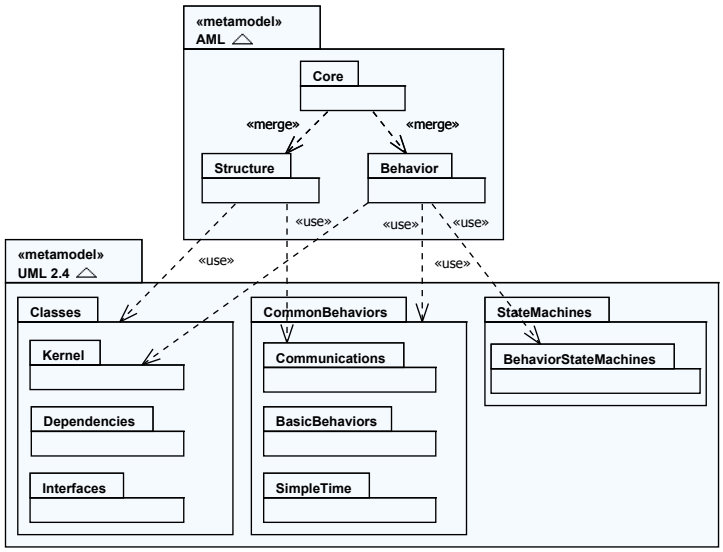


Abbildung 5.4: AML und verwendete Spracheinheiten

dellierung und Zustandsdiagramme. Daher beschäftigen sich zahlreiche Projekte damit, eine formale Teilmenge von UML oder ein „ausführbares“ UML zu schaffen, z. B. precise UML (pUML) [EK99], foundational UML (fUML) [FUM11] oder executable UML (xUML) [MB02]. Da sich unter diesen Projekten bislang kein breit akzeptierter Standard durchgesetzt hat, wurde AML trotzdem auf Basis von Standard-UML spezifiziert. Ausgehend von diesen Rahmenbedingungen handelt es sich bei AML also nicht um eine semantisch vollständig spezifizierte, widerspruchs- und fehlerfreie Sprache.

Da AML eine Erweiterung von UML darstellt, ist UML die Obermenge von AML. Es wurde allerdings nicht untersucht, ob jedes UML-Feature sinnvoll im Rahmen von AML-Modellen eingesetzt werden kann. Für die grundlegenden Ideen von AML ist nur eine Teilmenge von UML relevant. Für den Einsatz von AML sind v. a. die Einsatzgebiete und Möglichkeiten von UML-Klassen- und Zustandsdiagrammen von entscheidender Bedeutung, weshalb entsprechende Aspekte in den folgenden Unterabschnitten genauer betrachtet werden.

5.3.1 Verwendung von UML-Klassendiagrammen

Genau wie in der UML werden Klassendiagramme für die Erstellung des strukturellen Domänenmodells benutzt. Dabei dürfen konventionelle Elemente verwendet werden, die auch in Klassendiagrammen zum Einsatz kommen: *Klassen*, *Schnittstellen*, *Attribute*, *Operationen*, *Aufzählungstypen*, *primitive Datentypen* etc. Mittels (n-ären) *Assoziationen* und insbesondere *Aggregation* sowie *Komposition* können Beziehungen zwischen den Classifiern (wie z. B. Klassen) modelliert werden. Auch die üblichen Konzepte wie *Generalisierung*, *Schnittstellenrealisie-*

rung, *Sichtbarkeit*, *Multiplizität*, *abstrakte oder statische Klassen und Methoden*, *Neudefinition* (engl. redefinition) oder *Standardwerte* für Attribute stehen zur Verfügung. Außerdem können weitere Eigenschaften bzw. Schlüsselwörter einige Elemente genauer spezifizieren, z. B. *read-only* für Attribute, auf die nur lesend zugegriffen werden kann, *ordered* für geordnete Mengen oder Sequenzen u. v. m. Inzwischen ist auch die Object Constraint Language (OCL) [OCL12] fester Bestandteil der UML. Sie kann innerhalb von AML-Klassendiagrammen genutzt werden, um bspw. zusätzliche Randbedingungen oder die Definition von Methodenrümpfen im Modell unterzubringen.

Derartige Klassendiagramme sind inzwischen gemeinhin als Standardnotation in der objektorientierten Analyse und Design (OOAD) akzeptiert. Die Generierung von Code aus Klassendiagrammen wird ebenfalls oft eingesetzt und hat sich etabliert (vgl. Abschnitt 7.5, S. 261). Dies liegt nicht zuletzt daran, dass UML die Möglichkeiten von aktuellen objektorientierten Programmiersprachen weitestgehend abbildet. Allerdings sollte bei der Modellierung auf konkreter Ebene darauf geachtet werden, dass nur Sprachelemente verwendet werden, die auf der Zielplattform auch umgesetzt werden können. Im Rahmen dieser Arbeit wurde bspw. auf Mehrfachvererbung verzichtet (vgl. Abschnitt 7.2.1, S. 219).

5.3.2 Verwendung von UML-Zustandsdiagrammen

Um das Verhalten von bestimmen Classifiern (*uml::BehavioredClassifier*) und speziell die Animationen von Medienkomponenten zu modellieren, werden Zustandsdiagramme genutzt. Darin können die meisten von der UML bekannten Elemente genutzt werden: *Zustandsautomaten*, *Zustände*, *Pseudozustände*, *Transitionen* und *Regionen*. In diesem Zusammenhang werden allerdings nur Verhaltenszustandsautomaten genutzt und keine Protokollzustandsautomaten.

Ein größerer Unterschied besteht in der Menge der unterstützten Ereignisse. Mit der Einführung von Sensoren (siehe Abschnitt 5.4.5, S. 159) existieren in AML Möglichkeiten zur Verwendung von Ereignissen, die durch Aktivierung dieser Sensoren ausgelöst werden. Mit diesem Mechanismus und den verfügbaren Sensorarten sollen andere in UML existierende Mechanismen ersetzt werden.[2] Es sollten daher keine Ereignisse der folgenden Metaklassen modelliert werden: *CallEvent*, *AnyReceiveEvent*, *SignalEvent* und *ChangeEvent* (bzw. *MessageEvent*).

Außerdem sollte auf die Modellierung von „do“-Verhalten innerhalb von Zuständen verzichtet werden. Mit Animationszuständen und Animationsanweisungen existieren in AML andere Möglichkeiten, um das Verhalten einer Medienkomponente zu modellieren, während ein Zustand aktiv ist.

Wichtig sind in AML auch Zeitereignisse (*TimeEvent*). Zugehörige Trigger können genau wie in der UML an Zustandsübergängen notiert werden. Hierfür werden die Schlüsselworte „at“ und „after“ verwendet. Sie geben an, ob ein absoluter oder relativer Zeitpunkt modelliert wird. Der relative Zeitpunkt bezieht sich dabei auf den Zeitpunkt, wann der Quellzustand des Zustandsübergangs

[2] Der Einsatz von Sensoren ist allerdings nur in Zusammenhang mit Medienkomponenten möglich.

eingenommen wurde. Für den absoluten Zeitpunkt spielt die *Systemzeit* eine Rolle. Jedes modellierte System muss über eine solche Systemzeit verfügen, wobei zum Start des Systems der Zeitpunkt 0 angenommen wird. Weitere Details zum Zeitbegriff innerhalb von AML-Modellen folgen im nächsten Unterabschnitt. Zusätzlich zur Art des Zeitpunkts muss auch der Zeitpunkt selbst angegeben werden. Neben Konstanten als Zeitwerte können auch einfache mathematische Ausdrücke oder Aufrufe wirkungsfreier Methoden verwendet werden. Zusätzlich kann eine Zeiteinheit angegeben werden. Tab. 5.1 zeigt die in AML verfügbaren Einheiten.

ms, msec, msecond(s), millisecond(s)	Millisekunde(n)
s, sec, second(s), [ohne Angabe]	Sekunde(n)
m, min, minute(s)	Minute(n)
h, hour(s)	Stunde(n)
d, day(s)	Tag(e)

Tabelle 5.1: Unterstützte Zeiteinheiten

OCL kann innerhalb von AML-Zustandsdiagrammen genutzt werden, um Wächter von Transitionen zu spezifizieren. Auf diese Möglichkeit wird im späteren Verlauf der Arbeit aber verzichtet, da mit der Spracherweiterung AML/GT eine alternative Möglichkeit vorgestellt wird.

Ablaufsemantik des Zustandsautomaten

Die Ablaufsemantik der modellierten AML-Zustandsautomaten orientiert sich an der UML-Spezifikation. In Bezug zur Ereignisverarbeitung und der sogenannten *Run-to-completion*-Semantik sind folgende darin formulierten Aussagen auch für AML und den später präsentierten Ansatz von enormer Wichtigkeit:

> Event occurrences are detected, dispatched, and then processed by the state machine, one at a time. The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.
>
> The semantics of event occurrence processing is based on the *run-to-completion* assumption, interpreted as run-to-completion processing. Run-to-completion processing means that an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed. [. . .]
>
> The processing of a single event occurrence by a state machine is known as a *run-to-completion step*. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal activities (but not necessarily state (do) activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event occurrence will never

Priorität[a)]	Ereignis
!	AML-Sensoraktivierung[b)] (flüchtig) AML-Sensoraktivierung[b)] (dauerhaft) Zeitereignis Abschlussereignis *inv broken*-Ereignis[c)]

[a)] Im Falle mehrerer aktivierter Transitionen mit gleicher Priorität werden spezielle Regeln angewendet.
[b)] AML-Sensoren kann ein Prioritätswert zugewiesen werden, wodurch eine präzisere Priorisierung der Sensortrigger ermöglicht wird.
[c)] Dieser Trigger ist ein Sprachelement von AML/GT (siehe Abschnitt 6.5.2, S. 202).

Tabelle 5.2: Grundlegende Priorisierung von Transitionen

> be processed while the state machine is in some intermediate and inconsistent situation. [. . .]
>
> When an event occurrence is detected and dispatched, it may result in one or more transitions being enabled for firing. If no transition is enabled [. . .], the event occurrence is discarded and the run-to-completion step is completed.
>
> In the presence of orthogonal regions it is possible to fire multiple transitions as a result of the same event occurrence – as many as one transition in each region in the current state configuration. In case where one or more transitions are enabled, the state machine selects a subset and fires them. [. . .] The order in which selected transitions fire is not defined. ([UML11])

Darüber hinaus wird in der Spezifikation ein Spielraum für semantische Variationen bei der Abarbeitung von Ereignissen gelassen. So darf die Abarbeitung von Ereignissen und entsprechenden Zustandsübergängen bspw. Zeit benötigen. Für die AML wird daher klar festgelegt, dass für Zustandsübergänge und sämtliche damit verbundenen Aktionen kein Zeitaufwand angenommen wird (*Zero-time*-Semantik, vgl. [Cra06]).

Ereignispriorität

Die UML-Spezifikation erlaubt explizit, dass die Priorisierung von Ereignissen nach individuell festgelegten Regeln erfolgen darf (siehe oben). Für AML wurden daher Prioritäten definiert wie in Tab. 5.2 gezeigt.[3]

Falls mehrere Transitionen mit gleicher Priorität aktiviert sind, werden Transitionen, die von einem untergeordneten Zustand ausgehen, gegenüber Transitionen des übergeordneten Zustands bevorzugt (vgl. [UML11]). Stehen danach noch mehrere Transitionen zur Wahl, werden Transitionen mit Wächtern gegenüber

[3]Das Abschlussereignis (engl. *completion event*) wird darin anders priorisiert als erlaubt: „the completion event is dispatched before any other events in the pool“ ([UML11]).

Transitionen ohne Wächter mit Vorrang behandelt. Dies ist eine Erweiterung und wird nicht in der UML-Spezifikation erwähnt.

Ist auch nach obiger Vorgehensweise keine eindeutige Wahl möglich, definiert auch AML keine weitere Konfliktlösung. Eine Transition darf dann beliebig gewählt werden, egal ob durch einen deterministischen oder nicht-deterministischen Algorithmus.

5.4 AML-Kernelemente, Metamodell und Notation

Die in Abschnitt 5.2, S. 123, durch ein Beispiel eingeführten Elemente werden in diesem Abschnitt vollständig beschrieben und in Bezug auf das AML-Metamodell spezifiziert. Dazu gehört ebenfalls ein Überblick über wichtige Operationen und Zusicherungen der jeweiligen Metaklassen. Zusätzlich wird dargestellt, wie die Elemente in einem AML-Diagramm notiert werden. Den Abschluss jedes Unterabschnitts bilden Beispiele, die Bezug auf die animierten Sprachen in Abschnitt 2.3, S. 30, nehmen. Auf diese Weise wird gezeigt, wie einzelne Aspekte der Sprachen mittels AML modelliert werden können.

Zur genaueren Beschreibung werden einige AML-Konzepte auch formal dargestellt. Die Hilfsdefinitionen stellen allerdings eine stark vereinfachte Sicht auf Klassen, deren Attribute und Objekte dar. So werden z. B. Vererbung, Typhierarchien und zugehörige Mechanismen der UML nicht formal definiert. Außerdem wird darauf verzichtet, mögliche Beziehungen zwischen Klassen und damit auch zwischen Instanzen formal zu definieren, d. h., Attribute können nur primitive Werte annehmen und damit nicht auf andere Objekte referenzieren.

An dieser Stelle sei außerdem auf die in der Darstellung wichtige Unterscheidung zwischen Ausprägungen (von Klassen) und Instanzen hingewiesen. Im Folgenden umfasse eine „Ausprägung“ alle Daten (Attributwerte) eines Objekts einer bestimmten Klasse[4]. Die Begriffe Instanz und *Medienkomponenteninstanz* werden hingegen verwendet, wenn es sich um *Laufzeit-Objekte mit eigener Identität* handelt. So können zur Laufzeit mehrere Instanzen mit derselben Ausprägung, d. h. denselben Attributwerten, existieren.

Hilfsdefinitionen
Durch ein AML-Modell sei folgende Menge festgelegt:

- $\mathcal{T}$ bezeichne die endliche Menge einiger im AML-Modell spezifizierten Elemente und wird auch *Typmenge* genannt. Die Typmenge umfasst dabei die Menge der primitiven Datentypen $\mathcal{P} \subseteq \mathcal{T}$, die Menge der Klassen $\mathcal{C} \subseteq \mathcal{T}$ und damit auch die Menge der Medienkomponenten $\mathcal{M} \subseteq \mathcal{C}$.

Über $\mathcal{T}$ seien folgende Mengen definiert:

- $\mathcal{V}_{\mathcal{X}}$ bezeichne die Menge aller *Ausprägungen* der *Typen* in der Menge $\mathcal{X} \subseteq \mathcal{T}$.

[4] Im Falle eines primitiven Datentyps sei mit „Ausprägung“ schlicht der Wert gemeint.

- $\mathcal{A}_{\mathcal{X}}$ bezeichne die Menge aller Attribute der Klassen in der Menge $\mathcal{X} \subseteq \mathcal{C}$.

Über $\mathcal{T}$ seien zusätzlich folgende Typfunktionen definiert:

- $vtype : \mathcal{V}_{\mathcal{T}} \to \mathcal{T}$ ordnet jeder Ausprägung $v \in \mathcal{V}_{\mathcal{T}}$ ihren Typ $tp \in \mathcal{T}$ zu. Dabei muss gelten $v \in \mathcal{V}_{\{tp\}}$.

- $atype : \mathcal{A}_{\mathcal{C}} \to \mathcal{P}$ ordnet jedem Attribut $attr \in \mathcal{A}_{\mathcal{C}}$ den dafür spezifizierten primitiven Datentyp $pr \in \mathcal{P}$ zu.

Damit können die Attributwerte einer Klassen-Ausprägung abgefragt werden:

- $saval : \mathcal{V}_{\mathcal{C}} \times \mathcal{A}_{\mathcal{C}} \rightharpoonup \mathcal{V}_{\mathcal{P}}$ ordnet jedem Attribut $attr$ von jeder Klassen-Ausprägung $v \in \mathcal{V}_{\mathcal{C}}$, d. h., $saval$ ist definiert, falls $attr \in \mathcal{A}_{\{vtype(v)\}}$, einen *(statischen) Attributwert* $x \in \mathcal{V}_{\mathcal{P}}$ zu. Es muss gelten $vtype(x) = atype(attr)$, da der Attributtyp berücksichtigt werden muss.

Zur Laufzeit eines entsprechenden Systems über $\mathcal{T}$ ist weiterhin definiert:

- $sys = (\mathcal{OBJ}, val, \ldots)$ bezeichne den *aktuellen Systemzustand*. Er enthält eine endliche Menge von *Instanzen* $\mathcal{OBJ}$, eine Funktion $val : \mathcal{OBJ} \to \mathcal{V}_{\mathcal{C}}$, die jeder Instanz $obj \in \mathcal{OBJ}$ ihre aktuelle Ausprägung $v \in \mathcal{V}_{\mathcal{C}}$ zuordnet, und weitere Elemente, die in diesem Abschnitt noch beschrieben werden. $\mathcal{SYS}_{\mathcal{T}}$ bezeichne die Menge aller Systemzustände über $\mathcal{T}$.

- $\mathcal{OBJ}_{\mathcal{X}} = \{obj \in \mathcal{OBJ} \mid val(obj) \in \mathcal{V}_{\mathcal{X}}\}$ bezeichne bei gegebenem Systemzustand sys die Menge aller Instanzen der Klassen in der Menge $\mathcal{X} \subseteq \mathcal{C}$.

Zusätzlich werden folgende Symbole verwendet:

- $\phi \notin \mathbb{T}^{\omega}$ bezeichne den *undefinierten Zeitpunkt*. Er ist ein Element der Menge $\mathbb{T}^{\phi} = \mathbb{T} \cup \{\phi\}$.

- $\psi \notin \mathcal{V}_{\mathcal{T}}$ bezeichne den *undefinierten Wert*. Er ist ein Element der Menge $\mathcal{V}_{\mathcal{X}}^{\psi} = \mathcal{V}_{\mathcal{X}} \cup \{\psi\}$, wobei $\mathcal{X} \subseteq \mathcal{P}$.

Außerdem werden folgende Standardfunktionen genutzt:

- Sei $(K, \leq)$ eine partielle Ordnung. Das Element $x \in M$ einer Teilmenge $M \subseteq K$ ist das *größte Element*, falls für alle $y \in M$ gilt $y \leq x$. Das größte Element wird auch kurz als $max\ M$ notiert. Analog steht $min\ M$ für das kleinste Element. Im Folgenden kann bei Verwendung davon ausgegangen werden, dass ein größtes bzw. kleinstes Element existiert.

- Die Abrundungsfunktion ordnet einem $x \in \mathbb{R}$ die größte ganze Zahl zu, die kleiner oder gleich x ist: $\lfloor x \rfloor = max\ \{y \in \mathbb{Z} \mid y \leq x\}$. △

5.4.1 Medienkomponenten

Den Kern des statischen Teils von AML bilden sogenannte *Medienkomponenten*. Sie sind eine Spezialisierung und Erweiterung von UML-Klassen und orientieren sich an deren Notation. Medienkomponenten werden anstelle von gewöhnlichen Klassen verwendet, falls Instanzen eine mediale Repräsentation besitzen. Der Begriff „mediale Repräsentationen“ umfasst dabei ein breites Spektrum, beginnend von visuellen Erscheinungsformen (z. B. Text, stehende und bewegte Bilder, 3D-Objekte) über Akustik (z. B. Sprache, Musik, Soundeffekte) bis hin zu weiteren Formen (z. B. Haptik), die in der Informationstechnologie noch kaum erschlossen sind. Eine Übersicht und Einordnung möglicher Repräsentionsformen wird in [Ste00] präsentiert.

Trotz der vielfältigen Möglichkeiten soll sich der Medienkomponentenbegriff in dieser Arbeit auf zweidimensionale Formen und Bilder beschränken, d. h., es wird von Medienkomponenten mit visueller 2D-Darstellung ausgegangen. Dies ist auch die wichtigste Repräsentationsform für (zweidimensionale) animierte Sprachen.

Die konkrete Repräsentation bzw. Darstellung einer einzelnen Medienkomponente ist aus dem Modell nicht ersichtlich. Das Zeichnen einer Medienkomponente übernimmt ein Algorithmus, der bei der Implementierung der Medienkomponente angegeben werden muss. Es ist aber möglich, dass ein AML-Basisframework grafische Primitiven in Form von Medienkomponenten inkl. Implementierung zur Verfügung stellt (siehe Abschnitt 5.5, S. 182ff). Wird eine solche grafische Primitive bspw. als *Basismedienkomponente* verwendet, so wird auch das Aussehen „vererbt“.

Aufgrund ihrer Ableitung von Klassen, erlauben Medienkomponenten die Verwendung von normalen Klassenmerkmalen, wie etwa Attribute, Operationen oder Konstruktoren. Auch Assoziationen, Generalisierung, Schnittstellenrealisierung, Instanziierung und ähnliche Konzepte werden in gewohnter Weise unterstützt. Neben den bekannten Techniken ermöglichen Medienkomponenten bspw. die Modellierung von inneren Merkmalen, die den strukturellen Aufbau der Medienkomponente in Form von Unterkomponenten beschreiben (siehe Abschnitt 5.4.2, S. 137). Auch die sogenannten Animatoren, die zur Spezifikation von Animationsabläufen dienen, können nur innerhalb von Medienkomponenten eingesetzt werden (siehe Abschnitt 5.4.3, S. 143). Außerdem können Sensoren für eine Medienkomponente modelliert werden, um das Verhalten der Medienkomponente zu steuern (siehe Abschnitt 5.4.5, S. 159).

Metamodell Abb. 5.5 zeigt die Metaklasse *MediaComponent* als Ableitung von *uml::Class*. Gemäß Metamodell kann eine Medienkomponente beliebig viele innere Merkmale (*ownedInnerProperty*) und beliebig viele Animatoren (*ownedAnimator*) besitzen. Auch jeder Sensor (*sensor*) muss einer Medienkomponenten (in der Rolle als Eigentümer) zugeordnet werden. Die Operationen der Metaklasse *MediaComponent* sollen den Zugriff auf bestimmte Elementmengen vereinfachen. *getAllAnimators* vereint bspw. die Animatoren der aktuellen Medienkomponente mit den Animatoren der Basismedienkomponenten (siehe Listing B.1/4). △

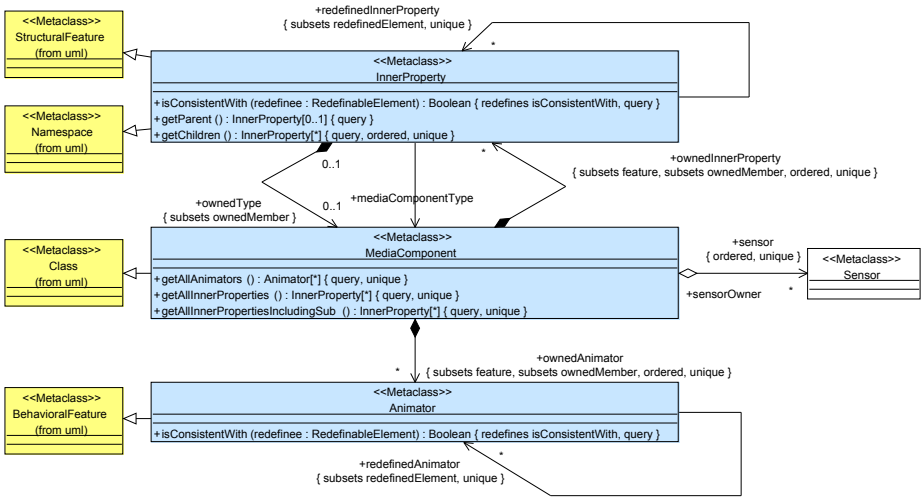


Abbildung 5.5: AML-Metamodell: Medienkomponenten und Innere Merkmale

Zusicherungen Medienkomponenten können lediglich von anderen Medienkomponenten abgeleitet werden, nicht jedoch von gewöhnlichen UML-Klassen (siehe Listing B.2/3). Eine Schnittstellenrealisierung durch Medienkomponenten ist allerdings erlaubt. △

Notation Medienkomponenten werden notiert wie gewöhnliche UML-Klassen. Um die Klasse allerdings als Medienkomponente zu markieren, wird in die rechte obere Ecke ein spezielles Stereotyp-Icon gezeichnet (siehe *MediaComponent2D* in Abb. 5.6). Alternativ kann auch die UML-typische Schreibweise mit Guillemets genutzt werden: *«MediaComponent»*. Zusätzlich kann bei Medienkomponenten neben den typischen Compartments für Attribute und Operationen ein zusätzliches Compartment für innere Merkmale verwendet werden. △

Beispiel 5.1 (Medienkomponente – Stellen/Transitions-Pfeil)
B/E-Netze bestehen aus unterschiedlichen visuellen Elementen. Stellen werden als Kreise dargestellt, Transitionen benötigen Blöcke, und für die Kanten dazwischen sollen Pfeile verwendet werden. Diese Elemente werden daher als Medienkomponenten modelliert. Abb. 5.6 (links) zeigt die für Kanten vorgesehene Medienkomponente *PTArrow*. Sie ist von zwei anderen Medienkomponenten *MediaComponent2D* und *Arrow* abgeleitet.

MediaComponent2D definiert dabei Grundeigenschaften für zweidimensionale Visualisierungen. Es werden Attribute für Positionierung und Skalierung unterstützt, ebenso wie Attribute, welche die Dimensionen oder den Ankerpunkt für die Positionierung bestimmen. Eine schematische Darstellung der Medienkomponente wird in Abb. 5.6 (rechts) abgebildet. Das Schema zeigt die Ideen, auf der die Darstellung der Medienkomponente beruht. Es veranschaulicht, wie Attribute

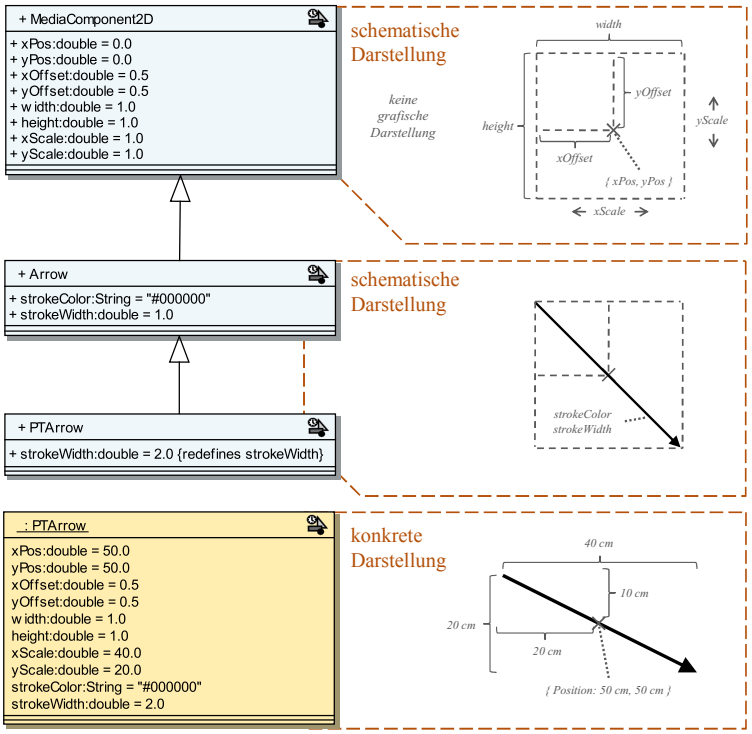


Abbildung 5.6: Stellen/Transitions-Pfeil als Medienkomponente

die Lage der Komponente bestimmen sollen und skizziert so einen Teil des implementierten Algorithmus, der für die Visualisierung der Medienkomponente sorgt. Details über grundlegende Medienkomponenten wie *MediaComponent2D*, deren Attribute und weitere Bestandteile werden in Abschnitt 5.5, S. 175, ausführlich behandelt.

Arrow spezialisiert das Grundschema von *MediaComponent2D*, indem festgelegt wird, dass ein Pfeil mit entsprechender Lage und Ausdehnung gezeichnet werden soll. Zusätzlich werden weitere Attribute wie *strokeWidth* und *strokeColor* angeboten. Diese ermöglichen, Farbe und Stärke des Pfeils festzulegen. Die konkret für Pfeile zwischen Stellen und Transitionen benötigte Medienkomponente *PTArrow* ändert an diesem Schema nichts mehr. Sie definiert das Attribut *strokeWidth* lediglich neu, um die Standardstärke für Stellen/Transitions-Pfeile auf den Wert 2,0 zu setzen. Ob ein solcher Wert im Rahmen einer Applikation änderbar ist oder nicht, wird an dieser Stelle jedoch nicht bestimmt. Es wäre bspw. sinnvoll, wenn sich Position und Größe eines Pfeils innerhalb eines Editors für B/E-Netze beliebig ändern lässt, die Pfeilstärke allerdings fest definiert ist.

Abb. 5.6 (ganz unten) zeigt schließlich eine *PTArrow*-Medienkomponenteninstanz. Rechts neben dieser ist dargestellt, wie der Pfeil mit den angegebenen Beispielwerten konkret in einer Applikation dargestellt wird. △

5.4.2 Innere Merkmale

Medienkomponenten können mehrere *untergeordnete Medienkomponenten* (auch *Unterkomponenten*) besitzen, welche die innere Struktur der *übergeordneten Medienkomponente* festlegen (vgl. Kompositum-Entwurfsmuster [GHJJ96]). Zur eigenen grafischen Repräsentation der übergeordneten Medienkomponenten gehören dann auch alle grafischen Repräsentationen der Unterkomponenten, d. h., eine übergeordneten Medienkomponente wird immer zusammen mit ihren Unterkomponenten gezeichnet (vgl. Abb. 5.3, S. 125).

Die erwähnte innere Struktur kann mit Hilfe von *inneren Merkmalen* modelliert werden, die eine ähnliche Semantik besitzen wie UML-Attribute oder genauer UML-Parts.[5] Parts werden visuell jedoch typischerweise nur im Rahmen von Kompositionsstrukturdiagrammen eingesetzt. Innere Merkmale werden hingegen direkt im Klassendiagramm in einem speziell für diesen Zweck vorgesehenen Compartment einer Medienkomponente modelliert. Zudem bieten innere Merkmale gegenüber Parts zusätzliche Eigenschaften und Besonderheiten, die im Folgenden noch vorgestellt werden.

Multiplizität

Genau wie gewöhnliche Attribute unterstützen innere Merkmale die Spezifikation von Multiplizitäten. Beispielsweise kann eine Medienkomponente namens *Auto* ein inneres Merkmal *reifen:Reifen[4]* besitzen. Die Multiplizität des Merkmals beträgt 4 und als Typ ist *Reifen* definiert, welcher wiederum eine Medienkomponente sein muss, da für innere Merkmale nur Medienkomponenten als Typ verwendet werden dürfen. Jede Medienkomponenteninstanz von *Auto* besitzt somit genau vier untergeordnete Medienkomponenteninstanzen vom Typ *Reifen*.

Unterkomponentenhierarchie

Zusätzlich können innere Merkmale hierarchisch modelliert werden, was im AML-Diagramm innerhalb der Medienkomponente entsprechend veranschaulicht wird. In dieser Hierarchie können neben *direkten Unterkomponenten* auch Unterkomponenten von Unterkomponenten (im Rahmen derselben übergeordneten Medienkomponente) modelliert werden. Auf diese Weise entsteht ein Hierarchiebaum (vgl. *PendulumClock* in Abb. 5.1, S. 124).

Eine derartige Hierarchisierung wird verwendet, um die Lage der einzelnen Medienkomponenten und insbesondere deren Grafiken in Abhängigkeit zu stellen. Eine übergeordnete Medienkomponente beeinflusst dabei die Lage aller untergeordneten Medienkomponenten. Wird die übergeordnete Medienkomponenteninstanz neu positioniert, gedehnt, gedreht etc., so gilt dies in selber Weise auch für die untergeordnete Medienkomponenteninstanz. Anders formuliert ist die Lage der Untergeordneten relativ zur Lage der Übergeordneten. Außerdem gilt, dass die Existenz einer Untergeordneten abhängig von der Existenz der

[5] UML-Parts sind UML-Attribute, die spezielle Bedingungen erfüllen. Sie spezifizieren Klasseninstanzen, die durch Komposition zu ihrem Besitzer (*uml::StructuredClassifier*) gehören [UML11].

Übergeordneten ist. Die hierarchischen Beziehungen zwischen den Elementen entsprechen also automatisch dem UML-Aggregationstyp „Komposition“.

Die vollständige Unterkomponentenhierarchie ist allerdings nicht immer direkt aus dem Compartment einer einzelnen Medienkomponente ablesbar. Durch Vererbung, Neudefinition und verwendete Typen kann die tatsächliche Hierarchie komplexer sein als im Diagramm lokal bei einer Medienkomponente dargestellt.

Beispiel 5.2 (Unterkomponentenhierarchie)
Das Diagramm in Abb. 5.7 (a) soll diesen Sachverhalt demonstrieren. Die vollständige Medienkomponentenhierarchie, die daraus nicht sofort ersichtlich ist, wird in (c) abgebildet.

Durch das Diagramm in (a) wird eine Medienkomponente *A* spezifiziert, die ein inneres Merkmal *b* enthält. Obwohl nicht direkt in *A* notiert, würde eine Medienkomponenteninstanz von *A* neben Medienkomponenteninstanz *b* aber noch weitere Medienkomponenteninstanzen besitzen, nämlich *m1* und *m2*. Dies ist der Fall, da *b* die Medienkomponenteninstanzen *m1* und *m2* besitzt, denn für *b* ist der Typ *B'* modelliert, der *m2* als direkte Unterkomponente und *m1* (die direkte Unterkomponente von *B*) durch Vererbung erhält.[6]

Geht man nun davon aus, dass *B'* nur modelliert wird, um an einer Stelle, nämlich innerhalb von *A*, verwendet zu werden, dann gibt es eine alternative Notation zu (a). Die Medienkomponenten *A* in Abb. 5.7 (a) und (b) führen quasi zur gleichen Struktur (c). In (b) wird allerdings die verkürzte Schreibweise angewandt, indem in *A* mehrere Ebenen für innere Merkmale notiert werden. Als Typ von *b* ist zwar *B* angegeben, allerdings wird im (abstrakten) AML-Modell ein von *B* abgeleiteter Typ zugewiesen. Dieser Typ ist in der Regel ohne Namen und wird im Rahmen dieser Arbeit auch *implizite Medienkomponente* genannt.[7] Das zusätzliche innere Merkmal *m2* ist nur Bestandteil von dieser impliziten Medienkomponente und nicht etwa von *B*.

Außerdem wird in Abb. 5.7 (a) dem Attribut *attr* mittels Neudefinition ein neuer Standardwert für *B'* zugewiesen. Die Darstellung in (b) verwendet hierfür ebenfalls eine äquivalente, aber verkürzte Schreibweise mit Schlüsselwort *with* direkt unter dem inneren Merkmal. △

Neudefinition von inneren Merkmalen

Neudefinition wird in objektorientierten Systemen v. a. bei Operationen im Rahmen von Polymorphie und dynamischer Bindung häufig verwendet. Dabei kann Neudefinition auch bei Attributen sinnvoll sein, z. B. um deren Standardwerte neu zu setzen (vgl. Beispiel 5.2). Problematisch ist aber, wenn bspw. Typen der Attribute verändert werden sollen. In diesem Fall wäre es möglich, dass auf bestimmte Attribute und Datentypen ausgelegte Algorithmen der Basisklassen

[6] Diese Prinzipien gelten auch für andere Elemente wie bspw. Sensoren, die allerdings erst in Abschnitt 5.4.5, S. 159, genauer behandelt werden. Im Beispiel wird ein Sensor *S* für *B'* modelliert (a). Er ist dadurch ebenfalls Teil einer Instanz von *A*, wie in (c) gezeigt.

[7] In den Beispieldiagrammen mit Medienkomponenteninstanzen wird der Name der Medienkomponente verwendet, von der abgeleitet wird, und das Zeichen ' angehängt.

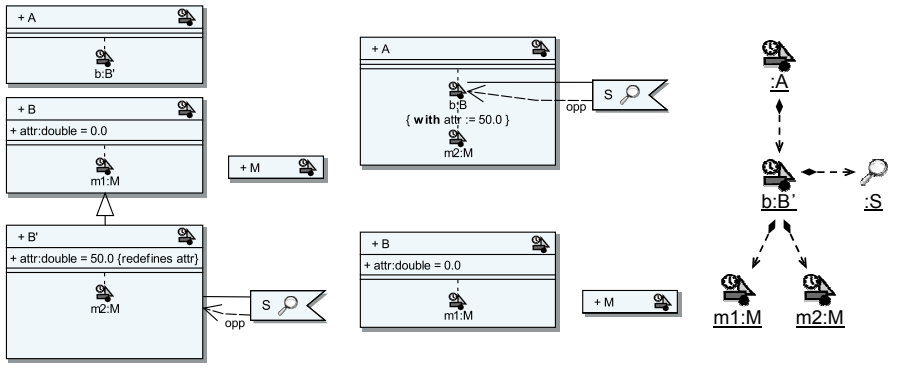


Abbildung 5.7: Medienkomponentenhierarchie und Darstellungsmöglichkeiten

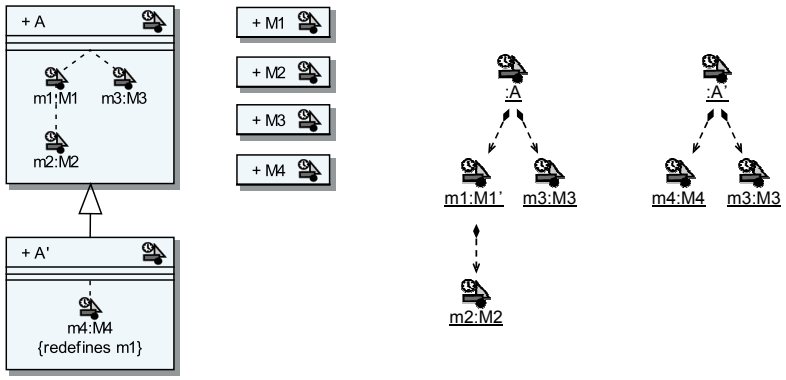
für Unterklassen nicht mehr funktionieren. UML verbietet daher derartige Neudefinitionen, was im Rahmen der Metaklassen-Operation *isConsistentWith* und Zusicherungen auch überprüft wird (vgl. [UML11]).

Eine Neudefinition von inneren Merkmalen ist genau wie bei Operationen und Attributen ebenfalls möglich. Bei einem inneren Merkmal (eigentlich eine spezielle Art von Attribut) kann es ebenfalls zu den erwähnten Problemen kommen, nämlich wenn dessen Typ bei der Neudefinition verändert wird. Trotzdem ist dies nicht generell verboten. Die Überlegung ist, dass innere Merkmale vordergründig grafische Aspekte repräsentieren, bei denen es durchaus Sinn machen kann, diese vollständig neu zu definieren. Beispielsweise wäre es möglich, dass ein Rechteck – der Typ eines inneren Merkmals – durch Neudefinition in der Unterklasse als Kreis definiert wird. Derartige Neudefinitionen sind aber nur unter bestimmten Voraussetzungen und mit Einschränkungen möglich. Im Rahmen dieser Arbeit soll darauf aber nicht weiter eingegangen werden.

Beispiel 5.3 (Neudefinition von Inneren Merkmalen)
Abb. 5.8 (a) zeigt, wie ein inneres Merkmal neu definiert werden kann. In Medienkomponente *A'*, die von Medienkomponente *A* abgeleitet wird, ersetzt *m4* das innere Merkmal *m1* komplett. Dies führt dazu, dass weder *m1* noch *m2* Teil einer Medienkomponenteninstanz von *A'* sind. Eine Medienkomponenteninstanz von *A'* besitzt lediglich die Medienkomponenteninstanzen *m3* und *m4*, wie in Abb. 5.8 (b) dargestellt. △

Metamodell Abb. 5.5, S. 135, enthält die Metaklasse für innere Merkmale namens *InnerProperty*. Sie ist, wie die Metaklasse von UML-Attributen, von *uml::StructuralFeature* abgeleitet.

Als Typ muss dem inneren Merkmal eine Medienkomponente zugewiesen werden. Handelt es sich dabei um eine implizite Medienkomponente, so ist diese abgeleitete Medienkomponente im inneren Merkmal enthalten (*ownedType*). Um



(a) (b)

Abbildung 5.8: Neudefinition von inneren Merkmalen

dies zu erlauben, müssen innere Merkmale auch die Fähigkeiten eines *uml:: Namespace* unterstützen.

Die Abbildung verdeutlicht auch, dass beliebig viele innere Merkmale durch Komposition in einer Medienkomponenten enthalten sein können. Das Attribut *ownedInnerProperty* bezieht allerdings nur innere Merkmale ein, die direkte Unterkomponenten der Medienkomponente darstellen. Um den vollständigen Baum der inneren Merkmale im Kontext der Medienkomponente zu analysieren, können die Operationen *getParent* und *getChildren* von *InnerProperty* genutzt werden. Diese Operationen helfen beim Traversieren der Hierarchie (siehe Listing B.1/51 und Listing B.1/61).

Die Operation *getAllInnerProperties* bezieht im Gegensatz zu *ownedInner-Property* auch Vererbung, Neudefinition, Merkmaltyp etc. in die Ereignismenge ein (siehe Listing B.1/17). Die Operation *getAllInnerPropertiesIncludingSub* liefert zusätzlich sogar innere Merkmale, die indirekt ein Teil der Medienkomponente sind, sich also tiefer in der Hierarchie befinden (siehe Listing B.1/31). △

Zusicherungen Es ist nicht möglich, statische innere Merkmale zu modellieren, da innere Merkmale immer zu einer Medienkomponenteninstanz gehören (siehe Listing B.2/7). Auch weitere Zusicherungen für innere Merkmale, die zum Teil schon erwähnt wurden, sind in Listing B.2 zu finden. △

Notation Innere Merkmale werden in einem eigenen Compartment innerhalb der Medienkomponente notiert. Die inneren Merkmale werden dort in einer Baumstruktur angeordnet, wobei die Medienkomponente selbst die Wurzel des Baums darstellt (vgl. *PendulumClock* in Abb. 5.1, S. 124). Nur die Merkmale der obersten Reihe – also mit einer Knotentiefe von 1 – gehören direkt zur Medienkomponente (*ownedInnerProperty*). Die darunter liegenden inneren Merkmale gehören immer zu einer impliziten Medienkomponente, die dann im jeweiligen Fall notwendig ist.

Die Knoten des Baumes werden mit dem Stereotyp-Icon der entsprechenden Medienkomponenten dargestellt.[8] Als Kanten zwischen den einzelnen Knoten (die Medienkomponente als Wurzelknoten eingeschlossen) werden gestrichelte Linien verwendet. Direkt unter dem inneren Merkmal werden Name, Typ, Multiplizität etc. des inneren Merkmals in der von Attributen bekannten Schreibweise festgehalten.

Zusätzlich ist es mittels kompakter Beschriftungen möglich, die Standardwerte von Attributen des verwendeten Medienkomponententyps neu zu definieren. Auch in diesem Fall ist eine implizite Medienkomponente notwendig, um darin die entsprechenden Attribute neu zu definieren. Die Kurzsyntax hierfür lautet „*attributname* := *konstanter Wert*“. Mehrere Neudefinitionen werden dabei kommasepariert aufgelistet. Der Text wird außerdem durch das Schlüsselwort *with* eingeleitet und in einem „{ }“-Block eingebettet. △

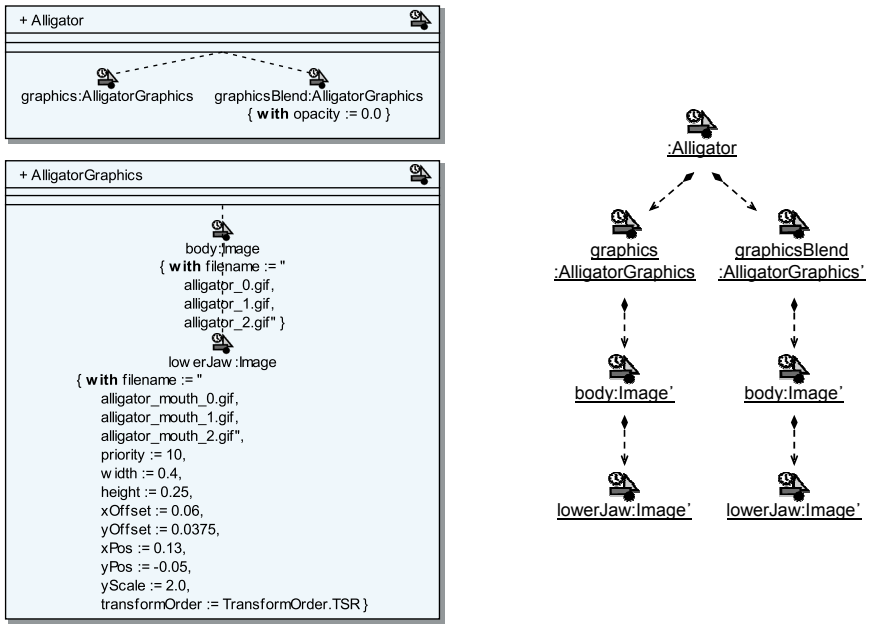
Beispiel 5.4 (Innere Merkmale von Alligator Eggs)
Innere Merkmale werden bspw. für die Spezifikation von Alligatoren der Sprache Alligator Eggs eingesetzt. Abb. 5.9 zeigt dabei, wie die Medienkomponente *Alligator* definiert werden kann. Die Darstellung nutzt die höchstmögliche Detailstufe für AML-Diagramme, d. h., alle Details wie Koordinaten und Dateinamen werden exakt festgelegt.

Die Medienkomponente *Alligator* enthält die inneren Merkmale *graphics* und *graphicsBlend*, jeweils vom Typ *AlligatorGraphics*. Diese Medienkomponente besteht wiederum aus zwei Unterkomponenten *body* und *lowerJaw*, wobei *lowerJaw* als Unterkomponente relativ zu *body* positioniert wird. Beide Unterkomponenten erhalten ihre visuelle Repräsentation durch Bilder.[9] Für *Image* können mehrere Dateinamen angegeben werden, was dafür genutzt werden kann, den Alligator in verschiedenen Farben anzuzeigen, d. h., die Dateien müssen dieselbe Grafik in unterschiedlichen Farbtönen enthalten. Durch das Attribut *frame* von *Image* kann dann die Farbe des Alligators gesetzt werden. So wird bei einem Wert von 0 ein roter Alligator angezeigt werden, bei 1 ein grüner Alligator etc.

Abb. 5.10 (a) zeigt beide Medienkomponenten in ihrer grafischen Repräsentation mit den spezifizierten Standardwerten aus Abb. 5.9. Die Werte sind dabei so gewählt, dass die Grafik im Seitenverhältnis 1 : 1 abgebildet wird. Sie kann später mit beliebigen Werten skaliert werden. Es wird ebenfalls gezeigt, welche Referenzpunkte (X) durch die Angaben festgelegt und wie beide Komponenten zueinander positioniert werden. Diese Struktur ist hinsichtlich der späteren Nutzung und den erforderlichen Animationen bewusst gewählt worden. Abb. 5.10 (b) zeigt, welche Attribute der Medienkomponente typischerweise in einer Umgebung für Alligator Eggs verändert werden. Zum einen können Alligatoren frei positioniert und skaliert werden. Zum anderen können einzelne Teile gedreht werden, was bspw. im Rahmen der β-Reduktion genutzt werden muss.

[8] Die Sprache AML definiert lediglich ein derartiges Stereotyp-Icon. Die Spracherweiterung AML/GT (siehe Kapitel 6) führt jedoch weitere Icons für spezielle Medienkomponenten ein.

[9] Der Typ *Image* wird in Abschnitt 5.5, S. 182ff, mit allen Attributen vorgestellt



(a) (b)

Abbildung 5.9: Alligator-Medienkomponente und innere Merkmale

Bei der β-Reduktion bewegt der fressende Alligator sein Maul, was mit einer Änderung des Rotationswinkels des Unterkiefers erreicht wird (Attribut *graphics.body.lowerJaw.angle*). Bei einem sterbenden Alligator dreht sich der ganze Alligatorkörper (Attribut *graphics.body.angle*). Dies soll inklusive Unterkiefer geschehen, weshalb der Unterkiefer als Unterkomponente des Körpers definiert wird. Dass sich der Unterkiefer dabei langsam schließt, kann ebenfalls realisiert werden. Beide Animationen werden in Abb. 5.10 (c) illustriert.

Warum die Medienkomponente *Alligator* zwei innere Merkmale *graphics* und *graphicsBlend* gleichen Typs spezifiziert, lässt sich mit der Notwendigkeit von Farbüberblendungen während einer α-Konversion begründen. Diese lässt sich realisieren, indem die durch *AlligatorGraphics* spezifizierte Grafik doppelt vorhanden ist. Standardmäßig ist allerdings nur *graphics* sichtbar. Das innere Merkmal *graphicsBlend* wird hingegen komplett ausgeblendet. Dies wird durch das Setzen des *opacity*-Standardwerts (siehe Tab. 5.6, S. 177) auf 0,0 umgesetzt. Sobald eine Farbüberblendung notwendig ist, werden die Unterkomponenten von *graphicsBlend* zunächst auf die neue Farbe gesetzt. Das kontinuierliche Senken des *opacity*-Werts von *graphics* von 1,0 auf 0,0 und das kontinuierliche Erhöhen des *opacity*-Werts von *graphicsBlend* von 0,0 auf 1,0 entspricht dann der Animation einer Farbüberblendung. △

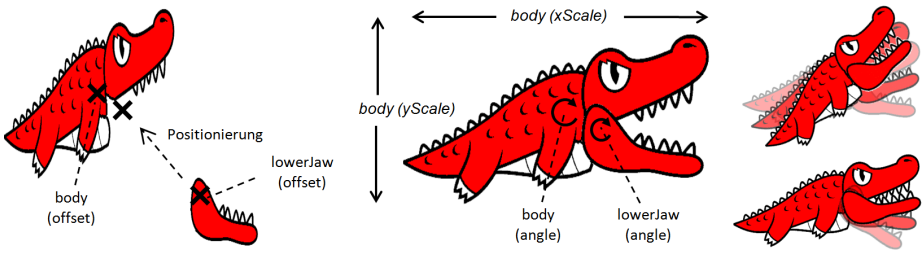


Abbildung 5.10: Medienkomponente *AlligatorGraphics*

5.4.3 Animatoren

Im Rahmen von AML wird Animation als eine bereits zu deren Beginn vollständig festgelegte, ständige Wertänderung eines Attributs über einen bestimmten Animationszeitraum definiert, wobei der Verlauf der Wertänderung inkl. Animationszeitraum von einem sogenannten *Animator* bestimmt wird. Animatoren sind dabei zunächst als abstraktes Konzept zur Beschreibung verschiedener Animationsarten und -verläufe zu verstehen. Sie müssen im AML-Modell als Element einer Medienkomponente, ähnlich wie UML-Operationen, modelliert werden und stellen auf diese Art einen wiederverwendbaren Funktionsblock für Wertänderungen bzw. Animationen dar. Die Implementierung eines Animators erfolgt, ähnlich wie bei Operationen, nicht im AML-Modell selbst.

Ein spezifizierter Animator kann zur Systemlaufzeit instanziiert und einem Attribut zugewiesen werden. Die gleichzeitige Zuweisung mehrerer *Animatorinstanzen* auf ein Attribut ist dabei nicht möglich. Ein Attribut mit zugeordneter Animatorinstanz wird auch als *animiert* bezeichnet, denn die Animatorinstanz bestimmt den Verlauf einer zeitabhängigen Wertänderung. Hierfür ordnet die Animatorinstanz dem Attribut zu jedem Zeitpunkt innerhalb des Animationszeitraums einen Wert zu. Der entsprechende Verlauf kann zusätzlich durch individuelle *Animatorparameter* beeinflusst werden, die bereits bei der Erzeugung der Animatorinstanz angegeben werden müssen.

Beeinflusst eine Animatorinstanz den Attributwert einer grafischen Eigenschaft der Medienkomponente, so resultiert dies bei ständiger Visualisierung auch in grafischer Animation. Die Animation des Attributs für die y-Position einer Medienkomponenteninstanz kann bspw. zu einer kontinuierlichen Bewegung entlang der y-Achse führen. Abb. 5.11 zeigt die Visualisierung der Murmel aus Avalanche zu unterschiedlichen Zeitpunkten. Ihrem Attribut *yPos* (y-Position) wurde eine Animatorinstanz zugeordnet, die den Wert gemäß einer beschleunigten Bewegung (bzw. dem freien Fall) im Laufe der Zeit ändert.

Im Folgenden werden einige Begriffe hinsichtlich Animatoren formal eingeführt und erläutert. Die ersten Definitionen zeigen, was zur vollständigen Spezifikation von Animatoren notwendig ist. Danach folgen Definitionen bzgl. der Erzeugung

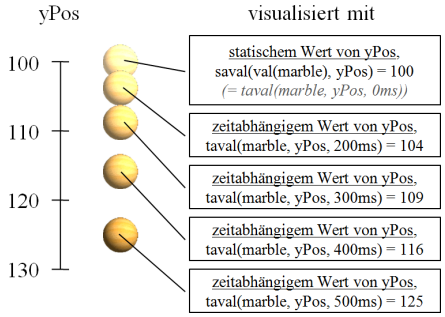


Abbildung 5.11: Animation durch Wertänderung

und Verwendung von Animatorinstanzen während der Laufzeit. Abschließend wird beschrieben, wie Animatoren in AML konkret modelliert werden.

Vollständige Spezifikation von Animatoren

Zur Spezifikation eines Animators müssen neben dem Eigentümer eine *Animatorsignatur* und zwei Funktionen angegeben werden. Diese Funktionen legen die Animation in Abhängigkeit einer *Animatorkonfiguration* fest. Diese muss bei der Erzeugung einer Animatorinstanz angegeben werden und zur Signatur kompatibel sein. Es folgen die formalen Definitionen und Erläuterungen zu diesen Begriffen.

Definition 5.1 (Animatorsignatur)
Gegeben sei eine Typmenge $\mathcal{T}$. Eine *Animatorsignatur* über $\mathcal{T}$ ist ein Tripel $sig = (\mathcal{ST}, \diamondsuit, tp)$, wobei

- $\mathcal{ST} \subseteq \mathcal{P}$ die Menge der durch den Animator *animierbaren Datentypen* festlegt.
- $\diamondsuit : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ ist eine zweistellige Boolesche Funktion, die spezifiziert, ob der undefinierte Wert ψ und der undefinierte Zeitpunkt ϕ als Angaben von Zielwert und Dauer zulässig sind.
- $tp = (tp_1, tp_2, \ldots, tp_{tpn}) \in \mathcal{T}^{tpn}$ ist das Tupel der *Animatorparametertypen* mit $tpn \in \mathbb{N}_0$ als Anzahl der Parameter.

$\mathcal{SIG}_{\mathcal{T}}$ bezeichne die Menge aller Animatorsignaturen über $\mathcal{T}$. △

Die Signatur legt fest, für welche Typen und damit Attribute der Animator anwendbar ist. So können Animatoren bspw. für Zahlen, Farben, Formen, Zeichenketten o. ä. anwendbar sein. Die Signatur legt auch fest, ob der Animator zur Berechnung der Animation in der Animatorkonfiguration die Angabe von Zielwert und Dauer benötigt (genaueres in den folgenden beiden Definition), oder ob auch der undefinierte Wert ψ und der undefinierte Zeitpunkt ϕ (vgl. S. 133) verwendet werden dürfen bzw. müssen. Durch die Verwendung einer Booleschen

Funktion ist es möglich, die Angabe in Abhängigkeit zu stellen, z. B. „entweder Zielwert oder Dauer müssen angegeben werden“ (Antivalenz).

Definition 5.2 (Animatorkonfiguration)
Gegeben sei eine Typmenge $\mathcal{T}$. Eine *Animatorkonfiguration* über $\mathcal{T}$ ist ein Tupel $cfg = (\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, args)$, wobei

- $\hat{x} \in \mathcal{V}_{\mathcal{P}}$ der *Startwert*,
- $\tilde{x} \in \mathcal{V}_{\mathcal{P}}^{\psi}$ der *angegebene Zielwert*, wobei gelten muss $vtype(\hat{x}) = vtype(\tilde{x})$, falls $\tilde{x} \neq \psi$,
- $\hat{t} \in \mathbb{T}$ der *Startzeitpunkt*,
- $\tilde{d} \in \mathbb{T}^{\phi}$ die *angegebene Dauer* und
- $args = (arg_1, arg_2, \ldots, arg_{argn}) \in {\mathcal{V}_{\mathcal{T}}}^{argn}$ das Tupel der *Argumentwerte* mit $argn \in \mathbb{N}_0$ als Anzahl der Werte ist.

$\mathcal{CFG}_{\mathcal{T}}$ bezeichne die Menge aller Animatorkonfigurationen über $\mathcal{T}$. △

Als Animatorkonfiguration wird der Datensatz bezeichnet, welcher für die Erzeugung einer Animatorinstanz notwendig ist. Die darin enthaltenen Werte können beeinflussen, wie der Animator einen Attributwert ändert. Zu den notwendigen Werten gehören in jedem Fall ein Startwert des zu verändernden Attributs und der Startzeitpunkt der Animation. Ob Zielwert oder Dauer angegeben werden müssen, spezifiziert der verwendete Animator bzw. dessen Signatur.

Die nun folgende Definition zeigt, in welcher Form eine Signatur die Menge der kompatiblen Konfigurationen festlegt:

Definition 5.3 (Kompatible Animatorkonfigurationen)
Gegeben sei eine Typmenge $\mathcal{T}$ und eine Animatorsignatur $sig \in \mathcal{SIG}_{\mathcal{T}}$. Die Menge der zu Signatur $sig = (\mathcal{ST}, \diamondsuit, (tp_1, tp_2, \ldots, tp_{tpn}))$ *kompatiblen Animatorkonfigurationen* $\mathcal{CFG}_{sig} \subseteq \mathcal{CFG}_{\mathcal{T}}$ ist wie folgt definiert:

$$
\begin{aligned}
\mathcal{CFG}_{sig} = \{\ (\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, args) \in \mathcal{CFG}_{\mathcal{T}} \mid \\
\hat{x} \in \mathcal{V}_{\mathcal{ST}} \quad & \wedge \\
\tilde{x} \in \mathcal{V}_{\mathcal{ST}}^{\psi} \quad & \wedge \\
(\ \tilde{x} = \psi\ ) \ \diamondsuit\ (\ \tilde{d} = \phi\ ) \quad & \wedge \\
args \in (\mathcal{V}_{\{tp_1\}} \times \mathcal{V}_{\{tp_2\}} \times \ldots \times \mathcal{V}_{\{tp_{tpn}\}}) \quad & \}\ .
\end{aligned}
$$

△

Schließlich lassen sich Animatoren wie folgt definieren:

Definition 5.4 (Animator)
Gegeben sei eine Typmenge $\mathcal{T}$. Ein *Animator* über $\mathcal{T}$ ist
ein Tupel $an = (aow, sig, af, ad)$ mit folgenden Komponenten:

- $aow \in \mathcal{M}$ ist der Animatoreigentümer.
- $sig \in \mathcal{SIG}_{\mathcal{T}}$ ist die Animatorsignatur.

- $af : \mathcal{CFG}_{sig} \times \mathbb{T} \to \mathcal{V}_{\mathcal{P}}$ ist die *Animatorfunktion*, die jeder Animatorkonfiguration $cfg = (\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, args) \in \mathcal{CFG}_{sig}$ und jedem absoluten Zeitpunkt $t \in \mathbb{T}$ einen *zeitabhängigen Wert* $x \in \mathcal{V}_{\mathcal{P}}$ zuordnet. Dabei muss gelten: $vtype(af(cfg, t)) = vtype(\hat{x})$.
- $ad : \mathcal{CFG}_{sig} \to \mathbb{T}^{\omega}$ ist die *Animationsdauerfunktion*, die einer Animatorkonfiguration $cfg \in \mathcal{CFG}_{sig}$ eine *Animationsdauer* $t \in \mathbb{T}^{\omega}$ zuordnet.

$\mathcal{AN}_{\mathcal{T}}$ bezeichne die Menge aller Animatoren über eine Typmenge $\mathcal{T}$. △

Diese Definition zeigt noch einmal die Elemente, die für eine Animatorspezifikation notwendig sind. Da Animatoren in AML für Medienkomponenten modelliert werden, haben Animatoren eine solche als Eigentümer. Die Signatur legt fest, wie und auf welche Attribute der Animator angewendet werden kann (vgl. Definition 5.1). Die Animatorfunktion bestimmt den Verlauf einer Animation. Die Animationsdauerfunktion wird verwendet, um die Dauer der Animation zu bestimmen, wobei auch unendlich lange Animationen möglich sind. An dieser Stelle muss allerdings angemerkt werden, dass die in der Animatorkonfiguration angegebene Dauer $\tilde{d}$ lediglich eine Vorgabe ist und nicht zwangsweise mit der durch *ad* berechneten Animationsdauer übereinstimmen muss. Analog muss der spezifizierte Zielwert $\tilde{x}$ nicht dem tatsächlichen Wert nach Ablauf der Animationsdauer entsprechen, und auch der angegebene Startwert $\hat{x}$ muss nicht zwangsweise von der Animatorfunktion für den Startzeitpunkt $\hat{t}$ zurückgegeben werden.[10]

Verwendung von Animatoren zur Laufzeit

Nach den Definitionen, welche die Spezifikation von Animatoren betreffen, folgen nun Definitionen und Erläuterungen, die verdeutlichen, wie Animatoren während der Laufzeit eingesetzt werden. Dies umfasst die Erzeugung von Animatorinstanzen, die Zuweisung von Animatorinstanzen zu Attributen, auch genannt *Aktivierung*, und die Art und Weise, wie Animatorinstanzen infolgedessen die Wertänderung der Attribute beeinflussen.

Definition 5.5 (Animatorinstanz)
Gegeben sei eine Typmenge $\mathcal{T}$. Eine *Animatorinstanz* über $\mathcal{T}$ ist ein Paar $ian = (an, cfg)$ mit $an = (aow, sig, af, ad) \in \mathcal{AN}_{\mathcal{T}}$ und $cfg \in \mathcal{CFG}_{sig}$.

$\mathcal{IAN}_{\mathcal{T}}$ bezeichne die Menge aller Animatorinstanzen über $\mathcal{T}$. △

Definition 5.6 (Inaktive und aktive Animatorinstanz)
Gegeben sei eine Typmenge $\mathcal{T}$. Ein aktueller Systemzustand *sys* über $\mathcal{T}$ enthält die partielle Funktion $app : \mathcal{OBJ}_{\mathcal{M}} \times \mathcal{A}_{\mathcal{M}} \rightharpoonup \mathcal{IAN}_{\mathcal{T}} \cup \{\overline{ian}\}$, die jedem Attribut *attr* von jeder Medienkomponenteninstanz $obj \in \mathcal{OBJ}_{\mathcal{M}}$, d. h., *app* ist definiert, falls $attr \in \mathcal{A}_{\{vtype(val(obj))\}}$, entweder die *inaktive Animatorinstanz* $\overline{ian} \notin \mathcal{IAN}_{\mathcal{T}}$ oder eine *aktive Animatorinstanz* $ian \in \mathcal{IAN}_{\mathcal{T}}$ zuordnet.

[10] Dadurch ist es bspw. unproblematisch, falls sich zur Laufzeit eine Dauer von 0 ergibt, aber unterschiedliche Start- und Zielwerte modelliert wurden. Derartige Fälle können daher individuell umgesetzt werden.

Falls eine aktive Animatorinstanz ian=$((aow,(\mathcal{ST},\Diamond,tp),af,ad),(\hat{x},\tilde{x},\hat{t},\tilde{d},args))$ zugeordnet wird, muss außerdem gelten $obj \in \mathcal{OBJ}_{\{aow\}}$, $vtype(\hat{x}) = atype(attr)$ und $atype(attr) \in \mathcal{ST}$. △

Die Erzeugung einer Animatorinstanz erfolgt zur Laufzeit, und sie ist somit Teil des Systemzustands. Sie kann für eine Medienkomponenteninstanz erstellt werden, deren Medienkomponente (bzw. Basismedienkomponente) einen entsprechenden Animator besitzt. Bei der Erzeugung müssen alle notwendigen Werte (Start-/Zielwert, Startzeitpunkt und Dauer) und weitere Argumentwerte gemäß Signatur festgelegt werden. Danach kann die Animatorinstanz aktiviert werden, indem sie einem Attribut einer Medienkomponenteninstanz zugewiesen wird, d. h., *app* des aktuellen Systemzustands *sys* wird entsprechend angepasst. Der Typ des Attributs muss sich dabei in der Menge der animierbaren Datentypen befinden und zum Startwert passen. Außerdem darf jedem Attribut maximal eine solche Animatorinstanz zugewiesen werden. Sobald keine Animatorinstanz mehr für das Attribut aktiv sein soll, kann die Zuweisung durch die inaktive Animatorinstanz wieder aufgehoben werden, auch genannt *Deaktivierung*.

Hilfsdefinitionen
Bei gegebener Typmenge $\mathcal{T}$ seien für Animatorinstanzen der *tatsächliche Endzeitpunkt* und der *tatsächliche Zielwert* durch folgende Funktionen gegeben:

- $\check{t} : \mathcal{IAN}_{\mathcal{T}} \to \mathbb{T}^{\omega}$ ordnet jeder Animatorinstanz $ian \in \mathcal{IAN}_{\mathcal{T}}$, wobei $ian = ((aow, sig, af, ad), cfg)$ mit $cfg = (\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, args)$, den *tatsächlichen Endzeitpunkt* zu:

$$\check{t} : ian \mapsto \hat{t} + ad(cfg) \quad .$$

- $\check{x} : \mathcal{IAN}_{\mathcal{T}} \to \mathcal{V}_{\mathcal{P}}^{\psi}$ ordnet jeder Animatorinstanz $ian \in \mathcal{IAN}_{\mathcal{T}}$, wobei $ian = ((aow, sig, af, ad), cfg)$, den *tatsächlichen Zielwert* zu:

$$\check{x} : ian \mapsto \begin{cases} \psi & , \text{falls } \check{t}(ian) = \omega \\ af(cfg, \check{t}(ian)) & , \text{falls } \check{t}(ian) \neq \omega \end{cases} \quad . \qquad \triangle$$

Es ist jederzeit möglich, den statischen Wert eines Attributs abzufragen (vgl. *saval* auf S. 133), was im Normalfall einem Speicherzugriff auf den für das Attribut gespeicherten Wert entspricht. Unter Einsatz von AML-Animatoren ist dieser Wert aber nicht mehr zwangsweise der aktuelle Wert eines animierten Attributs. Vielmehr sollte der zeitabhängige Wert des Attributs abgefragt werden, wobei zusätzlich der Zeitpunkt der Abfrage erforderlich ist. Nur falls dem Attribut die inaktive Animatorinstanz zugewiesen ist, entspricht der zeitabhängige Wert immer dem statischen Wert. Abb. 5.11, S. 144, zeigt, welchen Einfluss die Verwendung der beiden Funktionen bei der Visualisierung haben kann. Folgende Definition zeigt formal, wie der zeitabhängige Wert abgefragt wird:

Definition 5.7 (Zeitabhängiger Attributwert)
Gegeben sei der aktuelle Systemzustand $(\mathcal{OBJ}, val, app, \ldots) \in \mathcal{SYS}_{\mathcal{T}}$. Die partielle Funktion $taval : \mathcal{OBJ}_{\mathcal{M}} \times \mathcal{A}_{\mathcal{M}} \times \mathbb{T} \rightharpoonup \mathcal{V}_{\mathcal{P}}$ ordnet jedem Attribut $attr$ von jeder Medienkomponenteninstanz $obj \in \mathcal{OBJ}_{\mathcal{M}}$, d. h., $taval$ ist definiert, falls $attr \in \mathcal{A}_{\{vtype(val(obj))\}}$, in Abhängigkeit von der absoluten Zeit $t \in \mathbb{T}$ den *zeitabhängigen Attributwert* $x \in \mathcal{V}_{\mathcal{P}}$ zu. Sie ist gegeben durch

$$taval : (obj, attr, t) \mapsto \begin{cases} saval(val(obj), attr) \\ \text{, falls } app(obj, attr) = \overline{ian} \\ af(cfg, t) \\ \text{, falls } app(obj, attr) = ((aow, sig, af, ad), cfg) \end{cases} .\triangle$$

Dabei gilt $vtype(x) = atype(attr)$, d. h., $taval$ liefert immer einen Wert des richtigen Typs, da nach Definition gilt: $vtype(saval(val(obj), attr)) = atype(attr)$ und $vtype(af(cfg, t)) = vtype(af((\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, args), t)) = vtype(\hat{x}) = atype(attr)$.

Schließlich definiert AML, dass eine aktive Animatorinstanz bei Erreichen des tatsächlichen Endzeitpunkts deaktiviert wird, d. h., die Zuordnung der Animatorinstanz zum Attribut wird dann aufgehoben. Dies kann auch bereits zu einem früheren Zeitpunkt passieren, z. B. falls ein entsprechender Animationszustand (siehe Abschnitt 5.4.4, S. 151) verlassen wird.

Wird eine Animatorinstanz deaktiviert, so muss der statische Wert des Attributs aktualisiert werden. Das Laufzeitsystem muss hierfür den zeitabhängigen Wert unmittelbar vor der Deaktivierung auslesen. Dieser wird zum neuen statischen Wert. Anders formuliert: wird infolge einer Änderung des aktuellen Systemzustands zum Zeitpunkt $t \in \mathbb{T}$, geschrieben $sys \overset{t}{\Rightarrow} sys'$ bzw. $(\mathcal{OBJ}, val, app, \ldots) \overset{t}{\Rightarrow} (\mathcal{OBJ}', val', app', \ldots)$, die Animatorinstanz für Attribut $attr \in \mathcal{A}_{\mathcal{M}}$ einer Medienkomponenteninstanz $obj \in \mathcal{OBJ}_{\mathcal{M}}$ deaktiviert, d. h. $obj \in \mathcal{OBJ}'_{\mathcal{M}}$, $app(obj, attr) \neq \overline{ian}$ und $app'(obj, attr) = \overline{ian}$, so muss gelten $taval(obj, attr, t) = saval(val'(obj), attr)$.

An dieser Stelle sei erwähnt, dass bei regulärer Verwendung eines Attributs in AML-Modellen, z. B. innerhalb von Bedingungen, Bezug auf seinen zeitabhängigen Wert genommen wird. So werden auch durch Bedingungssensoren (vorgestellt in Abschnitt 5.4.5, S. 161) zeitabhängige Werte von Attributen überwacht. Zur Abfrage muss das Laufzeitsystem die aktuelle Systemzeit zur Verfügung stellen. Wenn einem Attribut ein neuer Wert zugewiesen wird, gilt außerdem, dass der statische Wert des Attributs verändert wird.

Metamodell Bei der Erstellung eines AML-Modells müssen verwendete Animatoren angegeben werden. Sie gehören dabei zu einer Medienkomponente (vgl. Abb. 5.5, S. 135) und entsprechende Animatoren können nur von dieser Medienkomponente oder abgeleiteten Medienkomponenten genutzt werden.

Im AML-Modell werden Animatoren allerdings nicht vollständig spezifiziert. Es werden lediglich Teile der Animatorensignatur aus Definition 5.1, S. 144, angegeben. Modelliert werden neben dem Namen des Animators nur noch dessen Parameter. Animatoren sind daher von *uml::BehavioralFeature* abgeleitet, wovon bspw. auch UML-Operationen abgeleitet werden. Im Gegensatz zu Operationen

besitzen Animatoren aber keinen eigenen Typ oder Bedingungen. Interessant ist dagegen die Möglichkeit, Animatoren wie Operationen abstrakt zu deklarieren oder neu zu definieren. Unterklassen können den Animator dann unterschiedlich implementieren.

Die Spezifikation der animierbaren Datentypen und der restlichen Signatur ist im AML-Modell nicht möglich. Diese Einschränkung liegt darin begründet, dass es problematisch wäre, solche Typmengen oder Details in einem Diagramm darzustellen. Auch die Modellierung der Animatorfunktionen und Animationsdauerfunktion würde sich schwierig gestalten. Daher wurde von einer vollständigen Animatorspezifikation im Modell abgesehen, ähnlich wie bei Operationen. Ob es Spezifikationsmöglichkeiten mittels OCL o. ä. Mechanismen gibt, wurde allerdings nicht untersucht. △

Notation Aufgrund ihrer Eigenschaften ist die Notation von Animatoren der Notation von UML-Operationen sehr ähnlich. Animatoren werden daher in das gleiche Compartment eingetragen wie die Operationen. Nach dem Namen des Animators folgt die Liste der Parameter. Es wird allerdings kein Rückgabetyp spezifiziert. Zur visuellen Unterscheidung werden Animatoren außerdem mit *«Animator»* gekennzeichnet. △

In den folgenden Beispielen und Definitionen wird angenommen, dass in jedem AML-Modell folgende UML-Typen vorhanden sind: *Integer*, *Real*, *Boolean* und *IAnimator*. Die Typen *Integer* und *Boolean* bezeichnen hierbei die primitiven UML-Standarddatentypen für ganze Zahlen bzw. boolesche Werte. *Real* für reelle Zahlen ist zwar kein Standarddatentyp, wird in der Spezifikation allerdings im Rahmen von Wertliteralen beschrieben [UML11]. In einer Programmiersprache wie Java werden diese Datentypen bspw. durch *int*, *boolean* und *double* repräsentiert. *IAnimator* sei eine Schnittstelle der in diesem Abschnitt definierten Animatorinstanzen (siehe Listing 7.1, S. 220). Anstelle der Typnamen *Integer*, *Boolean*, *Real* und *IAnimator* werden $\mathbb{Z}$, $\mathbb{B}$, $\mathbb{R}$ und $\mathcal{IAN}_{\mathcal{T}}$ symbolisch sowohl für diese Typen selbst als auch für die Menge der Werte der entsprechenden Typen verwendet. Die Menge $\mathbb{T}^{\omega}$ wird ebenfalls benötigt. Entsprechende Zeitwerte werden im Modell mit Werten vom Typ *Real* hinterlegt.

Als Eigentümer der Animatoren wird jeweils *MediaComponent2D* angegeben. Dies ist eine Medienkomponente, die im AML-Basisframework existiert und damit in allen AML-Modellen dieser Arbeit vorhanden ist (siehe Abschnitt 5.5, S. 175).

Beispiel 5.5 (Animator für lineare Interpolation)
Die folgenden Komponenten definieren den Animator für lineare Interpolation $an_{li} = (aow_{li}, sig_{li}, af_{li}, ad_{li})$ mit $sig_{li} = (\mathcal{ST}_{li}, \diamondsuit_{li}, tp_{li})$:

- $aow_{li} = MediaComponent2D$,
- $\mathcal{ST}_{li} = \{\mathbb{R}\}$,
- $\diamondsuit_{li} : (spec_{\tilde{x}}, spec_{\tilde{d}}) \mapsto spec_{\tilde{x}} \wedge spec_{\tilde{d}}$,

d. h., sowohl Zielwert als auch Dauer müssen bei der Erzeugung der Animatorinstanz angegeben werden,

- $tp_{li} = ()$ für die Argumente $args = ()$,
  d. h., es werden keine Argumente unterstützt,
- $af_{li} : ((\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, ()), t) \mapsto \begin{cases} \hat{x} + \frac{\tilde{x}-\hat{x}}{\tilde{d}}(t - \hat{t}) & , \text{falls } \tilde{d} \neq 0 \\ \tilde{x} & , \text{falls } \tilde{d} = 0 \end{cases}$ und
- $ad_{li} : (\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, ()) \mapsto \tilde{d}$.

Abb. 5.22, S. 176, zeigt, wie dieser Animator in AML-Diagrammen notiert und modelliert werden kann. Außer der Angabe des Namens *interpolateLinear* und der Animatorparameter (in diesem Fall keine) können in AML keine weiteren Details modelliert werden.

Verwendung findet der Animator bspw., um die Zeiger der analogen Uhr in Abb. 5.2, S. 125, zu drehen. Die Änderung der Zeigerwinkel durch die lineare Interpolation zwischen 0° und 360° erfolgt dabei gemäß Definition stufenlos, d. h., auch die Zeiger werden stufenlos gedreht (kontinuierlich). Soll diese Animation anders dargestellt werden, z. B. für eine analoge Uhr, deren Zeiger stufenweise von Punkt zu Punkt springt (diskret), genügt es, einen anderen, leicht modifizierten Animator zu spezifizieren und für den Winkel des Zeigers zu verwenden. Ein geeigneter Animator kann mit folgender Animatorfunktion af_{lid} spezifiziert werden, die auf der linearen Interpolation basiert. Im Gegensatz zu dieser werden in Abhängigkeit eines Animatorparameters *step* aber nur diskrete Werte berechnet:

$$af_{lid}((\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, (step)), t) = \lfloor \frac{af_{li}((\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, ()), t)}{step} \rfloor \cdot step \ .$$

Im Falle des Minutenzeigers müsste für *step* der Wert $\frac{360}{60}$ gewählt werden, für den Stundenzeiger $\frac{360}{12}$. △

Beispiel 5.6 (Animator für gleichförmige Bewegung)
Die Komponenten des Animators für gleichförmige Bewegung (ohne Beschleunigung) $an_{lm} = (aow_{lm}, sig_{lm}, af_{lm}, ad_{lm})$ mit $sig_{lm} = (\mathcal{ST}_{lm}, \diamondsuit_{lm}, tp_{lm})$ sind definiert durch

- $aow_{lm} = MediaComponent2D$,
- $\mathcal{ST}_{lm} = \{\mathbb{R}\}$,
- $\diamondsuit_{lm} : (spec_{\tilde{x}}, spec_{\tilde{d}}) \mapsto spec_{\tilde{x}} \neq spec_{\tilde{d}}$,
  d. h., entweder Zielwert oder Dauer (Antivalenz) muss angegeben werden,
- $tp_{lm} = (\mathbb{R})$ für die Argumente $args = (ve)$,
  wobei das Argument $ve \in \mathbb{R}$ die (Änderungs-)Geschwindigkeit für den zeitabhängigen Wert des animierten Attributs angibt,

- $af_{lm} : ((\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, (ve)), t) \mapsto ve \cdot (t - \hat{t}) + \hat{x}$ und
- $$ad_{lm} : (\hat{x}, \tilde{x}, \hat{t}, \tilde{d}, (ve)) \mapsto \begin{cases} \tilde{d} & , \text{falls } \tilde{d} \neq \phi \\ 0 & , \text{falls } \tilde{d} = \phi \wedge ve = 0 \wedge \hat{x} = \tilde{x} \\ \omega & , \text{falls } \tilde{d} = \phi \wedge ve = 0 \wedge \hat{x} \neq \tilde{x} \\ \omega & , \text{falls } \tilde{d} = \phi \wedge ve \neq 0 \wedge \frac{\tilde{x}-\hat{x}}{ve} < 0 \\ \frac{\tilde{x}-\hat{x}}{ve} & , \text{falls } \tilde{d} = \phi \wedge ve \neq 0 \wedge \frac{\tilde{x}-\hat{x}}{ve} \geq 0 \end{cases} .$$

Dieser Animator wird ebenfalls in Abb. 5.22, S. 176, unter dem Namen *moveLinear* dargestellt. Die Geschwindigkeit *velocity* ist Teil der Parameterliste. △

5.4.4 Animationszustände und Animationsanweisungen

Wie im vorherigen Abschnitt beschrieben wird, kann eine Animatorinstanz erzeugt und aktiviert werden, d. h., dem Attribut einer Medienkomponenteninstanz zugewiesen werden, um den zeitlichen Verlauf dessen Werts zu bestimmen. Bisher wurde allerdings nicht beschrieben, wann eine solche Erzeugung und Aktivierung erfolgt und wie diese modelliert werden kann. AML kennt hierfür eine spezielle Art von Zustand, den sogenannten *Animationszustand*. Diese können wie gewöhnliche UML-Zustände für die Verhaltensmodellierung von Medienkomponenten innerhalb eines AML-Zustandsautomaten eingesetzt werden.

An dieser Stelle sei darauf hingewiesen, dass die dem Zustandsautomaten zugewiesene Medienkomponente bzw. Klasse in der restlichen Arbeit auch *Zustandsautomateneigentümer* genannt wird, teilweise auch dann, wenn damit eigentlich die Medienkomponenteninstanz bzw. Instanz gemeint ist. In diesem Zusammenhang wird auch der Begriff „*Zustandsautomatenkontext*“ verwendet. Merkmale (z. B. Attribute oder Methoden) im Zustandsautomatenkontext sind die Merkmale, die dem Zustandsautomateneigentümer oder einer Basisklasse bzw. Basismedienkomponente angehören. Im Falle einer Medienkomponente werden dabei auch die jeweiligen Merkmale der Unterkomponenten einbezogen.

Innerhalb von Animationszuständen können *Animationsanweisungen* modelliert werden, wobei mindestens eine solche Anweisung für einen Animationszustand existieren muss. Eine Animationsanweisung soll beschreiben, wie eine Animatorinstanz bei Betreten des Animationszustands erzeugt und einem Attribut im Zustandsautomatenkontext zugewiesen wird. Nach der Zuweisung ist die Animatorinstanz aktiv und das Attribut wird animiert, bis der tatsächliche Endzeitpunkt der Animatorinstanz erreicht oder die Animationsanweisung *abgebrochen* wird.

Animationsanweisungen repräsentieren eine spezielle Art von „do“-Aktivität des Animationszustands. Dies hat bspw. zur Folge, dass der Animationszustand während der Animation durch Ereignisse und passende Trigger (z. B. Sensortrigger, siehe Abschnitt 5.4.5, S. 159, und insbesondere Beispiel 5.10, S. 162) verlassen werden kann. In diesem Fall werden alle Animationsanweisungen des Animationszustands abgebrochen, d. h., die jeweiligen Animatorinstanzen werden deaktiviert. Die Attribute erhalten dabei ihre zeitabhängigen Werte zum Zeitpunkt des Abbruchs als neue statische Werte (vgl. Abschnitt 5.4.3, S. 148).

Falls keine Unterbrechung stattfindet und alle Animationsanweisungen ihr Ende erreichen, d. h., der letzte tatsächliche Endzeitpunkt der Animatorinstanzen erreicht wird, wird ein „completion“-Ereignis für den Animationszustand ausgelöst. Entsprechende Transitionen mit Abschlussereignis-Trigger können zu diesem Zeitpunkt schalten. Zusätzliche Voraussetzung ist, dass keine Unterzustände aktiv sind, wobei finale Zustände nicht einbezogen werden. Umgekehrt kann ein „completion“-Ereignis frühestens dann ausgelöst werden, wenn auch die Animationsanweisungen ihr Ende erreicht haben.

Außerdem ist es möglich, im Rahmen einer Animationsanweisung die Aktivierung mehrerer Animatorinstanzen sequentiell zu verketten. Diese Verkettung kann durch einen Animator, genannt *Sequenzanimator*, beschrieben werden, der bestimmte Animatorinstanzen nacheinander anwendet. Diese Animatorinstanzen müssen hierfür als Parameter an den Sequenzanimator übergeben werden:

Definition 5.8 (Sequenzanimator für Animationsanweisungen)
Der *Sequenzanimator* $an_{sq}=(aow_{sq}, sig_{sq}, af_{sq}, ad_{sq})$ mit $sig_{sq}=(\mathcal{ST}_{sq}, \Diamond_{sq}, tp_{sq})$ ist mit folgenden Komponenten definiert:

- $aow_{lm} = MediaComponent2D$,
- $\Diamond_{sq} : (spec_{\tilde{x}}, spec_{\tilde{d}}) \mapsto (spec_{\tilde{x}} = falsch) \wedge (spec_{\tilde{d}} = falsch)$,
  d. h., weder Zielwert noch Dauer dürfen angegeben werden, da diese durch die Animatorinstanzen der Sequenz festgelegt werden,
- $tp_{sq} = (\mathcal{IAN}_{\mathcal{T}}, \mathcal{IAN}_{\mathcal{T}}, \ldots, \mathcal{IAN}_{\mathcal{T}})$ für die Argumente
  $$args = (ian_1, ian_2, \ldots, ian_{argn}) \ ,$$
  wobei die Argumente die Animatorinstanzen
  $$ian_i = ((aow_i, (\mathcal{ST}_i, \Diamond_i, tp_i), af_i, ad_i), (\hat{x}_i, \tilde{x}_i, \hat{t}_i, \tilde{d}i, args_i)) \in \mathcal{IAN}_{\mathcal{T}}$$
  mit $i \in \mathbb{N}^+ \wedge i \leq argn$ angeben, die sequentiell durch diesen Animator verwendet werden sollen.
- $af_{sq} : (cfg, t) \mapsto \begin{cases} af_1(cfg_1, t) & , \text{falls } t < \hat{t}_1 \\ af_i(cfg_i, t) & , \text{falls } \hat{t}_i \leq t < \hat{t}_{i+1} \text{ für ein } i \in \mathbb{N}^+ \wedge i < argn \\ af_{argn}(cfg_{argn}, t) & , \text{falls } \hat{t}_{argn} \leq t \end{cases}$ und
- $ad_{sq} : cfg \mapsto \sum_{i=1}^{argn} ad_i(cfg_i)$.

Für eine korrekte Spezifikation und einen stetigen Übergang zwischen den Animatorinstanzen gelten zudem die Einschränkungen: $\hat{t} = \hat{t}_1$, $\hat{x} = \hat{x}_1$ und $\forall i \in \mathbb{N}^+ \wedge i < argn : (\check{t}(ian_i) \neq \omega) \rightarrow (\hat{t}_{i+1} = \check{t}(ian_i) \wedge \hat{x}_{i+1} = \check{x}(ian_i))$. Auch die Menge der animierbaren Datentypen ist eingeschränkt auf:

$$\mathcal{ST}_{sq} = \bigcap_{i \in \mathbb{N}^+ \wedge i < argn} \mathcal{ST}_i \quad . \qquad \triangle$$

	Falls eine *Umkehr* verwendet wird, so wird die Animation nach ihrem Ende noch einmal rückwärts abgespielt. Die Modifikation wird allerdings ignoriert, falls die Animationsdauer der Animatorinstanz unendlich ist.
	Bei Verwendung von *Wiederholung*, wird die Animation nach ihrem Ende (und nach einer möglichen Umkehr) wiederholt. $eval(x) \in \mathbb{R}$ legt dabei die Anzahl der Wiederholungen fest. Ein Wert von 3,0 würde z. B. bedeuten, dass die Animation drei mal abgespielt wird. Bei einem Wert von 0,5 läuft die Animation nur bis zur Hälfte ab usw. Unendlich viele Wiederholungen können ebenfalls mit $eval(x) < 0$ oder dem Symbol ∞ spezifiziert werden.
	Bei Verwendung einer *Fortsetzung*, wird die Animation nach der eigentlichen Animationsdauer der Animatorinstanz nicht gestoppt, d. h., die Animation wird unendlich lange fortgesetzt.
	Falls eine *Geschwindigkeit* verwendet wird, so wird die Abspielgeschwindigkeit der gesamten Animation durch den Faktor $eval(x) \in \mathbb{R}$ modifiziert. Ein Wert von 2,0 bedeutet bspw., dass die Geschwindigkeit verdoppelt und die Animationsdauer der Animatorinstanz halbiert wird.
	Durch Verwendung einer *maximalen Dauer* wird die gesamte Animationsdauer der Animatorinstanz auf $eval(x) \in \mathbb{T}$ begrenzt.

Tabelle 5.3: Modifikationen für Animationsanweisungen

Animatormodifikation

Neben solchen Animatorsequenzen können Animationsanweisungen sogenannte *Animatormodifikationen* einsetzen, um den verwendeten Animator vor dem Erzeugen einer Animatorinstanz durch typische Modifikationen anzupassen. Dadurch kann der Animationsverlauf zusätzlich verändert werden. Beispiele für solche Modifikationen sind die Änderung der Animationsgeschwindigkeit, die Verwendung von Animationsschleifen etc.

Die im Folgenden beschriebenen Animatormodifikationen sind fester Bestandteil von AML und können daher innerhalb von Animationsanweisungen genutzt werden. Die Animatormodifikationen werden in Tab. 5.3 zusammen mit ihrer Notation innerhalb der AML-Diagramme zunächst informell beschrieben. Anstelle des x in den Grafiken kann entweder eine Konstante oder ein Ausdruck (eine einfache mathematische Funktionen oder ein wirkungsfreier Methodenaufruf im Zustandsautomatenkontext) notiert werden. Die Auswertung von x während der Systemlaufzeit wird mit *eval(x)* bezeichnet. Der resultierende Wert wird anschließend für den Modifikationsparameter eingesetzt.

Formal sind die Animatormodifikation und die konkret verfügbaren Animatormodifikationen wie folgt definiert. Es wird erneut davon ausgegangen, dass bestimmte Typen immer Teil der gegebenen Typmenge sind: $\mathbb{B}, \mathbb{R}, \mathbb{T}^\omega \in \mathcal{T}$ (vgl. Abschnitt 5.4.3, S. 149).

Definition 5.9 (Animatormodifikation)
Gegeben sei eine Typmenge $\mathcal{T}$ und eine Folge von Modifikationsparametertypen $(tp_1, \ldots, tp_{mn}) \in \mathcal{T}^{mn}$ mit $mn \in \mathbb{N}_0$. Eine *Animatormodifikation* über $\mathcal{T}$ und diese Folge ist eine Funktion $m : \mathcal{AN}_\mathcal{T} \times (\mathcal{V}_{tp_1} \times \cdots \times \mathcal{V}_{tp_{mn}}) \to \mathcal{AN}_\mathcal{T}$, die jedem Animator $an \in \mathcal{AN}_\mathcal{T}$ in Abhängigkeit der *Modifikationsargumente* $p \in \{p_1, \ldots, p_{mn} \mid p_1 \in \mathcal{V}_{tp_1} \wedge \ldots \wedge p_{mn} \in \mathcal{V}_{tp_{mn}}\}$ einen modifizierten Animator $an' \in \mathcal{AN}_\mathcal{T}$ zuordnet. △

Definition 5.10 (Animatormodifikation für „Umkehr“)
Die Animatormodifikation für die Modifikation „Umkehr“ ist die Funktion $m_T : \mathcal{AN}_\mathcal{T} \times \mathbb{B} \to \mathcal{AN}_\mathcal{T}, ((aow, sig, af, ad), tu) \mapsto (aow, sig, af'_T, ad'_T)$, wobei

$$af'_T : (cfg, t) \mapsto \begin{cases} af(cfg, 2 \cdot \check{t}(ian) - t) & \text{, falls } t > \check{t}(ian) \wedge tu = wahr \\ af(cfg, t) & \text{, sonst} \end{cases} \quad \text{mit}$$

$$ian = ((aow, sig, af, ad), cfg) \quad \text{und}$$

$$ad'_T : cfg \mapsto \begin{cases} ad(cfg) & \text{, falls } tu = falsch \\ 2 \cdot ad(cfg) & \text{, falls } tu = wahr \end{cases} . \quad \triangle$$

Definition 5.11 (Animatormodifikation für „Wiederholung“)
Die Animatormodifikation für die Modifikation „Wiederholung“ ist die Funktion $m_R : \mathcal{AN}_\mathcal{T} \times \mathbb{R} \to \mathcal{AN}_\mathcal{T}, ((aow, sig, af, ad), re) \mapsto (aow, sig, af'_R, ad'_R)$, wobei

$$af'_R : (cfg, t) \mapsto \begin{cases} af(cfg, t - min\{\lfloor \frac{t-\hat{t}}{ad(cfg)} \rfloor, re\} \cdot ad(cfg)) \\ \qquad \text{, falls } ad(cfg) \neq 0 \wedge re \geq 0 \\ af(cfg, t - \lfloor \frac{t-\hat{t}}{ad(cfg)} \rfloor \cdot ad(cfg)) \\ \qquad \text{, falls } ad(cfg) \neq 0 \wedge re < 0 \\ af(cfg, t) \\ \qquad \text{, falls } ad(cfg) = 0 \end{cases} \quad \text{und}$$

$$ad'_R : cfg \mapsto \begin{cases} re \cdot ad(cfg) & \text{, falls } ad(cfg) \neq 0 \wedge re \geq 0 \\ \omega & \text{, falls } ad(cfg) \neq 0 \wedge re < 0 \\ 0 & \text{, falls } ad(cfg) = 0 \end{cases} . \quad \triangle$$

Definition 5.12 (Animatormodifikation für „Fortsetzung“)
Die Animatormodifikation für die Modifikation „Fortsetzung“ ist die Funktion $m_C : \mathcal{AN}_\mathcal{T} \times \mathbb{B} \to \mathcal{AN}_\mathcal{T}, ((aow, sig, af, ad), co) \mapsto (aow, sig, af, ad'_C)$, wobei

$$ad'_C : cfg \mapsto \begin{cases} ad(cfg) & \text{, falls } co = falsch \\ \omega & \text{, falls } co = wahr \end{cases} . \quad \triangle$$

Definition 5.13 (Animatormodifikation für „Geschwindigkeit“)
Die Animatormodifikation für die Modifikation „Geschwindigkeit“ ist die Funktion $m_V : \mathcal{AN}_\mathcal{T} \times \mathbb{R} \to \mathcal{AN}_\mathcal{T}, ((aow, sig, af, ad), ve) \mapsto (aow, sig, af'_V, ad'_V)$, wobei

$$af'_V : (cfg, t) \mapsto \begin{cases} af(cfg, \hat{t} + ve \cdot (t - \hat{t})) & \text{, falls } ve > 0 \\ af(cfg, \hat{t}) & \text{, falls } ve \leq 0 \end{cases} \quad \text{und}$$

$$ad'_V : cfg \mapsto \begin{cases} \frac{ad(cfg)}{ve} & \text{, falls } ve > 0 \\ \omega & \text{, falls } ve \leq 0 \end{cases} \quad . \qquad \triangle$$

Definition 5.14 (Animatormodifikation für „Maximale Dauer“)
Die Animatormodifikation für die Modifikation „Maximale Dauer“ ist die Funktion $m_M : \mathcal{AN}_\mathcal{T} \times \mathbb{T}^\omega \to \mathcal{AN}_\mathcal{T}, ((aow, sig, af, ad), md) \mapsto (aow, sig, af, ad'_M)$, wobei

$$ad'_M : cfg \mapsto min\{ad(cfg), md\} \; . \qquad \triangle$$

Da mehrere Animatormodifikationen gleichzeitig genutzt werden können und die Reihenfolge der Anwendung entscheidend ist, werden Animatormodifikationen nur in Form der folgenden Animatormodifikation verwendet, welche alle bisherigen Animatormodifikationen miteinander verknüpft.

Definition 5.15 (Animatormodifikation für Animationsanweisungen)
Die Animatormodifikation für AML-Animationsanweisungen ist die Funktion

$$m_A : \mathcal{AN}_\mathcal{T} \times \mathbb{B} \times \mathbb{R} \times \mathbb{B} \times \mathbb{R} \times \mathbb{T}^\omega \to \mathcal{AN}_\mathcal{T}$$
$$(an, tu, re, co, ve, md) \mapsto m_M(m_V(m_C(m_R(m_T(an, tu), re), co), ve), md) \; . \; \triangle$$

Metamodell Abb. 5.12 zeigt die Metaklasse *AnimationState* als Ableitung von *uml::State*. Animationszustände sind somit eine Erweiterung von UML-Zuständen und erben damit ihre Eigenschaften und Möglichkeiten. Zusätzlich können Animationsanweisungen hinzugefügt werden, wobei mindestens eine Animationsanweisung für einen Animationszustand notwendig ist.

Animationsanweisungen werden durch die Metaklasse *AnimationInstruction* repräsentiert. Diese ist von *FeatureSelector* und somit *uml::Feature* abgeleitet, d. h., es muss ein Attribut (*feature*) gewählt werden, das von der Animationsanweisung betroffen sein soll.

Jede *AnimationInstruction* enthält außerdem eine Liste von *AnimationInstructionStep*-Elementen. Ein solches Element repräsentiert einen Abschnitt der Animation, wobei jedem Abschnitt ein Animator und eine Teilkonfiguration dessen zugewiesen werden muss, nämlich angegebener Zielwert *targetValue*, angegebene Dauer *duration* und die Argumentwerte *arguments* für die Animatorparameter. Es wird jeweils der Wert *null* für den undefinierten Wert ψ oder den undefinierten Zeitpunkt ϕ verwendet.

Diese Teilspezifikation ermöglicht die Erzeugung einer Sequenzanimatorinstanz gemäß Definition 5.8. Die jeweiligen Startwerte und Startzeitpunkte müssen

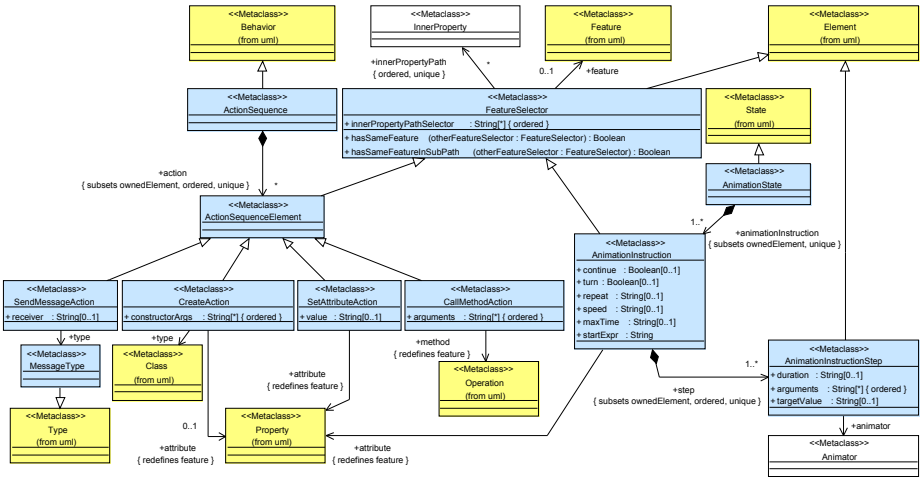


Abbildung 5.12: AML-Metamodell: Animationszustände

nicht festgelegt werden, da diese durch die Vorgaben und die für den Sequenzanimator definierten Einschränkungen berechnet werden können. Der Startzeitpunkt der ersten Animatorinstanz entspricht immer dem Zeitpunkt des Betretens des zugehörigen Animationszustands, d. h., er wird zur Laufzeit automatisch gesetzt und muss nicht modelliert werden. Lediglich der Startwert für die erste Animatorinstanz muss modelliert werden. Dieser wird innerhalb der Anweisung selbst (*startExpr*) eingetragen.

Die Metaklasse *AnimationInstruction* unterstützt außerdem die Verwendung der Standard-Animatormodifikationen und erlaubt die Angabe der Werte für die Modifikationsparameter (vgl. Definition 5.15): *turn* (Umkehr), *repeat* (Wiederholung), *continue* (Fortsetzung), *speed* (Geschwindigkeit) und *maxTime* (maximale Dauer).

Die beschriebenen Werte für die Animatorkonfiguration (*targetValue*, *duration*, *startExpr*, *arguments*) oder Animatormodifikation – sofern nicht vom Typ Boolean – müssen im Modell nicht zwangsweise als Konstanten angegeben werden. Es kann ebenfalls ein Ausdruck (eine einfache mathematische Funktionen oder ein wirkungsfreier Methodenaufruf im Zustandsautomatenkontext) modelliert werden, der dann zur Systemlaufzeit ausgewertet und zur Erzeugung der jeweiligen Animatorinstanz verwendet wird. △

Zusicherungen Animationszustände und damit auch Animationsanweisungen können nur in Zustandsautomaten verwendet werden, die das Verhalten von Medienkomponenten spezifizieren (siehe Listing B.2/35). Zusätzlich darf eine Animationsanweisung nur Animatoren einsetzen, die auch Elemente des Zustandsautomateneigentümers oder einer Basismedienkomponente sind (siehe Listing B.2/52). Das für die Animationsanweisung gewählte Attribut muss sich

im Zustandsautomatenkontext befinden, d. h., *FeatureSelector* erlaubt auch die Selektion von Attributen einer Unterkomponente.

Durch Definition 5.6, S. 146, wird verdeutlicht, dass jedem Attribut einer Medienkomponenteninstanz maximal eine aktive Animatorinstanz zugeordnet werden kann. Daher darf zu jedem Zeitpunkt maximal eine Animationsanweisung ein bestimmtes Attribut beeinflussen. Dies wird bereits im AML-Modell sichergestellt. Die Überprüfung, dass ein Attribut nicht durch mehrere Animationsanweisungen innerhalb eines Zustands beeinflusst wird, ist trivial (siehe Listing B.2/40). Allerdings müssen die Animationsanweisungen von möglichen Parallelzuständen ebenfalls in Betracht gezogen werden. Für AML-Modelle gilt daher folgende Restriktion: zwei Animationsanweisungen, die dasselbe Attribut betreffen, dürfen nicht in Animationszuständen eingesetzt werden, die möglicherweise gleichzeitig aktiv sind.[11] Diese Zusicherung ist für das Metamodell entsprechend formuliert. △

Notation Animationsanweisungen werden im unteren Teil ihres Animationszustands notiert, der durch eine gestrichelte Linie vom oberen Teil getrennt wird. Zusätzlich weist die Überschrift „Animation Instructions“ auf die Auflistung der Animationsanweisungen hin.

Eine einzelne Animationsanweisung beginnt mit dem Namen des Attributs, dem ein Animator zugewiesen werden soll. Für Attribute einer Unterkomponente wird die aus der OCL bekannte, typische Schreibweise verwendet (vgl. Abb. 5.2, S. 125). Es folgen ein Doppelpunkt und der Startwert. Danach wird für jeden Schritt ein Pfeil mit anschließendem Zielwert notiert. Dabei steht auch das Schlüsselwort *res* („result“) zur Verfügung, falls kein Zielwert angegeben werden soll (vgl. Abb. 5.14, S. 158). Auf dem Pfeil wird der Name des zu verwendenden Animators geschrieben, wobei die Argumentwerte in Klammern hinzugefügt werden. Wird kein Animator explizit angegeben, wird der Standardanimator verwendet, der meist dem Animator für lineare Interpolation (vgl. Beispiel 5.5, S. 149) entspricht. Auch die Dauer kann auf dem Pfeil (evtl. nach dem potentiellen Animator) angegeben werden. Sie wird in einem „{ }“-Block eingeschlossen. Unterstützt werden dabei die Zeiteinheiten aus Tab. 5.1, S. 130. Neben Konstanten können für die einzelnen Werte auch einfache mathematische Funktionen oder wirkungsfreie Methodenaufrufe im Zustandsautomatenkontext verwendet werden.

Animatormodifikationen werden ganz rechts durch ihre jeweiligen Symbole (vgl. Tab. 5.3, S. 153) festgehalten. Bei Fehlen eines Symbols kommt die entsprechende Modifikation nicht zum Einsatz bzw. es wird ein Standardwert für die Animatormodifikation angenommen, so dass die Animatormodifikation einen Animator nicht verändert. △

[11] Eine solche Modellierung ist also auch dann unzulässig, falls es aufgrund der Sachlage eigentlich unmöglich ist, dass beide Animationszustände zur Laufzeit gleichzeitig aktiv sind. Andernfalls wären weitreichende Analysen notwendig, um über die Zulässigkeit entscheiden zu können.

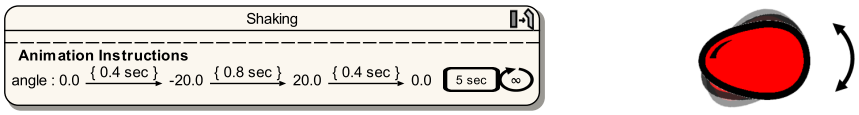


Abbildung 5.13: Animationszustand eines wackelnden Alligatoreis

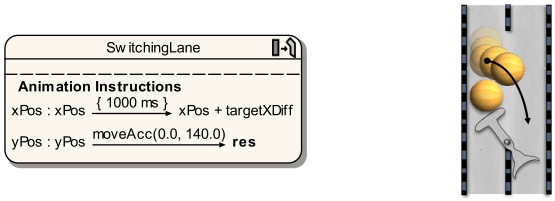


Abbildung 5.14: Avalanche-Murmel wechselt ihre Bahn

Beispiel 5.7 (Animationszustand für Alligatorei)
Für Alligator Eggs werden zahlreiche Animationszustände benötigt, wobei auch die Verkettung mehrerer Animatoren und Animatormodifikationen eingesetzt werden. Abb. 5.13 zeigt den Animationszustand *Shaking* eines Alligatoreis, der beide Möglichkeiten verwendet.

Die Verkettung wird genutzt, um den Wert des Rotationswinkels des Eis nacheinander von 0,0 über −20,0 und 20,0 zurück auf 0,0 zu ändern. Die Änderung erfolgt dabei über einen Zeitraum von insgesamt 1,6 Sekunden (0,4 sec, 0,8 sec und 0,4 sec). Der Wert wird zwischen den einzelnen Abschnitten linear interpoliert, da kein Animator explizit angegeben und somit der Animator für lineare Interpolation verwendet wird.

Zusätzlich wird modelliert, dass dieser Animationsverlauf unendlich oft wiederholt werden soll. Nachdem der letzte Wert – nach jeweils 1,6 Sekunden – erreicht wird, soll der Rotationswinkel wieder auf den ersten Wert gesetzt werden und die zeitabhängige Wertänderung beginnt erneut. Die maximale Dauer der Animation ist allerdings auf 5 Sekunden gesetzt, d. h., dass die Animatorinstanz exakt nach dieser Dauer deaktiviert wird und die Animation stoppt. Der zeitabhängige Wert von *angle* zu diesem Zeitpunkt beträgt −10,0. Dieser Wert wird unmittelbar nach der Animation auch verwendet, um den neuen statischen Wert des Attributs zu setzen. △

Beispiel 5.8 (Animationszustand für den Bahnwechsel einer Murmel)
In Avalanche kommen Animationszustände ebenfalls zum Einsatz. Animationszustand *SwitchingLane* in Abb. 5.14 beschreibt die Bewegung einer Kugel, die ihre Bahn aufgrund der Kollision mit einer blockierten Kugel ändern muss. Da die Position durch die Attribute *xPos* und *yPos* festgelegt wird und beide verändert werden müssen, werden zwei Animationsanweisungen innerhalb des Animationszustands benötigt.

Die Wertänderung für die x-Position wird mit dem Animator für lineare Interpolation durchgeführt. Die Änderung beginnt bei der aktuellen x-Position

und endet an einer um *targetXDiff* versetzten Stelle. Der Wert für Attribut *targetXDiff* muss dabei vor Beginn der Animation gesetzt werden. Auf diese Weise kann der Animationszustand sowohl für einen Bahnwechsel von links nach rechts (positiver Wert), als auch für den Wechsel von rechts nach links (negativer Wert) genutzt werden. Nach 1000 Millisekunden soll der Zielwert erreicht werden, und die Animation des Attributs *xPos* ist beendet.

Die y-Position wird mit dem Animator *moveAcc* verändert, der eine gleichmäßig beschleunigte Wertänderung umsetzt. Als Parameterwert für die Anfangsgeschwindigkeit wird 0,0 verwendet, während die Beschleunigung 140,0 beträgt. Der Animator benötigt weder die Angabe einer Dauer noch eines Zielwert. Als Zielwert wird daher das Schlüsselwort *res* angegeben. Die Animation wird fortgesetzt, solange der Animationszustand aktiv ist, da *moveAcc* im gegebenen Fall eine unendliche Animationsdauer vorsieht. △

5.4.5 Sensoren

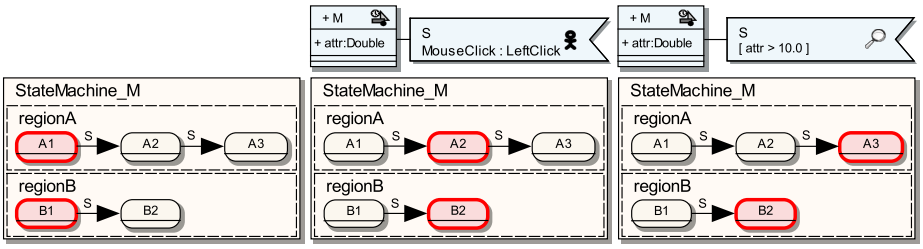
Ein weiteres wichtiges Element von AML sind *Sensoren*. Sie sind strukturelle Elemente der AML und können benutzt werden, um Ereignisse und Bedingungen innerhalb des Systems zu überwachen. Welche Ereignisse und Bedingungen ein Sensor überwacht, wird durch dessen Art und mögliche Parameter bestimmt, wobei (zusätzliche) Bedingungen immer mittels UML-Zusicherungen spezifiziert werden können.

Im strukturellen Teil des AML-Modells müssen Sensoren einem *Sensoreigentümer* zugeordnet werden, wobei ausschließlich Medienkomponenten als Eigentümer zulässig sind. Zur Systemlaufzeit wird der Sensor dann zusammen mit seinem Eigentümer instanziiert und beginnt nach dessen Erzeugung mit der Überwachung des Systems. Die *Sensorinstanz* überwacht dabei in der Regel Ereignisse und Bedingungen, welche die zugehörige Medienkomponenteninstanz (in diesem Kontext auch *Sensoreigentümerinstanz*) betreffen.

Nachfolgend wird beschrieben, wie Sensoren innerhalb von Zustandsautomaten eingesetzt werden können. Anschließend werden die wichtigsten Arten von Sensoren beschrieben, die in AML verfügbar sind: Bedingungssensoren (*ConstraintSensor*) mit der wichtigen Unterart Kollisionssensoren (*CollisionSensor*), Benutzersensoren (*UserSensor*) und Nachrichtensensoren (*MessageSensor*).

Sensoren als Trigger und Ereignis

Während einige Sensorarten an *Nachrichten* geknüpft werden (Benutzer- und Nachrichtensensoren), gibt es Sensorarten, die nicht an Nachrichten geknüpft werden (Bedingungs- und Kollisionssensoren). Ist ein Sensor nicht an eine Nachricht geknüpft, so ist eine Sensorinstanz *dauerhaft* aktiv, sofern bestimmte Bedingungen erfüllt werden. Die Sensorinstanz wird erst dann deaktiviert, wenn die Bedingung nicht mehr erfüllt wird. Ist ein Sensor an eine Nachricht geknüpft und werden mögliche Zusatzbedingungen zu diesem Zeitpunkt erfüllt, so wird



Ausgangssituation (a) | flüchtige Aktivierung (b) | dauerhafte Aktivierung (c)

Abbildung 5.15: Sensortrigger und Sensoraktivierung

die Sensorinstanz *flüchtig* aktiviert, d. h., sie wird sofort deaktiviert, nachdem das System auf die Aktivierung reagieren konnte.

Solange die Sensorinstanz aktiviert ist, kann sie Zuständsänderungen in ihrer Sensoreigentümerinstanz auslösen. Für diesen Zweck kann ein Sensorname als UML-Trigger, in diesem Fall *Sensortrigger* genannt, an einer Transition notiert werden. Solange eine Sensorinstanz aktiviert ist, dürfen Transitionen mit entsprechendem Sensortrigger geschaltet werden, sofern auch die Wächterbedingung der Transition erfüllt wird.

Im Falle einer flüchtigen Aktivierung ist zu beachten, dass lediglich eine einzige Transition und ggf. parallele Transitionen geschaltet werden dürfen, nicht aber sequentielle Transitionen.[12] Die flüchtige Aktivierung einer Sensorinstanz entspricht daher im Sinne der Semantik von UML-Zustandsautomaten einem Ereignis, das in eine interne Ereigniswarteschlange eingetragen und vom Zustandsautomaten im Rahmen eines „Run-to-completion“-Schritts verarbeitet wird. Danach wird das Ereignis „konsumiert“. Falls keine Transition das Ereignis konsumieren kann, wird es sofort entfernt (vgl. [UML11]).

Eine dauerhafte Aktivierung kann ebenfalls als Ereignis interpretiert werden. Allerdings wird das Ereignis niemals „konsumiert“, d. h., es bleibt in der internen Ereigniswarteschlange. Es wird aber entfernt, sobald die Sensorinstanz wieder deaktiviert wird. Ein solches Konzept bzw. Ereignis wird bspw. in [Bee94] als „continuous event“ bezeichnet.

Beispiel 5.9 (Flüchtige und dauerhafte Aktivierung)
In Abb. 5.15 wird das unterschiedliche Verhalten von flüchtiger und dauerhafter Aktivierung dargestellt. Sie zeigt eine Medienkomponente M mit zwei unterschiedlichen Sensoren S. Die jeweiligen Zustandsautomaten sind identisch. Sensor S wird als Sensortrigger an allen Transitionen notiert, d. h., die Transitionen können geschaltet werden, sobald die jeweilige Sensorinstanz von S aktiviert wird.

[12] Die Begriffe „parallel“ und „sequentiell“ beziehen sich dabei auf die Region, in der die Transitionen modelliert wurden (in der gleichen Region oder orthogonalen Regionen) und nicht etwa auf die gleichzeitige Ausführung von Zustandsübergängen, da Zustandsübergänge zeitlich immer sequentiell durchgeführt werden (vgl. Abschnitt 5.3.2, S. 130).

In der Ausgangssituation befindet sich eine Medienkomponenteninstanz von M in der in (a) gezeigten Zustandskonfiguration, d. h., die Zustände *A1* und *B1* sind aktiv. Wird nun die zugehörige Sensorinstanz von S aktiviert, d. h., eine überwachte Nachricht trifft ein (b) oder *attr* wird auf einen Wert über 10,0 gesetzt (c), so führt dies zu einer Zustandsänderung. Welche Zustandskonfiguration letztendlich nach der Aktivierung der Sensorinstanz eingenommen wird, ist davon abhängig, ob die Aktivierung flüchtig oder dauerhaft ist. Bei einer flüchtigen Aktivierung (b) resultieren die Zustände *A2* und *B2*. Bei dauerhafter Aktivierung (c) resultieren die Zustände *A3* (nach *A2*) und *B2*, da die Sensorinstanz aktiv bleibt. △

Es dürfen niemals zwei Sensorinstanzen gleichzeitig flüchtig aktiviert werden. In der Regel werden sie in der Reihenfolge aktiviert und abgearbeitet, in denen die Nachrichten, die zur flüchtigen Aktivierung führen, im System ankommen. Für den Fall, dass sich keine Reihenfolge bestimmen lässt, legt die *Priorität* der Sensoren fest, in welcher Reihenfolge die flüchtige Aktivierung stattfindet. Sensoren mit höherem Prioritätswert werden dabei zuerst aktiviert. In ähnlicher Weise wird auch die Priorität von dauerhaft aktivierten Sensoren ausgewertet. Muss zwischen zwei aktiven Transitionen gewählt werden, die aufgrund der dauerhaften Aktivierung von Sensorinstanzen schalten können, so wird die Transition gewählt, deren Sensortrigger (bzw. der zugehörige Sensor) den höheren Prioritätswert besitzt. Im Falle eines gleichen Prioritätwerts ist nicht definiert, welcher Sensor bzw. welche Transition gewählt wird (vgl. Abschnitt 5.3.2, S. 131).

Bedingungssensor

Bedingungssensoren sind in der Lage bestimmte Bedingungen bzw. den Zustand des Systems zu überwachen. Sie arbeiten selbstständig und ohne auf bestimmte Nachrichten zu reagieren. Sensorinstanzen werden daher (dauerhaft) aktiviert, solange eine formulierte Bedingung erfüllt wird. Dabei können sowohl Bedingungen in Bezug zur Sensoreigentümerinstanz als zu anderern Medienkomponenteninstanzen überwacht werden. Letzteres ist durch die Modellierung sogenannter *Verursacher* für einen Sensor möglich.

Formal können Bedingungssensoren wie folgt charakterisiert werden:

Definition 5.16 (Bedingungssensor)
Ein *Bedingungssensor* über eine Typmenge $\mathcal{T}$ ist ein Tripel $cs = (csow, csop, csc)$ mit folgenden Komponenten:

- $csow \in \mathcal{M}$ ist der Eigentümer des Bedingungssensors.
- $csop = (cop_1, \ldots, cop_{csopn}) \in \mathcal{M}^{csopn}$ ist das Tupel der *Verursacher* mit der Anzahl der Verursacher $csopn \in \mathbb{N}_0$.
- $csc : \mathcal{OBJ}_{\{csow\}} \times \mathcal{OBJ}_{\{cop_1\}} \times \ldots \times \mathcal{OBJ}_{\{cop_{csopn}\}} \times \mathcal{SYS}_{\mathcal{T}} \times \mathbb{T} \to \mathbb{B}$ ist das (mehrstellige) Prädikat des Bedingungssensors.

$\mathcal{CS}_{\mathcal{T}}$ bezeichne die Menge aller Bedingungssensoren über eine Typmenge $\mathcal{T}$.△

Das Prädikat des Bedingungssensors formuliert die vom Sensor überwachte Bedingung. Es wird modelliert als UML-Zusicherungen des Bedingungssensors. Möglich sind sowohl grobe Beschreibungen als auch präzise Formulierungen mittels OCL-Ausdrücken oder direkt mittels Code der Zielplattform.

Das Prädikat kann in jedem Fall die Sensoreigentümerinstanz als Subjekt einbeziehen. Der für das Prädikat formulierte Ausdruck befindet sich sogar im Kontext der Sensoreigentümerinstanz, d. h., sie kann mit einem Schlüsselwort wie *self* (OCL) oder *this* (Java) referenziert werden. Zusätzlich können im Ausdruck auch andere Medienkomponenteninstanzen referenziert werden, um bspw. eine bestimmte Korrelation zwischen den Medienkomponenteninstanzen zu überwachen. Hierfür ist es notwendig, andere Medienkomponenten als Verursacher des Bedingungssensors zu modellieren. Der Ausdruck des Prädikats kann dann auch Medienkomponenteninstanzen der modellierten Verursacher (*Verursacherinstanzen*) einbeziehen, wobei der *Verursachername* die jeweilige Verursacherinstanz referenziert. Auch der aktuelle Systemzustand kann verwendet werden. Dies ermöglicht zusammen mit der aktuellen Systemzeit die Überwachung der zeitabhängigen Werte von Attributen (vgl. Definition 5.7, S. 148).

Die folgende Definition zeigt, unter welchen Umständen ein Bedingungssensor als aktiviert gilt:

Definition 5.17 (Aktivierung einer Bedingungssensorinstanz)
Gegeben seien der aktuelle Systemzustand $sys \in \mathcal{SYS}_{\mathcal{T}}$, ein Bedingungssensor $cs = (csow, (cop_1, cop_2, \ldots, cop_{csopn}), csc) \in \mathcal{CS}_{\mathcal{T}}$ und eine Medienkomponenteninstanz $obj_{csow} \in \mathcal{OBJ}_{csow}$.

Eine *Bedingungssensorinstanz* des Typs cs, die zur Sensoreigentümerinstanz obj_{csow} gehört, ist zum Zeitpunkt $t \in \mathbb{T}$ *aktiviert*, falls gilt

$$\begin{aligned}\exists (obj_1, obj_2, \ldots, obj_{csopn}) \in (\mathcal{OBJ}_{\{cop_1\}} \times \mathcal{OBJ}_{\{cop_2\}} \times \ldots \times \mathcal{OBJ}_{\{cop_{csopn}\}}) : \\ csc(obj_{csow}, obj_1, obj_2, \ldots, obj_{csopn}, sys, t) \quad \wedge \\ |\{obj_{csow}, obj_1, obj_2, \ldots, obj_{csopn}\}| = csopn + 1 \quad .\end{aligned}$$ △

Die Sensoreigentümerinstanz, referenziert durch obj_{csow}, ist dabei festgelegt. Für die Verursacherinstanzen, referenziert durch $obj_1, obj_2, \ldots, obj_{csopn}$, wird das gesamte System durchsucht. Existiert im System eine Kombination, mit dem das Prädikat des Bedingungssensors als wahr ausgewertet werden kann, so wird der Sensor (bzgl. der gefundenen Verursacherinstanzen) aktiviert. Gemäß Definition ist dabei wichtig, dass sich alle Medienkomponenteninstanzen paarweise voneinander unterscheiden, d. h., unterschiedliche gefunden werden.

Die gefundenen Verursacherinstanzen können dann im Rahmen des Wächters oder des Effekts einer Transition mit entsprechendem Sensortrigger verwendet werden, wobei erneut der Verursachername der Referenzierung dient. Sollten mehrere Tupel von Verursacherinstanzen zur Aktivierung des Bedingungssensors gefunden werden können, wird zur Laufzeit ein beliebiges Tupel gewählt.

Beispiel 5.10 (Vergleich von Bedingungssensor und Wächter)
Abb. 5.16 zeigt, wie ein einfacher Bedingungssensor S (ohne Verursacher) benutzt wird. Seine Bedingung wird durch den Ausdruck $attrA > 10{,}0$ spezifiziert. In

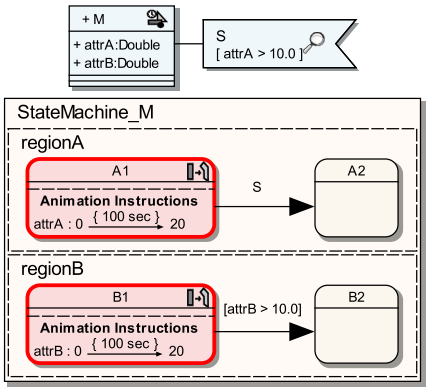


Abbildung 5.16: Vergleich von Bedingungssensor und Wächter

der dargestellten Situation befindet sich eine Medienkomponenteninstanz von M in den Animationszuständen *A1* und *B1*. Während diesen Zuständen werden die Attribute *attrA* und *attrB* jeweils durch lineare Interpolation innerhalb von 100 Sekunden von 0,0 auf 20,0 geändert. Der Wert 10,0 wird jeweils nach 50 Sekunden erreicht. Zu diesem Zeitpunkt wird also auch der Bedingungssensor S aktiviert. Dies führt zum sofortigen Zustandsübergang von *A1* zu *A2*. Der neue statische Wert von *attrA* entspricht danach 10,0.

Diese Möglichkeit zur Unterbrechung unterscheidet Sensortrigger wesentlich von einer Transition, an der ein Wächter notiert ist, wie z. B. bei der Transition von *B1* zu *B2*. Für eine Transition ohne explizite Angabe eines Triggers wird immer der Abschlussereignis-Trigger angenommen, d. h., die Transition wird nur geschaltet, falls die modellierte Bedingung exakt zum Zeitpunkt des Abschlussereignisses erfüllt wird. Im konkreten Fall wird die Animation zuerst beendet, d. h., nach 100 Sekunden erreicht *attrB* den Wert 20,0 und das Abschlussereignis tritt ein. Daraufhin wird der Zustand *B2* eingenommen, da die Bedingung $attrB > 10{,}0$ erfüllt wird. △

Kollisionssensor

Neben gewöhnlichen Bedingungssensoren bietet AML sogenannte *Kollisionssensoren*. Diese sind Bedingungssensoren, die immer ein besonderes Prädikat mit einschließen, welches zusätzlich zu möglichen anderen Bedingungen gelten muss. Dieses besondere Prädikat ist die räumliche Kollision oder auch Überlappung der Sensoreigentümerinstanz mit einer oder mehreren anderen Medienkomponenteninstanzen. Aus diesem Grund muss für diesen Sensortyp mindestens ein Verursacher als potentieller Gegenpart für eine Kollision angegeben werden.

Kollisionssensoren werden als eigene Sensorenart in AML eingeführt, weil Kollisionen und dadurch notwendige Reaktionen in vielen animierten Systemen benötigt werden. Gleichzeitig ist es nicht trivial, eine Kollisionsbedingungen als textuellen Ausdruck in einem gewöhnlichen Bedingungssensor zu formulieren.

Außerdem sind Kollisionssensoren ein Beispiel dafür, dass weitere Arten von Sensoren für besondere Formen der Überwachung spezifiziert werden können.

Benutzersensor

Benutzersensoren werden modelliert, um Interaktionen zwischen Benutzer und System überwachen zu können. Der Begriff „Benutzer“ ist dabei abstrakt zu verstehen, da es sich sowohl um einen menschlichen Benutzer als auch um ein beliebiges anderes „externes System“ handeln kann, das mit dem modellierten System kommuniziert. Somit ist es die Aufgabe von Benutzersensoren Nachrichten zu überwachen, die „von außen“ an das System geschickt werden. Dabei beachtet eine Benutzersensorinstanz nur Nachrichten, die an seine Sensoreigentümerinstanz gesendet werden.

Typischerweise wird ein solcher Sensor flüchtig aktiviert, wenn mit der Maus auf die Medienkomponenteninstanz geklickt wird oder bei Tastenanschlägen, während die Instanz selektiert ist. Welche Benutzernachrichten im konkreten Fall zur Aktivierung führen, kann durch eine Detailspezifikation in den Attributen des Sensors angegeben werden. AML stellt mit *UserSensorType* einige konventionelle Typen von Nachrichten zur Verfügung, die durch das Attribut *type* zugewiesen werden, z. B. „Mausklick“, „Tastendruck“ etc. Weitere Details können durch Attribut *details* folgen, z. B. der Name der Maus-/Tastaturtaste o. ä. Die angebotenen Möglichkeiten sind aber lediglich als Beispiele zu verstehen, da AML die jeweilige Interpretation von solchen Detailspezifikationen einer konkreten Implementierung überlässt und dadurch beliebig erweitert werden kann.

Nachrichtensensor

Für Medienkomponenten, die auf Nachrichten von anderen Medienkomponenten reagieren müssen, können *Nachrichtensensoren* eingesetzt werden. Diese Sensorenart ist also ein Hilfsmittel, um Interaktionen zwischen den Medienkomponenten zu realisieren.

Jede Nachrichtensensorinstanz kann Nachrichten eines bestimmten *Nachrichtentyps* empfangen. Dieser Nachrichtentyp muss für den Nachrichtensensor festgelegt werden. Der Nachrichtentyp selbst ist ein Modellelement, für das ein beliebiger Name bestimmt werden muss.

Die Sensorinstanz wird flüchtig aktiviert, sobald eine Nachricht des spezifizierten Typs an ihre Sensoreigentümerinstanz geschickt wird. Das Senden einer Nachricht erfolgt dabei über das Aktionssequenzelement „Nachricht senden“ (siehe Abschnitt 5.4.6, S. 170).

Auch für einen Nachrichtensensor können Verursacher spezifiziert werden. Im Gegensatz zu Bedingungssensoren repräsentieren Verursacher von Nachrichtensensoren aber die Medienkomponenten, die entsprechende Nachrichten versenden. Es können auch mehrere Verursacher modelliert werden, um mehrere Typen von Sendern abzudecken.

Bei der Aktivierung der Nachrichtensensorinstanz wird überprüft, ob der Typ des Nachrichtensenders (auch *Senderinstanz*) einem spezifizierten Verursacher

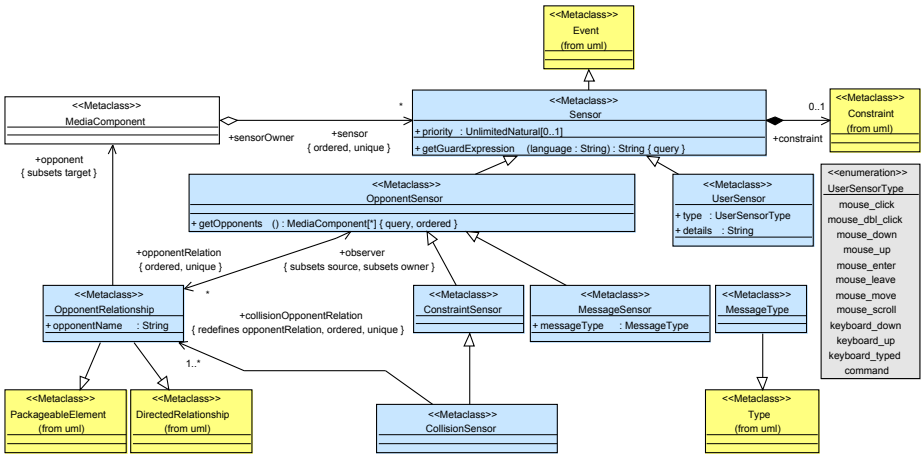


Abbildung 5.17: AML-Metamodell: Sensoren

oder einer davon abgeleiteten Medienkomponente entspricht, wobei Senderinstanz und Sensoreigentümerinstanz identisch sein dürfen. Ist dies der Fall, so kann die Senderinstanz mit dem Verursachernamen referenziert werden. Falls der Typ der Senderinstanz nicht kompatibel mit dem Verursacher ist, so ist die Referenz nicht bzw. auf *null* gesetzt.

Nachrichtensensoren ermöglichen zusammen mit Nachrichtentypen und dem Versenden von Nachrichten eine ähnliche Funktionalität wie UML-Signale, Signalereignisse und *SendSignal*-Aktionen. Die UML-Gegenstücke sind allerdings komplexer und benötigen spezielle Modellelemente, die in AML-Modellen typischerweise nicht eingesetzt werden. So benötigen SendSignal-Aktionen bspw. einen Eingabe-Pin als Ziel, und Signale können Attribute enthalten [UML11], d. h., dass auch ein Objektfluss in den Diagrammen modelliert werden muss. Die entsprechenden AML-Konstrukte bieten solche erweiterten Möglichkeiten nicht, sind aber einfacher im Zustandsdiagramm zu verwenden und fügen sich nahtlos in die bisherigen Konzepte und die Notation mittels Sensoren ein.

Metamodell Wie Abb. 5.17 zeigt, ist die Metaklasse für Sensoren namens *Sensor* von UML-Ereignissen (*uml::Event*) abgeleitet, um als Sensortrigger verwendet werden zu können. Zudem kann jeder Sensor optional eine UML-Zusicherung beinhalten. Diese Zusicherung stellt dann die (zusätzliche) Bedingung des Sensors dar, um aktiviert zu werden. Zur Spezifikation der Priorität des Sensors wird das Attribut *priority* zur Verfügung gestellt.

Medienkomponenten können eine beliebige Anzahl von Sensoren besitzen. In Bezug auf einen Sensor ist die Medienkomponente in der Rolle des Sensoreigentümers (*sensorOwner*). Der Sensor wird jedoch als eigenständiges Element innerhalb eines Packages modelliert und wird nicht der Medienkomponente untergeordnet.

Für jede Sensorart existiert eine eigene Metaklasse. Darin sind die spezifischen Attribute untergebracht. Ein Benutzersensor (*UserSensor*) muss den Typ sowie

zusätzliche Angaben in den Attributen *type* und *details* festhalten. Als Typen stehen dabei die Literale der Enumeration *UserSensorType* zur Verfügung. Sie umfassen neben unterschiedlichen Maus- und Tastaturereignissen auch einen vielseitig verwendbaren Typ *command*. Die restlichen Sensorarten Bedingungssensor (*ConstraintSensor*), Kollisionssensor (*CollisionSensor*) und Nachrichtensensor (*MessageSensor*) sind Sensoren, für die Verursacher modelliert werden können und dementsprechend von *OpponentSensor* abgeleitet werden. Für die Modellierung der Verursacher steht Metaklasse *OpponentRelationship* zur Verfügung, die wie andere gerichtete UML-Beziehungen von *DirectedRelationship* abgeleitet wird und auch so verwendet wird, d. h., sie verbindet Sensor mit Verursacher. Für den vereinfachten Zugriff auf die Verursacher kann außerdem die Methode *getOpponents* in *OpponentSensor* verwendet werden (siehe Listing B.1/38).

Nachrichtensensoren müssen schließlich den Typ der überwachten Nachricht mittels *messageType* angeben. Der Nachrichtentyp selbst ist Teil des Modells. Die zugehörige Metaklasse heißt *MessageType*. Sie wird von Typ *uml::Type* abgeleitet. Da sie nicht von *uml::Classifier* abgeleitet wird (vgl. UML-Signale), unterstützen Nachrichtentypen keine Generalisierung, und sie können auch keine Merkmale besitzen. Obwohl beide Konzepte vorstellbar wären, wurde aufgrund der zusätzlichen Komplexität davon abgesehen. △

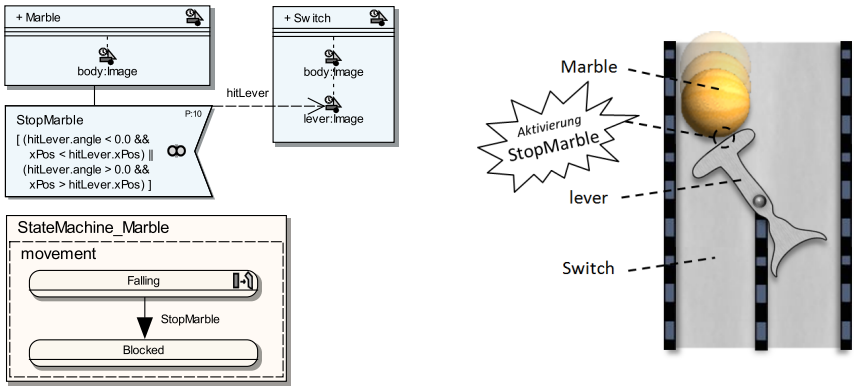
Zusicherungen Ein Sensor darf nur dann als Trigger verwendet werden, wenn er innerhalb eines Zustandsautomatens eingesetzt wird, der das Verhalten des Sensoreigentümers oder einer davon abgeleiteten Medienkomponente modelliert (siehe Listing B.2/24). △

Notation Sensoren werden durch Blöcke mit Einkerbung an der rechten Seite dargestellt (vgl. *StopMarble* in Abb. 5.18). Die Priorität des Sensors wird, falls der Prioritätswert ungleich 0 ist, mit einem kleinen „P: *Wert*“ in der oberen rechten Ecke notiert. Der Sensoreigentümer muss mittels durchgezogener Linie mit dem Sensor verbunden werden. Zwischen dem Sensor und seinen Verursachern werden hingegen gestrichelte Beziehungspfeile mit einfachem Pfeilkopf in Richtung des Verursachers verwendet. Sowohl der Beginn der durchgezogenen Linie als auch der Pfeil können dabei mit einem inneren Merkmal verbunden sein. Als Sensoreigentümer bzw. Verursacher des Sensors gilt in einem solchen Fall nicht das innere Merkmal, sondern die implizite Medienkomponente des inneren Merkmals (vgl. Beispiel 5.2, S. 138).

Bedingungssensor	*«ConstraintSensor»*	
Kollisionssensor	*«CollisionSensor»*	
Benutzersensor	*«UserSensor»*	
Nachrichtensensor	*«MessageSensor»*	

Tabelle 5.4: Stereotyp-Icons für Sensorarten

Der Sensortyp wird als Stereotyp(-Icon) an der rechten Seite des Sensorblocks notiert. Tab. 5.4 zeigt eine Übersicht über verfügbare Stereotypen.



(a) (b)

Abbildung 5.18: Kollisionssensor in Avalanche

Benutzersensoren müssen angeben, auf welche Benutzernachricht reagiert werden soll. Diese Angabe wird direkt unter dem Sensornamen mit Syntax „*Benutzersensortyp : Details*“ notiert. Bei Spezifikation von „*KeyboardDown : A*“ wird eine Sensorinstanz bspw. aktiviert, wenn eine Nachricht über das Drücken der Taste „A“ an die Sensoreigentümerinstanz geschickt wird. In ähnlicher Weise muss bei Nachrichtensensoren direkt unter dem Sensornamen der Nachrichtentyp angegeben werden. Die Syntax lautet: „*<Nachrichtentyp>*“.

Nachrichtentypen werden ähnlich wie primitive Datentypen modelliert. Das Rechteck, das den Typ darstellt, enthält lediglich den Namen und das Stereotyp-Icon, das auch für Nachrichtensensoren verwendet wird (vgl. *MoveTokenToTrans* in Abb. 5.19). Alternativ kann auch *«MessageType»* textuell hinzugefügt werden.

Die (zusätzliche) Bedingung eines Sensors wird bei jeder Sensorart am unteren Sensorrand in einem „[]“-Block festgehalten. Die Schreibweise soll dabei an die typische Schreibweise für Wächter erinnern. △

Beispiel 5.11 (Kollisionssensor in Avalanche)
Bei der Modellierung von Avalanche werden Kollisionssensoren benötigt, um Kollisionen von Murmeln mit anderen Murmeln oder Hebeln zu überwachen. Abb. 5.18 (a) zeigt einen Kollisionssensor namens *StopMarble*. In der oberen rechten Ecke wird dessen Prioritätswert 10 dargestellt. Der Kollisionssensor gehört zur Medienkomponente einer Murmel (*Marble*). Als Verursacher wird der Hebel (*lever*) einer Weiche (*Switch*) modelliert.

Der Kollisionssensor ist dafür verantwortlich, dass eine fallende Murmel stoppt, sobald sie auf einen solchen Hebel trifft. Eine derartige Situation wird in Abb. 5.18 (b) illustriert. Ohne die zusätzlich formulierte Bedingung würde der Sensor aktiviert werden, sobald sich Murmel und Hebel berühren. Allerdings soll die Bedingung sicherstellen, dass die Murmel die Oberseite des Hebels trifft. Dies ist entweder der Fall, wenn der Hebel nach links gekippt ist – überprüft durch den Winkel *angle* des Hebels *lever*, referenziert durch den Verursachernamen

hitLever) – und die x-Position der Murmel sich links vom Zentrum des Hebels befindet, oder wenn Hebel und Murmel sich rechts befinden. Nur in diesen beiden Situationen soll die Murmel auch stoppen. Andernfalls trifft die Murmel den Hebel weiter unten, was dazu führt, dass der Hebel umgelegt wird.

Der Zustandsautomat von *Marble* zeigt den fallenden Zustand der Murmel (*Falling*) und den blockierten Zustand (*Blocked*). Sobald Sensor *StopMarble* aktiviert wird, soll der Zustand von *Falling* auf *Blocked* gewechselt werden. Dies wird durch die Transition mit Sensortrigger *StopMarble* modelliert. △

Beispiel 5.12 (Benutzer-/Bedingungs-/Nachrichtensensor für B/E-Netze)
Die Dynamik in B/E-Netzen lässt sich ebenfalls gut mit AML-Sensoren und Zustandsautomaten beschreiben. In Abb. 5.19 werden drei der vorgestellten Sensorentypen verwendet. Für das Beispiel werden außerdem die Medienkomponenten *Place* und *Transition* benötigt. Sie stellen die jeweilige Repräsentation von Stellen und Transitionen des Netzes dar. Es wird zudem davon ausgegangen, dass jede Transition die Menge der Stellen des Vorbereichs und des Nachbereichs kennt. Auf diese kann im Rahmen dieses Beispiels mit den Referenzen *input* und *output* zugegriffen werden. Die Abfrage *isEnabled* soll zurückgeben, ob die Transition derzeit aktiv ist.

Der Benutzersensor *Fire* überwacht Klicks der linken Maustaste auf eine Transition. Der zugehörige Zustandsautomat *StateMachine_Transition* zeigt einen entsprechenden Sensortrigger *Fire* an dem Zustandsübergang von *NoFiring* auf *Firing/To*. Dieser Zustandsübergang wird somit bei einem Mausklick auf die Transition durchgeführt. Durch die festgelegte Zusatzbedingung reagiert der Sensor allerdings nur, d. h., es wird nur dann auf Mausklicks reagiert und der Zustand gewechselt, falls die Transition auch aktiv ist.

Die Aufgabe des Zustands *Firing/To* ist es, die erste Phase des Schaltvorgangs des Netzes einzuleiten. In dieser Phase müssen alle Stellen des Vorbereichs ihre Tokens in Richtung der Transition schicken. Daher muss für jede entsprechende Stelle eine Animation gestartet werden. Zum Starten wird der Zustandsübergang mit Sensortrigger *NoTokenMovementToTrans* genutzt. Der zugehörige Bedingungssensor ist aktiviert, wenn eine Stelle mit angegebener Bedingung gefunden wird. Diese Bedingung gibt unter Verwendung von Java-Ausdrücken an, dass die Stelle im Vorbereich sein muss (*input.contains*(*place*)) und die Animation für diese Stelle noch nicht begonnen hat (!*place.isAnimated*()). Die Referenz *place* gibt dabei die jeweilige Verursacherinstanz an, d. h. die überprüfte Stelle.

Falls eine solche Stelle gefunden wird und der Sensor daher aktiviert ist, wird der entsprechende Zustandsübergang ausgeführt (eine Schleife zurück zu *Firing/To*). Dabei wird ein Effekt ausgeführt, der die Nachricht *MoveTokenToTrans* an die gefundene Verursacherinstanz *place* sendet (entsprechende Effekte werden in Abschnitt 5.4.6, S. 169, beschrieben). Diese Nachricht kann von einer Nachrichtensensorinstanz (*MessageMoveTokenTo*) der Stelle empfangen werden. Der Zustandsautomat für Stellen *StateMachine_Place* zeigt, dass dadurch in den Zustand *MoveTokenToTransition* gewechselt wird. Dies geschieht unmittelbar nach dem Senden der Nachricht, da Trigger für Nachrichtensensoren ggü. den Triggern

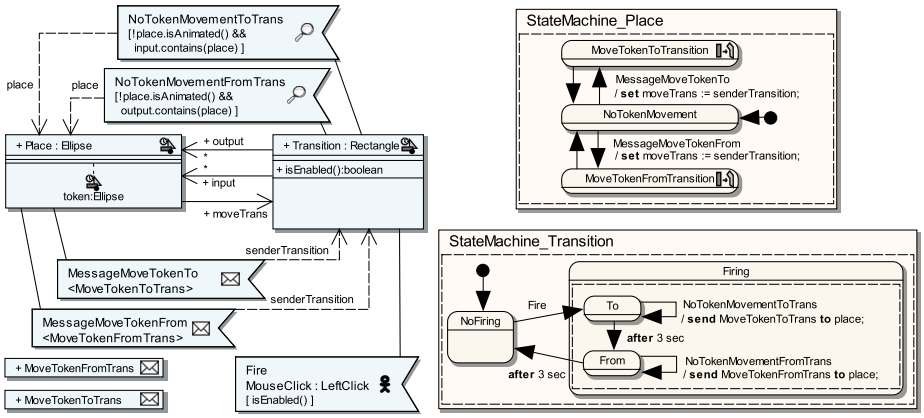


Abbildung 5.19: Sensoren in B/E-Netzen

für Sensoren mit dauerhafter Aktivierung (z. B. *NoTokenMovementToTrans*) bevorzugt werden (vgl. Tab. 5.2, S. 131).

Bei dem Wechsel in den Zustand *MoveTokenToTransition* wird das Attribut *moveTrans* auf die Senderinstanz der Nachricht gesetzt, welche durch Verursachername *senderTransition* referenziert wird. Das Attribut *moveTrans* wird eingesetzt, damit während der animierten Darstellung des Tokens (Unterkomponente *token*) die Position der Transition – es kommen nur Transitionen als Sender in Frage – als Ziel der Bewegung verwendet werden kann. Die Tokenanimation selbst wird dann durch den Animationszustand *MoveTokenToTransition* modelliert, der aber nicht im Detail dargestellt wird.

Nach drei Sekunden in Zustand *Firing/To* erfolgt der Zustandsübergang zu *Firing/From*. Drei Sekunden entsprechen dabei der Dauer, die auch für die Bewegungsanimation eines Tokens modelliert wird. Zustand *Firing/From* und die damit verknüpften Aktionen und Sensoren funktionieren analog zur bisherigen Vorgehensweise. Allerdings wird dadurch die zweite Phase des Schaltvorgangs eingeleitet, in der sich die Tokens zu den Stellen des Nachbereichs bewegen. Weitere drei Sekunden später ist auch dieser Vorgang beendet und Zustand *NoFiring* wird wieder eingenommen. △

5.4.6 Aktionsequenzen

Zustandsautomaten der UML erlauben es, für Zustandsübergänge und das Betreten („entry“) oder Verlassen („exit“) eines Zustands zusätzliche Effekte zu modellieren. Der Effekt muss dabei als UML-Verhalten (*uml::Behavior*), wie bspw. eine UML-Aktivität (*uml::Activity*), spezifiziert werden. Eine solche UML-Aktivität kann wiederum durch ein Aktivitätsdiagramm beschrieben werden. Diese Möglichkeit bietet zwar hohe Flexibilität, kann aber nur umständlich für einfache Effekte eingesetzt werden. Alternativ erlaubt es UML daher, einzelne

Aktionen mit implementierungsspezifischer Syntax und Semantik direkt zu notieren, z. B. innerhalb eines Modellelements vom Typ *uml::OpaqueBehavior*:

> The behavior expression may be an action sequence comprising a number of distinct actions including actions that explicitly generate events, such as sending signals or invoking operations. The details of this expression are dependent on the action language chosen for the model. ([UML11])

Um eine einfache und eindeutige Modellierung zu ermöglichen, führt die AML eine solche „action language“ ein. Mit der Sprache sollen (mehrere) typische Anweisungen ausgeführt werden können, sie soll verständlich sein und kompakt innerhalb des Zustandsautomaten dargestellt werden können. Die Aktionen sollen im Modell als Folge von *Aktionssequenzelementen*[13] mit beliebiger Anzahl modelliert werden. Diese sind in einem übergeordneten Element, genannt *Aktionssequenz*, eingebettet. Die Aktionssequenz stellt dabei das UML-Verhalten dar und kann somit als Effekt der Transition eingesetzt werden. Tritt ein Effekt mit Aktionssequenz ein, so werden alle in der Aktionssequenz befindlichen Aktionssequenzelemente sequentiell und in modellierter Reihenfolge ausgeführt.

Die AML definiert vier Arten von Aktionssequenzelementen: *Attribut setzen*, *Methode aufrufen*, *Nachricht senden* und *Instanz erzeugen*. Sie werden in den folgenden Unterabschnitten vorgestellt.

Aktionssequenzelement – Attribut setzen

Durch das *SetAttributeAction*-Element kann einem Attribut im Zustandsautomatenkontext ein neuer Wert zugewiesen werden.

Aktionssequenzelement – Methode aufrufen

Durch das *CallMethodAction*-Element können Methoden im Zustandsautomatenkontext aufgerufen werden.

Aktionssequenzelement – Instanz erzeugen

Bei Verwendung des *CreateAction*-Elements wird eine neue Instanz einer Klasse bzw. Medienkomponente erzeugt, indem der jeweilige Konstruktor aufgerufen wird. Die Referenz auf die erzeugte Instanz kann durch das *CreateAction*-Element direkt einem Attribut im Zustandsautomatenkontext zugewiesen werden.

Aktionssequenzelement – Nachricht senden

Das *SendMessageAction*-Element wird verwendet, um eine Nachricht an einzelne oder alle Medienkomponenteninstanzen (ein *Broadcast*) zu senden. Jeder Empfänger kann dabei einen oder mehrere Nachrichtensensorinstanzen für die

[13] Der Begriff „Aktionssequenzelement“ wurde deshalb gewählt, um ihn vom Begriff „(UML-)Aktion“ unterscheiden zu können.

entsprechende Nachricht besitzen (vgl. Abschnitt 5.4.5, S. 164). Beim Empfang werden die entsprechenden Sensorinstanzen dann flüchtig aktiviert. Nach dem Versenden der Nachricht wird sie wieder verworfen, unabhängig davon, ob die Empfänger die Nachricht durch vorhandene und aktivierbare Sensoren verarbeiten können oder nicht.

Zerstören von Instanzen

Instanzen werden in AML normalerweise nicht durch Aktionen zerstört bzw. es existiert kein entsprechendes Aktionssequenzelement. Es existieren allerdings andere Mechanismen, die zum Zerstören von Instanzen verwendet werden können.

Zum einen können die Zustandsautomaten der einzelnen Klassen für das Zerstören genutzt werden, da mit ihnen ebenfalls der Lebenszyklus einer Instanz modelliert werden kann. Der Terminierungsknoten (ein Pseudozustand, dargestellt durch ein großes „X“) kann in Zustandsdiagrammen dazu genutzt werden, das Ende des Lebenszyklus einer Instanz zu markieren. Die Instanz zerstört sich bei Erreichen dieses Pseudozustands selbst. Es ist natürlich möglich, dass andere Instanzen Zustandsübergänge zu einen solchen Pseudozustand auslösen (z. B. durch Nachrichten) und damit die zugehörige Instanz (indirekt) zerstören.

Eine weitere Möglichkeit ist die Verwendung von Kompositionsbeziehungen und deren Eigenschaften (vgl. [UML11]). Auch zwischen übergeordneten und untergeordneten Medienkomponenten wird dieser Beziehungstyp verwendet (vgl. Abschnitt 5.4.2, S. 137). Dadurch wird festgelegt, dass ein untergeordnetes Objekt dieser Beziehung nicht ohne das übergeordnete Objekt existieren kann. Um die untergeordnete Instanz zu zerstören, genügt es daher, die übergeordnete Instanz zu zerstören oder die Verknüpfung zwischen beiden Instanzen zu trennen, was durch die Änderung von Attributen bzw. Referenzen geschehen kann.

Metamodell Abb. 5.12, S. 156, zeigt die Metaklasse für Aktionssequenzen (*ActionSequence*). Sie ist von *uml::Behavior* abgeleitet und kann damit als Effekt einer Transition oder innerhalb eines Zustands („entry“ oder „exit“) eingesetzt werden. Eine Aktionssequenz kann beliebig viele Elemente (*ActionSequenceElement*) enthalten. Konkrete Klassen für Aktionssequenzelemente sind *SetAttributeAction*, *CallMethodAction*, *CreateAction* und *SendMessageAction*.

Jedes Aktionssequenzelement ist von *FeatureSelector* und somit *Feature* abgeleitet. Daher kann für jedes Aktionssequenzelement ein Merkmal im Zustandsautomatenkontext selektiert werden (*feature*). Auch Merkmale von Unterkomponenten sind damit erlaubt. Die konkrete Art des Aktionssequenzelements bestimmt, ob ein Attribut, ein Methode oder gar kein Merkmal gewählt werden muss. Sie bestimmt ebenfalls, wofür das gewählte Merkmal verwendet wird. So wird für *SetAttributeAction* oder *CreateAction* das jeweils zu setzende (bei *CreateAction* optionale) Attribut modelliert (*attribute*). Für *CallMethodAction* wird die aufzurufende Operation gewählt (*method*).

Ein *SetAttributeAction*-Element erfordert außerdem die Spezifikation des Werts (*value*) auf den das gewählte Attribut gesetzt werden soll. Bei einem *CallMethodAction*-Element müssen die Argumente (getrennt durch Komma) für den

Methodenaufruf angegeben werden (*arguments*). Für ein *CreateAction*-Element sind dies die Argumente für den Konstruktoraufruf (*constructorArgs*). Zusätzlich ist in diesem Fall erforderlich, die zu erzeugende Klasse anzugeben (*type*).

Für das Element *SendMessageAction* muss spezifiziert werden, welcher Nachrichtentyp verschickt werden soll (*type*). Optional ist die Spezifikation, an welche Medienkomponenteninstanz(en) die Nachricht geschickt werden soll (*receiver*). Dabei kann bspw. der Name eines Attributs im Zustandsautomatenkontext angegeben werden. Das Attribut muss dabei die Referenz des Empfängers enthalten. Es kann auch mehrere Referenzen (oder keine) enthalten, was zum Versenden entsprechend vieler Nachrichten führt. Falls *receiver* keine Zeichenkette enthält, so wird die Nachricht an alle Medienkomponenteninstanzen des gesamten Systems verschickt, einschließlich des Zustandsautomateneigentümers selbst, falls dies eine Medienkomponenteninstanz ist.

Für einige Angaben (*value*, und innerhalb von *arguments* und *receiver*) können Konstanten, einfache mathematische Ausdrücke oder wirkungsfreie Methodenaufrufe im Zustandsautomatenkontext verwendet werden. Auch die Verwendung von Verursachernamen zur Referenzierung von Verursacherinstanzen ist zulässig, falls das Aktionssequenzelement im Kontext einer Transition mit entsprechendem Sensor eingesetzt wird (vgl. *place* in Abb. 5.19, S. 169). △

Zusicherungen Für die beschriebenen Metaklassen existieren lediglich einfache Zusicherungen, die sich erwartungsgemäß für die jeweiligen Aktionssequenzelemente ergeben. Beispielsweise kann kein schreibgeschütztes Attribut gesetzt oder die Instanz einer abstrakten Klasse erzeugt werden. △

Aktionssequenzelement	Notation
Attribut setzen	**set** *attributname* := *wert*
Instanz erzeugen	**create** [*attributname* :=] *klasse*(*argument1*, ...)
Nachricht senden	**send** *nachrichtentyp* [**to** *empfänger*]
Methode aufrufen	**call** *methodenname*(*argument1*, *argument2*, ...)

Tabelle 5.5: Notation der Aktionssequenzelemente

Notation Aktionssequenzen werden textuell im Zustandsdiagramm als Effekt notiert. Die einzelnen Aktionssequenzelemente werden in gewünschter Reihenfolge notiert und durch Semikolon voneinander getrennt. Ein abschließender Semikolon ist optional.

Die spezifische Notation für die einzelnen Aktionssequenzelemente wird in Tab. 5.5 dargestellt. Die Schlüsselwörter sind darin in Fettdruck hervorgehoben, und die Abschnitte in „[]“-Blöcken sind die optionalen Angaben. Details wurden bereits in den vorangehenden Unterabschnitten erläutert. △

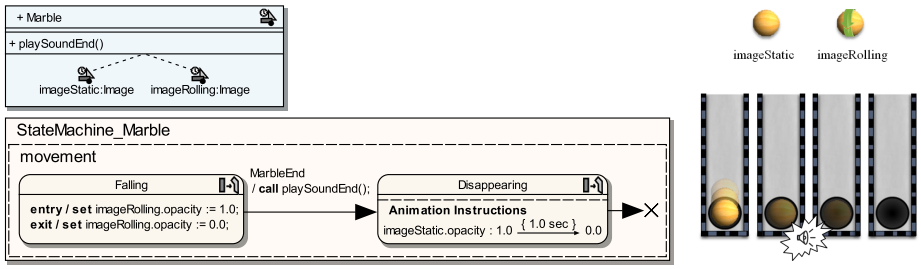


Abbildung 5.20: Attribut setzen, Methode aufrufen und Instanz zerstören

Beispiel 5.13 (Aktionssequenzen in Avalanche)
In Avalanche besteht die Medienkomponente *Marble* grafisch aus zwei Bildern bzw. inneren Merkmalen (siehe Abb. 5.20). Das erste Bild *imageStatic* zeigt eine Murmel in ruhendem Zustand. Das zweite Bild *imageRolling* ist ein animiertes Bild (bspw. unterstützt durch das Dateiformat *GIF* [Rob03]), das durch wechselnde Schattierungen eine rollende Murmel darstellt. Je nach aktuellem Zustand von *Marble* muss eines der beiden Bilder angezeigt werden, während das andere Bild komplett unsichtbar geschaltet ist.

Standardmäßig ist keines der beiden Bilder sichtbar. Wird allerdings der Zustand *Falling* aktiv, so wird die Deckkraft (*opacity*) von *imageRolling* auf ihr Maximum gesetzt. Hierfür wird ein *SetAttributeAction*-Element benutzt, das bei Eintritt des Zustands ausgeführt wird („entry“). Während des Zustands wird also eine rollende Kugel angezeigt, wobei sich der Begriff „rollend“ in diesem Zusammenhang nur auf das Bild und nicht auf eine etwaige Positionsänderung bezieht. Sobald der Zustand verlassen wird („exit“), muss das Attribut *opacity* des Bildes wieder auf das Minimum zurückgesetzt werden, d. h., die rollende Kugel ist nicht mehr sichtbar.

Der untere Teil des Beispiels zeigt außerdem, dass die Murmel auf einen Zustand *Disappearing* wechseln kann, z. B. falls sie das Ende der Bahn erreicht (der Kollisionssensor *MarbleEnd* wird nicht dargestellt). Diese Situation wird in Abb. 5.20 (rechts) abgebildet. Während der Zustandsänderung wird die Methode *playSoundEnd* der Murmel aufgerufen. Der Zweck des Methodenaufrufs ist es, ein Kollisionsgeräusch über die Lautsprecher wiederzugeben. Danach befindet sich die Murmel in Zustand *Disappearing*, in dem die Deckkraft des statischen Bildes zunächst auf das Maximum gesetzt wird. Das Bild soll danach innerhalb von einer Sekunde wieder vollständig durchsichtig sein, was durch lineare Interpolation animiert wird. Sobald nach dieser Sekunde alle Animationen abgeschlossen sind, d. h., ein „completion“-Ereignis eintritt, wird der Zustandsübergang zum Terminierungsknoten durchgeführt. Dadurch wird die komplette Medienkomponenteninstanz der Murmel (inkl. Unterkomponenten) zerstört. △

Beispiel 5.14 (Aktionssequenzen in Alligator Eggs)
Abb. 5.21 zeigt den Ausschnitt eines Modells für Alligator Eggs. In der oberen linken Ecke ist die statische Struktur abgebildet. Dort sind die drei wichtigsten

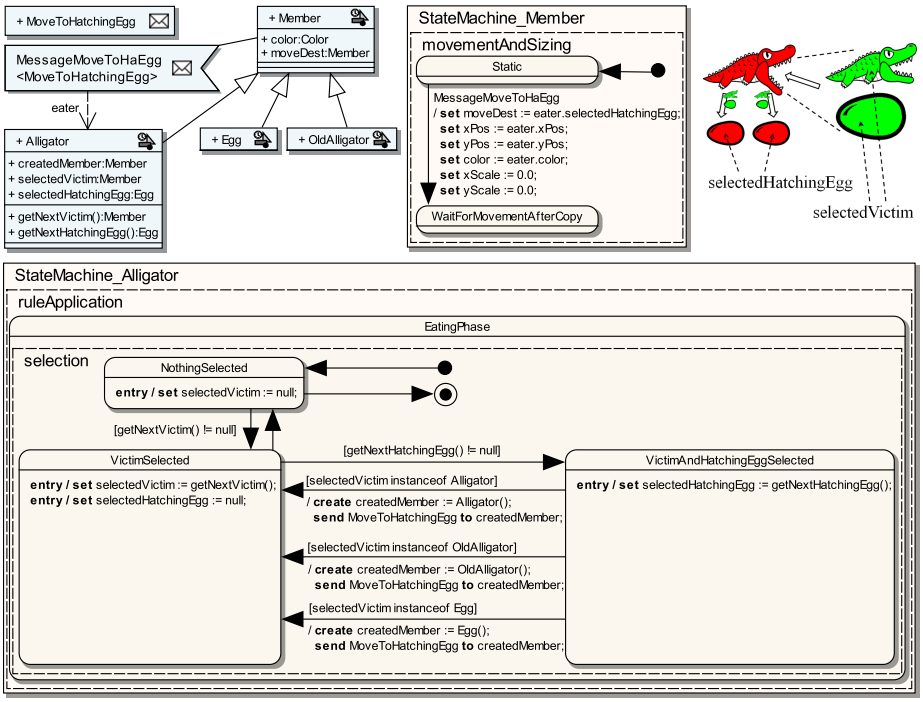


Abbildung 5.21: Instanz erzeugen und Nachrichten senden

Medienkomponenten *Alligator*, *Egg* und *OldAlligator* zusammen mit der Basismedienkomponente *Member* zu erkennen. Zudem wird für *Alligator* und *Member* jeweils ein Zustandsautomat dargestellt.

Das Beispiel soll einige der notwendigen Schritte bei der Fressregel (bzw. Beta-Reduktion) präsentieren, d. h. die Situation, bei der ein Alligator seine Beute frisst und aus jedem gleichfarbigen Ei seiner Familie eine Kopie der Beute schlüpft. Ein solcher Vorgang wird auch noch einmal in Abb. 5.21 (rechts oben) gezeigt. Bei der Animation wird zunächst jedes Elemente der Beute verkleinert und in das Maul des Alligators bewegt. Danach werden die Kopien der Beute vom sterbenden Alligator zu den entsprechenden Eiern bewegt. In Abb. 5.21 ist zu sehen, wie der dafür notwendige Kopiervorgang der Beute modelliert werden kann.

Vor Beginn der Animation des Fressvorgangs befindet sich der Alligator im Zustand *EatingPhase* von *StateMachine_Alligator*. Darin wird zunächst eines der Opfer innerhalb der Beute selektiert, wofür die Methode *getNextVictim* benutzt wird. Die Methode muss den aktuellen Systemzustand analysieren und ein Opfer wählen, für das noch keine Kopien angelegt wurden.[14] Danach wird zusätzlich

[14]Der notwendige Algorithmus soll in dem Beispiel keine tragende Rolle spielen. In dem Alligator Eggs-Modell der Sprache AML/GT, das zur Generierung des Editors letztlich verwendet wurde, kann ein einfacher Mechanismus benutzt werden.

eines der Eier gewählt, aus dem später neue Elemente schlüpfen sollen. Dies erfolgt nach einem ähnlichen Schema durch die Methode *getNextHatchingEgg*. Diese Selektionsschritte werden wiederholt, bis jede Kombination von Opfer und Ei einmal ausgewählt wurde. Zu diesem Zweck liefern die beiden Methoden nach jedem Schritt das nächste Opfer bzw. Ei und geben *null* zurück, falls jedes Element einmal zurückgegeben wurde.

Sobald eine Kombination selektiert ist (Zustand *VictimAndHatchingEggSelected*), kann das selektierte Opfer kopiert werden. Dies geschieht beim Zustandsübergang von *VictimAndHatchingEggSelected* zurück zu *VictimSelected*. Es existieren hierfür drei Transitionen, wobei die Art des selektierten Opfers entscheidet, welche gewählt wird. Das Prinzip aller Transitionen ist gleich. Zunächst wird eine neue Medienkomponenteninstanz erzeugt, wobei der verwendete Typ identisch mit dem des selektierten Opfers ist. Danach wird die Nachricht *MoveToHatchingEgg* an die neu erstellte Medienkomponenteninstanz geschickt, deren Referenz temporär in *createdMember* gespeichert wurde. Die neu erzeugte Medienkomponenteninstanz kann die Nachricht entsprechend empfangen und ändert seinen Zustand von *Static* (in *StateMachine_Member*) auf *WaitForMovementAfterCopy*. Dabei werden auch einige Attribute neu gesetzt. Die Position der neuen Medienkomponente wird bspw. auf die Position des fressenden Alligators gesetzt, der durch *eater* (der Senderinstanz der Nachricht) referenziert werden kann. Als Höhe und Breite wird allerdings noch ein Wert von 0 angegeben. Später kann die Medienkomponenteninstanz dann in Richtung des Eis bewegt werden, das durch *moveDest* referenziert wird. △

5.5 AML-Basisframework

Ein AML-*Basisframework* ist eine Sammlung von Medienkomponenten inkl. Attributen, Operationen und Animatoren, die bei der Modellierung von AML-Modellen bereits zur Verfügung stehen. Zusätzlich können darin Algorithmen zur Umsetzung von AML-Konzepten enthalten sein, z. B. das Zeichnen von Medienkomponenteninstanzen, Kollisionserkennung, etc.

Dieser Abschnitt stellt ein beispielhaftes Basisframework für AML-Modelle vor, das auch im Rest der Arbeit und für alle gezeigten Modellen verwendet wird. Es legt bspw. fest, welche grafischen Primitive verfügbar sind, wie deren Attribute Einfluss auf das Aussehen haben oder welche Animationen durch Standardanimatoren unterstützt werden. Dem Basisframework liegt eine vollständige Implementierung in Java-Code zugrunde und eine geringfügig erweiterte Version kann später v. a. im Zuge der Generierung von animierten Editoren genutzt werden (siehe Abschnitt 6.4, S. 198).

Für die Entwicklung eines Basisframeworks ist entscheidend, welche grundlegenden Arten von Medienkomponenten unterstützt werden sollen. Da für (zweidimensionale) animierte Sprachen in erster Linie Grafiken eine Rolle spielen, sind entsprechende Eigenschaften besonders wichtig. Alle Medienkomponenten sollen daher Attribute für Position, Größe, Rotationswinkel und Sichtbarkeit unterstützen. Dies wird erreicht, indem eine Basismedienkomponente (*Media-*

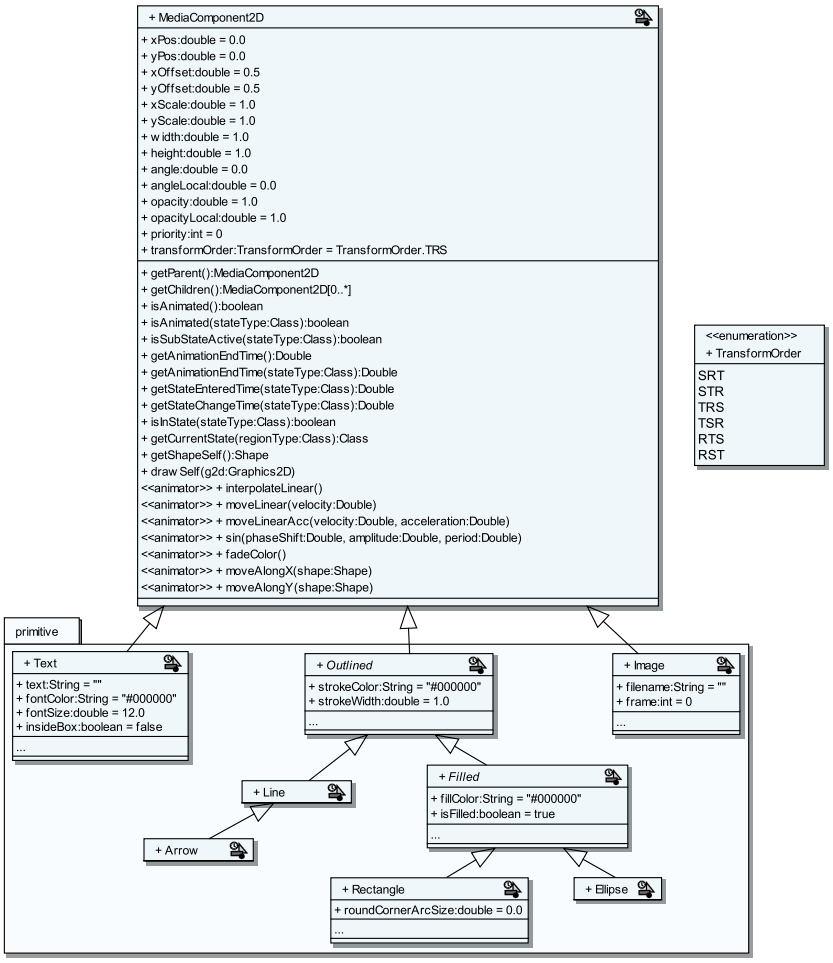


Abbildung 5.22: Medienkomponenten des AML-Basisframeworks

Component2D) mit derartigen Attributen modelliert wird. Generell wird davon ausgegangen, dass jede modellierte Medienkomponente direkt oder indirekt von *MediaComponent2D* abgeleitet wird, unabhängig davon, ob dies explizit angegeben wird oder nicht. Auf *MediaComponent2D* sollen auch Medienkomponenten für grafische Primitive, z. B. Rechteck, Text oder Linie, aufbauen. Abb. 5.22 zeigt *MediaComponent2D* und alle im Framework verfügbaren Medienkomponenten für grafische Primitive.

Verfügbare Attribute

Die in *MediaComponent2D* angegebenen Attribute werden von allen Medienkomponenten unterstützt. Sie werden in Tab. 5.6 noch einmal übersichtlich

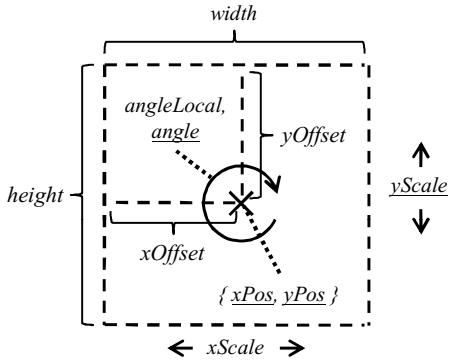


Abbildung 5.23: Bestimmung der Lage durch Basisattribute

aufgelistet. Die Position der Medienkomponente wird durch *xPos* und *yPos* bestimmt. Die Ausdehnung ist festgelegt durch *xScale* und *yScale*. Das Attribut für den Rotationswinkel heißt *angle* (in Grad). Und schließlich bestimmt das Attribut *opacity* die Deckkraft der Medienkomponente, wobei der Wert 1,0 volle Deckkraft und der Wert 0,0 Unsichtbarkeit bedeutet.

Attributname	**Transformation inkl. Unterelemente**	**Beschreibung**
xOffset	nein	Bezugspunkt (X-Achse) des Elements
yOffset	nein	Bezugspunkt (Y-Achse) des Elements
width	nein	Breite des Elements
height	nein	Höhe des Elements
angleLocal	nein	Rotationswinkel des Elements in Grad (Drehung um Bezugspunkt)
opacityLocal	nein	Deckkraft des Elements
xPos	ja	Position (X-Achse)
yPos	ja	Position (Y-Achse)
xScale	ja	Skalierung (X-Achse)
yScale	ja	Skalierung (Y-Achse)
angle	ja	Rotationswinkel (in Grad)
opacity	ja	Deckkraft
priority		Priorität bzgl. der Zeichenreihenfolge
transformOrder		Transformationsreihenfolge

Tabelle 5.6: Standardattribute

Die Darstellung in Abb. 5.23 zeigt schematisch, wie sich einige dieser Attribute auf das Zeichnen einer Medienkomponente auswirken. Darin werden auch weitere Attribute wie *width*, *height*, *xOffset* etc. dargestellt. Es werden damit ähnliche Aspekte der Grafik festgelegt, wie mit den zuvor erwähnten Attributen. Beispielsweise beeinflussen sowohl *xScale* als auch *width* die tatsächliche Breite der gezeichneten Grafik. Der wesentliche Unterschied besteht darin, dass die zuerst erwähnte Attributemenge Einfluss auf die Unterkomponenten (bzw.

inneren Merkmale) der Medienkomponente nimmt, da mit ihnen der Zeichenkontext modifiziert wird, während die zweite Attributmenge nur das Zeichnen der Medienkomponente selbst betrifft.

Als Beispiel soll erneut die Pendeluhr *PendulumClock* aus Abb. 5.1, S. 124, dienen. Wird das Gehäuse *casing* mittels *xScale* skaliert, so werden auch Ziffernblatt, Zeiger, Pendel usw. skaliert. Wird lediglich *width* von *casing* verändert, so ändert sich die Größe des Gehäuses unabhängig von den genannten Unterkomponenten.

Die Attribute, die Einfluss auf die Lage der Unterkomponenten nehmen, können technisch durch (affine) Transformationen (siehe [Mar82]) realisiert werden. Die verfügbaren Attribute ermöglichen dabei Translation (Verschiebung), Rotation (Drehung) und Skalierung (Streckung). Da es entscheidend ist, in welcher Reihenfolge diese Transformationen angewendet werden, gibt es ein zusätzliches Attribut namens *transformOrder* vom Typ *TransformOrder*, mit dem diese Reihenfolge festgelegt werden kann. Der Standardwert ist *TRS* (Translation/Rotation/Skalierung).

Ein weiteres, spezielles Attribut ist *priority*. Es legt fest, wann die entsprechende Grafik im Gegensatz zu den anderen Grafiken (in derselben Hierarchie von Medienkomponenteninstanzen) gezeichnet werden soll. Die daraus resultierende Zeichenreihenfolge bezieht sich aber ausschließlich auf das Zeichnen einer einzelnen Medienkomponente mit deren Unterkomponenten. Sie bezieht sich somit nicht darauf, in welcher Reihenfolge mehrere, hierarchisch voneinander unabhängige Medienkomponenteninstanzen gezeichnet werden.

Die grundlegende Zeichenroutine des Basisframeworks wird in einem späteren Unterabschnitt vorgestellt. Sie verdeutlicht noch einmal die Anwendung der beschriebenen Attribute.

Verfügbare Operationen

Die Basismedienkomponente *MediaComponent2D* bietet zusätzlich einige hilfreiche Operationen, die während der Laufzeit aufgerufen werden können, um verschiedene Statusinformationen abfragen zu können:

- *getParent* gibt die direkt übergeordnete Medienkomponenteninstanz zurück.
- *getChildren* gibt die direkt untergeordneten Medienkomponenteninstanzen zurück.
- *isAnimated* gibt zurück, ob eine Medienkomponenteninstanz aktuell animiert wird, d. h., mindestens ein Attribut wird animiert, wobei auch Unterkomponenten einbezogen werden.
- *isSubStateActive* gibt zurück, ob der angegebene Zustand aktiv ist und dieser auch aktive Unterzustände enthält. Finale Zustände werden dabei nicht als aktive Unterzustände betrachtet.
- *getAnimationEndTime* gibt den frühesten Zeitpunkt zurück, an dem alle Animationen der Medienkomponenteninstanz beendet sind, d. h., die letzte

aktive Animatorinstanz deaktiviert wird. Falls die Animation nie endet, wird *Double.POSITIVE_INFINITY* zurückgegeben.

- *getStateEnteredTime* gibt den Zeitpunkt zurück, wann der angegebene Zustand eingenommen wurde. Falls dieser nicht aktiv ist, wird *Double.NEGATIVE_INFINITY* zurückgegeben.
- *getStateChangeTime* gibt den Zeitpunkt zurück, wann der angegebene Zustand eingenommen wurde oder sich zuletzt ein untergeordneter Zustand von diesem geändert hat. Falls der angegebene Zustand nicht aktiv ist, wird *Double.NEGATIVE_INFINITY* zurückgegeben.
- *isInState* liefert zurück, ob sich die Medienkomponenteninstanz aktuell im angegebenen Zustand befindet.
- *getCurrentState* liefert den aktuellen Zustand der Medienkomponenteninstanz innerhalb der angegebenen Region.[15]
- *getShapeSelf* gibt den aktuellen Umriss der Medienkomponenteninstanz (ohne Unterkomponenten) zurück (Java-Klasse *Shape*). Typischerweise werden zur Bestimmung Attribute wie *xOffset*, *yOffset*, *width* etc. berücksichtigt aber nicht *xPos*, *yPos*, *xScale* etc.
- *drawSelf* ist für das Zeichnen der Medienkomponenteninstanz (ohne Unterkomponenten) verantwortlich und kann beliebig implementiert werden. Typischerweise werden zum Zeichnen Attribute wie *xOffset*, *yOffset*, *width* etc. berücksichtigt, aber nicht *xPos*, *yPos*, *xScale* etc.

Einige der Operationen benötigen den Datentyp *Class*. Dieser wird zur Über-/Rückgabe von Zuständen bzw. Regionen verwendet. Je nach Zielplattform und Umsetzung des Basisframeworks können an dieser Stelle aber unterschiedliche Techniken eingesetzt werden, um den entsprechenden Zustand oder die Region zu referenzieren. In einer Java-basierten Implementierung ist es bspw. möglich, Klassen für die modellierten Zustände und Regionen zu generieren (siehe Abschnitt 7.3.2, S. 227). In diesem Fall können *Klassenobjekte* an die Methoden übergeben (z. B. *mediaObj.isInSate(stateA.class)*) oder zurückgeben werden (z. B. *mediaObj.getCurrentState()*).

Die Operationen *isAnimated* und *getAnimationEndTime* existieren außerdem in einer zusätzlichen Version, die das entsprechende Ergebnis nicht ganzheitlich für die Medienkomponenteninstanz, sondern nur bezogen auf einen bestimmten Zustand zurückgeben. Falls der angegebene Zustand nicht aktiv ist, wird *false* bzw. *Double.NEGATIVE_INFINITY* zurückgegeben.

Die gelisteten Attribute und Operationen des Basisframeworks können bei der Modellierung erwartungsgemäß verwendet werden, z. B. auch im Rahmen von Aktionssequenzelementen. Einige der Attribute und Operationen werden aber auch in Algorithmen des Basisframeworks eingesetzt, um die durch AML

[15] Nur über die Grenzen einer Region hinweg sind mehrere aktive Zustände gleichzeitig möglich, wenn untergeordnete Zustände nicht einbezogen werden.

```
procedure drawMediaComponent(mediaComponent, drawContext)
    for prio = MIN_PRIO → MAX_PRIO do          ▷ Prioritätswerte durchlaufen
        drawMediaCompInternal(mediaComponent, drawContext, prio)
    end for
end procedure

procedure drawMediaCompInternal(mc, dc, prio)
    dc.save()                          ▷ Zeichenkontext auf Stack speichern
    dc.transform(mc.xPos, ...)         ▷ Zeichenkontext ändern, z. B. durch affine Transforma-
                                         tionen (genutzt wird dabei xPos, yPos, xScale etc.
                                         unter Berücksichtigung von transformOrder)
    if mc.priority = prio then         ▷ Nur zeichnen, falls der Prioritätswert übereinstimmt
        mc.drawSelf(dc)                ▷ Medienkomponente zeichnet nur sich selbst (genutzt
                                         wird dabei width, height, xPos etc.)
    end if
    for all smc ∈ mc.getChildren() do          ▷ Unterkomponenten mit ...
        drawMediaCompInternal(smc, dc, prio)   ▷ ... gleichem Prioritätswert zeichnen
    end for
    dc.restore()                       ▷ Zeichenkontext von Stack wiederherstellen
end procedure
```

Listing 5.1: Standard-Zeichenroutine

vorgegebenen Konzepte zu realisieren. Beispielsweise kann die Methode *getShapeSelf* die Grundlage für einen Algorithmus zur Kollisionserkennung darstellen. Die Methoden können aber auch für eigene Algorithmen verwendet werden. Mit der Methode *isAnimated* kann z. B. lokalisiert werden, wo Animationen stattfinden. Bei der Realisierung von Editoren für animierte Sprachen ermöglicht dies eine automatische Ausrichtung der Sicht auf ablaufende Animationen.

Zeichenalgorithmus

Listing 5.1 zeigt einen Algorithmus, mit dem eine Medienkomponenteninstanz unter Verwendung des vorgestellten Basisframeworks richtig gezeichnet wird. Hierfür werden die Medienkomponenteninstanz (*mediaComp*) und der Zeichenkontext (*drawContext*) an die Methode *drawMediaComponent* übergeben. Die übergebene Medienkomponenteninstanz darf dabei keine Unterkomponente sein, d. h., $mc.getParent() = null$, da Unterkomponenten zusammen mit ihren übergeordneten Medienkomponenteninstanzen gezeichnet werden.

Der Algorithmus muss die zeitabhängigen Werte der Attribute verwenden, um auch für eine animierte Visualisierung genutzt werden zu können. Außerdem beachtet der Algorithmus die Zeichenpriorität. Es werden hierfür alle möglichen Prioritätswerte (von MAX_PRIO bis MIN_PRIO) durchlaufen, um nacheinander die Medienkomponenteninstanzen der Hierarchie zu zeichnen, für die der entsprechende Prioritätswert gesetzt ist.[16] Außerdem wird der Zeichenkontext für eine konkrete Zeichenoperation (Aufruf von *drawSelf*) und den rekursiven Aufruf für Unterkomponenten jeweils modifiziert (Aufruf von *transform*). Verwendet

[16] Auf mögliche Optimierungen wird zugunsten einer einfachen Darstellung verzichtet.

werden dabei affine Transformationen unter Berücksichtigung der entsprechenden Attribute für die Lage der Medienkomponenteninstanz (*xPos*, *yPos*, *xScale* etc.) und *transformOrder* (vgl. Tab. 5.6). Um den Algorithmus kürzer fassen zu können, wird davon ausgegangen, dass auch die Deckkraft durch den Zeichenkontext bestimmt werden kann.

Mit einem ähnlichen Algorithmus kann auch die gesamte Form einer Medienkomponenteninstanz bestimmt werden. Die Berechnung erfolgt allerdings unter Verwendung von *getShapeSelf* anstatt *drawSelf*. Die Zeichenpriorität muss dabei nicht berücksichtigt werden.

Verfügbare Animatoren

Auch Standardanimatoren werden durch *MediaComponent2D* zur Verfügung gestellt. Der Animator *interpolateLinear* ist dabei der Animator für lineare Interpolation, der v. a. dann verwendet wird, falls bei einer Animationsanweisung kein Animator angegeben wird (vgl. Beispiel 5.5, S. 149). Auch der Animator *moveLinear* für gleichförmige Bewegung/Änderungen (vgl. Beispiel 5.6, S. 150) wird unterstützt. Soll zusätzlich eine konstante Beschleunigung in die Bewegung einberechnet werden, kann *moveLinearAcc* benutzt werden. Eine „sinusförmige Änderung“ wird durch *sin* angeboten. Die Merkmale der Sinuskurve können dabei festgelegt werden: Phasenverschiebung (*phaseShift*), Amplitude (*amplitude*) und Periode (*period*). Der Sinus-Animator kann bspw. genutzt werden, um die Drehbewegung des Pendels der Uhr (vgl. Abb. 5.2, S. 125) oder des Alligatormauls (vgl. Abschnitt 2.3.2, S. 42) zu realisieren. Beim Öffnen und Schließen des Mauls wird die Animation in geöffnetem bzw. geschlossenem Zustand immer langsamer, bis sie sich schließlich umkehrt. Diese Animation wird unendlich oft wiederholt.

Animator *fadeColor* ist ein Beispiel dafür, dass auch Werte animiert werden können, die nicht unmittelbar einer Zahl entsprechen. Mit diesem Animator kann eine Farbe in hexadezimalen Stringformat „RRGGBB“ (Rot/Grün/Blau) animiert werden, indem zwischen den einzelnen Basis- und Zielfarben linear interpoliert wird.

Auch komplexere Animatoren wie *moveAlongX* und *moveAlongY* werden angeboten. Als Parameter kann diesen Animatoren eine Medienkomponenteninstanz übergeben werden. Die Form der Medienkomponenteninstanz bestimmt dann den „Animationspfad“. Das bedeutet, dass der jeweilige Animator die Kontur der Medienkomponenteninstanz als Pfad interpretiert und aus dessen x- bzw. y-Koordinaten eine Wertänderungsfolge erstellt. Während der Animation werden diese Koordinatenwerte schrittweise durchlaufen und dem zeitabhängigen Wert des entsprechenden Attributs zugeordnet (inkl. linearer Interpolation zwischen zwei Punkten des Pfads).[17]

[17] Zwei Animatoren sind deshalb notwendig, da x- und y-Position einer Medienkomponente in zwei Attributen gespeichert werden (*xPos* und *yPos*). Wird eine Datenstruktur verwendet, die beide Werte enthält (oder evtl. drei Werte für dreidimensionale Daten), kann ein einziger Animator spezifiziert werden, der beide Koordinaten animiert.

Verfügbare grafische Primitive

Als grafische Primitive sind in dem vorgestellten Basisframework folgende Medienkomponenten verfügbar (vgl. Abb. 5.22, S. 176):

- *Line* und *Arrow* werden für Linien und Pfeile verwendet. Die Linie wird dabei von der oberen linken Ecke des Rechtecks, das durch die Attribute aufgespannt wird, zur unteren rechten Ecke gezeichnet, falls positive Werte für die Dimensionen verwendet werden. Im Falle des Pfeils befindet sich die Spitze in der unteren rechten Ecke. An der Linie selbst können Farbe (*strokeColor*) und Breite (*strokeWidth*) verändert werden (vgl. Beispiel 5.1, S. 135).
- *Rectangle* und *Ellipse* erlauben Rechtecke und Ellipsen. Zusätzlich zu Rahmen (vgl. Attribute von *Line*), muss bei diesen Medienkomponenten die Füllfarbe (*fillColor*) festgelegt werden, falls eine Füllung gewünscht wird (*isFilled*).
- *Text* ermöglicht grafischen Text, der beliebig positioniert werden kann. Das Attribut *text* muss dabei den anzuzeigenden Text enthalten. Als Stilmerkmale können Schriftfarbe (*fontColor*) und Schriftgröße (*fontSize*) angegeben werden. Anstatt die Größe in dieser Form anzugeben, gibt es noch einen Zeichenmodus, der mittels *insideBox* aktiviert werden kann. In diesem Modus wird der Text in das Rechteck eingepasst, das durch die Attribute aufgespannt wird.
- *Image* wird verwendet, falls Bilder gezeichnet werden sollen. Der jeweilige Dateiname wird in Attribut *filename* angegeben, wobei je nach Implementierung des Basisframeworks verschiedene Dateiformate erlaubt werden können. Beispiele sind die Formate für Rastergrafiken *JPEG* und *GIF*. Letzteres Format unterstützt auch die Animation durch mehrere Einzelbilder [Rob03]. Anstelle eines einfachen Dateinamens kann auch eine kommaseparierte Liste von Dateinamen angegeben werden. In diesem Fall gibt die Zahl *frame* an, welcher Dateiname zur aktuellen Anzeige benutzt werden soll. Der erste Dateiname wird dabei mit 0 indiziert (vgl. Beispiel 5.4, S. 141).

5.6 Verwandte Modellierungs- und Animationssprachen

Viele Modellierungssprachen, aber auch textuelle Sprachen und Paradigmen, haben das Design von AML beeinflusst. Dieser Abschnitt nennt daher einige der wichtigsten Einflüsse und verwandte Ansätze. In Bezug zu anderen Sprachen werden v. a. auch Unterschiede und Gemeinsamkeiten beschrieben, wobei UML oder Zustandsautomaten selbst nicht mehr thematisiert werden (vgl. Abschnitt 5.3, S. 127). Auf die Animationskonzepte und die Verwandtschaft zu (ggf. visuellen) Formalismen aus Abschnitt 4.4, S. 113, wird ebenfalls nicht mehr eingegangen.

Multimedia Modeling Language (MML)

AML basiert (neben der UML) auf der Arbeit für eine Sprache namens Multimedia Modeling Language (MML). MML ist eine grafische Modellierungssprache speziell für Multimedia-Anwendungen [Ple09], welche u. a. für die Modellierung von Benutzerschnittstellen (UIs) und (multi-)medialen Inhalten genutzt werden kann.

Das wesentliche Konzept, das AML übernommen hat, sind Medienkomponenten. MML unterstützt diese in ähnlicher Weise. Ein Unterschied ist allerdings, dass MML lediglich die Modellierung der Struktur, nicht aber eine detaillierte Spezifikation von Inhalten vorsieht. Aus MML-Modellen sollen lediglich Codegerüste für verschiedene Plattformen erstellt werden, die anschließend vervollständigt werden können. Dieser Mechanismus wird zusätzlich unterstützt durch sogenannte Medienartefakte (engl. media artifacts), welche MML-Medienkomponenten „manifestieren“, d. h., verschiedene Ausprägungen einer MML-Medienkomponente darstellen. Im Gegensatz dazu können AML-Medienkomponenten (und auch das AML-Modell selbst) – je nach Präferenz oder Bedarf – bereits sehr detailliert ausgearbeitet werden, d. h., das Aussehen einer AML-Medienkomponente kann exakt modelliert werden. Das Konzept von Medienartefakten wird dabei nicht unterstützt. Andererseits ist es durch die exakten Angaben in AML möglich, Animationen präzise zu modellieren, was in MML nicht (direkt) unterstützt wird. Ein weiterer Unterschied ist, dass MML-Medienkomponenten auch Medientypen wie Video und Audio unterstützen. AML-Medienkomponenten wurden bisher nur im Zusammenhang mit zweidimensionalen Repräsentationsformen verwendet.[18]

In MML kommen vier Diagrammarten bzw. Modelle zum Einsatz: Strukturmodell (engl. structural model), Szenenmodell (engl. scene model), Präsentationsmodell (engl. presentation model) und Interaktionsmodell (engl. interaction model). Während im Strukturmodell Typen (bspw. auch Medienkomponenten) und mögliche Beziehungen zwischen diesen festgelegt werden, sind die Ideen der anderen Modellteile sehr spezifisch und wurden auch nicht auf AML übertragen.

Im Szenenmodell können unterschiedliche „Szenen“ einer multimedialen Anwendung festgelegt werden. Beispiele für Szenen wären in einem Spiel – eine typische Multimedia-Anwendung – das Hauptmenü, die Hilfeseiten, die Highscore-Liste und die unterschiedlichen Level. Solche Szenen haben in den Diagrammen die Form von UML-Zuständen und werden auch durch Transitionen miteinander verknüpft, d. h., es wird festgelegt, wie die Szenen zusammenhängen und wie durch die Applikation navigiert werden kann. Im Präsentationsmodell kann danach die UI für jede Szene bestimmt werden. Dies geschieht durch die Spezifikation, wie die einzelnen Elemente bspw. durch Medienkomponenteninstanzen repräsentiert werden. Im Interaktionsmodell kann abschließend das Verhalten jeder Szene modelliert werden, v. a. im Hinblick auf Benutzerinteraktion. Die Modellierung basiert dabei auf UML-Aktivitätsdiagrammen. Darin werden u. a. auch Sensoren eingesetzt, die vorher für die Medienkomponenten spezifiziert werden können. Dies ist ein Konzept, das von AML aufgegriffen wurde, wobei

[18] Das Stereotyp-Icon für AML-Medienkomponenten ist von MML-Medienkomponenten mit Medientyp *Animation2D* abgeleitet.

sich die Sensoren von AML und MML aber bzgl. Metamodell, verfügbarer Arten und Eigenschaften deutlich unterscheiden.

Die Diagrammarten zeigen auch, dass MML in erster Linie dazu benutzt werden soll, um vollständige Multimedia-Applikationen zu modellieren. AML fokussiert hingegen die Modellierung von Medienkomponenten, deren Verhalten und Animation. Der Kontext, in dem diese Medienkomponenten eingebettet werden, spielt eine untergeordnete Rolle. Dadurch eignet sich AML im Gegensatz zu MML zur Spezifikation interaktiver animierter Sprachen.

Scene Structure and Integration Modelling Language (SSIML)

Die Sprache Scene Structure and Integration Modelling Language (SSIML) [Vit05] war ein weiterer Einflussfaktor für die Entwicklung der Konzepte von AML. SSIML ist eine Sprachfamilie, welche die strukturelle Beschreibung von 3D-Objekten, aufgaben-zentrierter 3D-Anwendungen (inkl. Interaktion und UIs) und die Spezifikation von 3D-Szenen erlaubt. Insofern ist SSIML auf 3D-Inhalte spezialisiert. Obwohl in AML theoretisch auch 3D-Objekte modelliert werden können, umfasst AML keine typische Konzepte, die üblicherweise in 3D-Umgebungen Anwendung finden und durch SSIML unterstützt werden: Kameraposition, Lichtquellen etc.

Den wichtigsten Einfluss nahm die Sprache SSIML/Behaviour. Mit dieser Sprache kann das Verhalten und die Animation von 3D-Objekten modelliert werden. Die Grundlage der Sprache bilden UML-Zustandsdiagramme. Neben gewöhnlichen Zuständen werden darin Animationszustände genutzt, die eine ähnliche Semantik wie AML-Animationszustände besitzen. Anstatt Animationsanweisungen zu nutzen, werden Animationszustände in SSIML/Behaviour weiter spezialisiert. So gibt es bspw. Animationszustände für sogenannte „key frame“-Animationen (Angabe von Schlüsselwerten, zwischen denen während der Animation interpoliert wird) oder physikalische Animationen (Angabe von Kräften, die auf das Objekt wirken). Zudem werden Sensoren, insbesondere Kollisionssensoren, in ähnlicher Weise genutzt wie in AML. Sie werden für 3D-Objekte modelliert und melden danach bestimmte Ereignisse in Bezug auf das 3D-Objekt.

Petri-Netz-basierte Ansätze

Es existieren ebenfalls Ansätze, Animationen formal mittels Petri-Netzen (vgl. Abschnitt 2.3.1, S. 32) zu beschreiben. Beispiele hierfür sind [MRR98] und [MPB12]. Meistens wird in den Petri-Netzen aber nur der Kontrollfluss der Animation modelliert, z. B. in [MRR98].

In [MPB12] werden Funktionen zur Neuberechnung animierter Werte genutzt, was an die Vorgehensweise mittels Animatoren erinnert. Allerdings wird in den (speziellen) Petri-Netzen zusätzlich die komplette Animationssemantik modelliert, z. B. wann eine Neuberechnung stattfindet, wie Attribute mit den neu berechneten

Werten gesetzt werden etc. Deshalb wirken die Modelle selbst für einfache Animationen sehr umfangreich.

Weitere visuelle Modellierungssprachen und Umgebungen

Object-Oriented Modeling of MultiMedia Applications (OMMMA) [SE99] bietet eine plattformunabhängige, UML-basierte Sprache (OMMMA-L) zur Modellierung von multimedialen Anwendungen. Diese Sprache kann in gewissen Aspekten mit MML verglichen werden. Bei OMMMA werden neben Klassendiagrammen und Zustandsdiagrammen für die Spezifikation des Systemverhaltens auch Präsentationsdiagramme (engl. presentation diagrams) verwendet, in denen die UI spezifiziert wird. Dabei wird die räumliche Anordnung der UI-Elemente in den Diagrammen teilweise widergespiegelt. Auch UML-Sequenzdiagramme können zur Modellierung eines vorgegebenen zeitlichen Verhaltens benutzt werden.

In [KDV07] wird ein Ansatz präsentiert, der geeignet ist, um die künstliche Intelligenz in Computerspielen zu modellieren. Als konkretes Beispiel wird das Verhalten eines Panzers in einem Computerspiel mit einer Variante von Rhapsody-Zustandsautomaten (siehe [HK04]) modelliert. Vor allem die Verwendung von Sensoren, die darin ein wichtiges Element zur Erzeugung von Ereignissen sind, wird darin herausgestellt.

Whizz'Ed [ECP95] ist eine grafische Programmierumgebung zur Erstellung interaktiver, animierter Anwendungen. Darin kann u. a. das animierte Verhalten von grafischen Objekten mittels Flussgraphen modelliert werden (siehe Abb. 5.24). Die Knoten der Graphen bilden anpassbare Kacheln, welche bestimmte Aufgaben erfüllen, z. B. die Rotation des damit verknüpften Objekts. Neben Kacheln für Animation existieren aber auch Kacheln, die Benutzeraktionen überwachen (z. B. Mausklicks auf das Objekt) und daher ebenfalls Sensoren genannt werden.

Eine weitere visuelle Sprache zur Modellierung von Animationen ist HandMove [Vod97]. Eine Besonderheit ist, dass darin Trajektorien so gezeichnet werden können, wie sie von animierten Objekten befolgt werden sollen (siehe Abb. 5.25). An verschiedenen Punkten der Trajektorie können zusätzliche Animationsanweisungen in Form von grafischen Blöcken (auch inkl. Formel) notiert werden, um z. B. das Objekt um seine Achse rotieren zu lassen etc.

An dieser Stelle sollten auch professionelle Entwicklungsumgebungen für multimediale Anwendungen kurz erwähnt werden. Auch darin kann oftmals die Komposition von grafischen Objekten und deren Animation spezifiziert werden. Wichtige Tools aus dieser Kategorie sind bspw. Adobe Flash Professional [Fla12] oder Expression Blend [Exp12]. Darin können grafische Objekte zunächst gezeichnet und strukturiert werden. Danach lassen sich Animationen durch zahlreiche Möglichkeiten spezifizieren, z. B. mittels „Timeline“, durch verschiedener Editore (z. B. dem „Motion Editor“ in Adobe Flash Professional) oder auch direkt durch Code. In den entsprechenden Modi erinnern die Entwicklungsumgebungen auch an reine Bildbearbeitungs- bzw. Animations-Software.

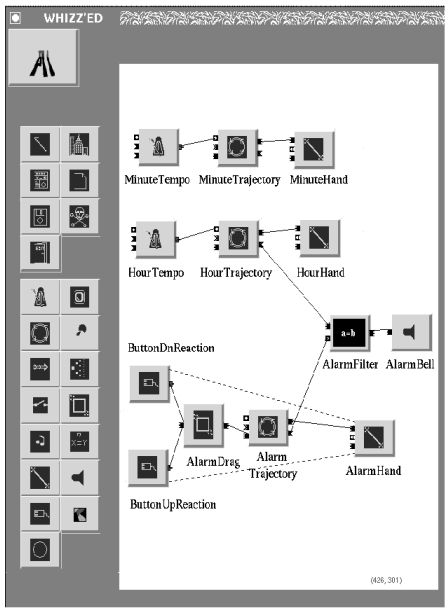


Abbildung 5.24: Whizz'Ed – Flussgraph (aus [ECP95])

Textuelle Sprachen

Neben visuellen Sprachen und Entwicklungsumgebungen gibt es auch textuelle Sprachen zur Modellierung oder Spezifikation von Animationen. Als erstes Beispiel soll die Sprache JavaFX Script [Ste09] dienen. Diese DSL wurde für das JavaFX-Framework [Jav12] (vor Version 2.0) eingesetzt, um JavaFX-Applikationen zu erstellen. Die Syntax von JavaFX Script erinnert an Java-Programme, bietet jedoch viele zusätzliche Konstrukte, die v. a. den deklarativen Anteil der Sprache erhöhen. Auch Animationen können darin deklarativ festgelegt werden. Listing 5.2 zeigt den Ausschnitt eines JavaFX-Scripts. Er soll veranschaulichen, wie Animationen darin programmiert werden können: es werden Werte (*values*) bzw. „key frames“ festgelegt, die zu bestimmten Zeitpunkten (*time*) während der Animation angenommen werden sollen. Wie sich die Werte zwischen den „key frames“ verhalten, kann mit dem Schlüsselwort *tween* und sogenannten Interpolatoren bestimmt werden. Später kann eine derartige Animationsspezifikation an die Attribute von grafischen Objekten mit dem Schlüsselwort *bind* „gebunden“ werden, d. h., die Attribute werden animiert (vgl. Abschnitt 5.4.3, S. 143). Viele dieser Konstrukte können mit Animationsanweisungen und Animatoren von AML verglichen werden.

Die Extensible Application Markup Language (XAML) ist eine Beschreibungssprache, die u. a. für Microsofts Silverlight eingesetzt wird. Sie wird hauptsächlich für die Gestaltung von GUIs genutzt und unterstützt auch einige Tags für Animationen [Mor08]. Beispielsweise kann das Tag „DoubleAnimation“ verwendet werden, um ein *double*-Attribut (bspw. die x-Position eines Elements)

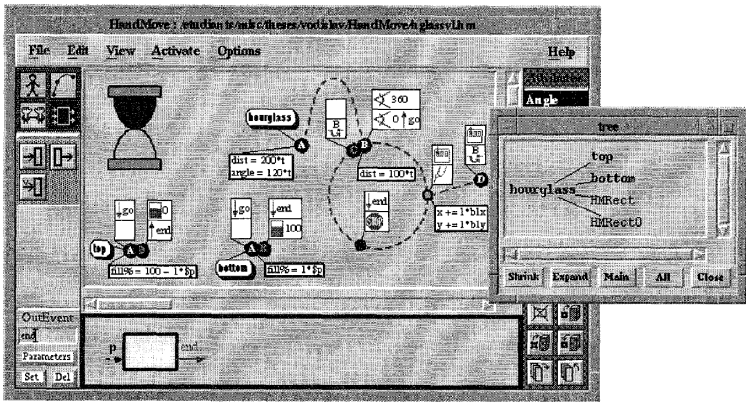


Abbildung 5.25: HandMove – Animationen inkl. Trajektorie (aus [Vod97])

über einen bestimmten Zeitraum zu verändern. Für eine derartig spezifizierte Animation können außerdem Eigenschaften gesetzt werden, wie *RepeatBehavior*, *AutoReverse* etc., die mit Animatormodifikationen verglichen werden können (vgl. Tab. 5.3, S. 153). Üblicherweise werden XAML-Dateien aber nicht von Hand erstellt, sondern es werden grafische GUI-Designer wie Expression Blend benutzt.

Auch die Virtual Reality Modeling Language (VRML) ist eine Beschreibungssprache, die (3D-)Animationen unterstützt [TRMR97]. Neben Animationen können sogar Interaktionsmöglichkeiten mit der virtuellen Welt spezifiziert werden, z. B. durch Sensoren.

Agentenbasierte Modellierung

Im Gegensatz zu vielen der bisher betrachteten Modellierungssprachen, zielt AML nicht auf die Modellierung kompletter Softwaresysteme oder UIs ab. Im Fokus stehen Medienkomponenten, deren Verhalten und Animationen modelliert werden. Häufig ist es für die Modellierung sinnvoll, das individuelle Verhalten von Medienkomponenten zu analysieren. Bei der Modellierung eines komplexen Systems wie Traffic (vgl. Abschnitt 2.3.4, S. 45) fällt dies stärker auf als bei der Modellierung einer Sprache wie Alligator Eggs. Während für Traffic Autos modelliert werden, die eigene Entscheidungen treffen und autonom handeln müssen, werden für Alligator Eggs einzelne Algorithmen und Phasen modelliert, welche die Menge der Diagrammelemente analysieren und steuern.

Systeme mit (meist vielen) eigenständigen Komponenten sind Schwerpunkt der sogenannten agentenbasierten Modellierung und Simulation (siehe [Gil07]). Das Systemverhalten ist darin durch das Verhalten einzelner Agenten gegeben, die autonom im System handeln und miteinander interagieren können. AML eignet sich zur Modellierung von agentenbasierten Systemen, wobei die Möglichkeiten ggü. spezialisierten Sprachen (meist DSLs) aber limitiert sind. Im Gegensatz zu typischen agentenbasierten Modellierungssprachen, z. B. einer Vari-

```
var durationMs : Number = 2000;
var xVal : Number = 50;
var yVal : Number = 50;
var anim = Timeline {
  keyFrames: [
    KeyFrame {
      time: 0ms
      values: [ xPos => 50, yPos => 50 ] },
    KeyFrame {
      time: bind 4000ms - (durationMs * 1ms)
      values: [
        xPos => endxPos tween Interpolator.LINEAR,
        yPos => endyPos tween Interpolator.LINEAR ] } ]
  repeatCount: Timeline.INDEFINITE
};
/* ... */
Line {
  startX: bind xPos
  startY: bind yPos
  endX: 200
  endY: 400 }
```

Listing 5.2: JavaFX-Beispiel

ante von Aktivitätsdiagrammen im SeSAm-System [KHF06], erlaubt AML dafür die Spezifikation der grafischen Repräsentation und Animation der Agenten.

In diesem Umfeld existieren neben (einfachen) visuellen Sprachen oder auch textuellen Programmiersprachen (inkl. Bibliotheken und Frameworks für agentenbasierte Programmierung) meist vollständige Entwicklungsumgebungen zur Erstellung von agentenbasierten Systemen inkl. UI. Beispiele hierfür sind AgentSheets (vgl. Abschnitt 2.2, S. 26), Repast Simphony [NTCO07] oder NetLogo [Sak12, Wil12]

Kapitel 6

AML for Graph Transformations (AML/GT)

Bisher wurden zwei verschiedene Ansätze zur Spezifikation von interaktiven animierten Sprachen vorgestellt. Zum einen wurde ein Ansatz beschrieben, der auf Graphen und GTRs basiert (Kapitel 3 und Kapitel 4). Zum anderen kann auch eine Modellierungssprache wie AML eingesetzt werden, um Teile einer Sprachspezifikation unter Verwendung von Klassen- und Zustandsdiagrammen zu modellieren (Kapitel 5). Beide Möglichkeiten bieten Vor- und Nachteile, was unweigerlich zur Frage führt, ob diese Ansätze kombiniert werden können. Daher wird in diesem Kapitel eine Modellierungssprache namens AML for Graph Transformations (AML/GT) präsentiert, die eine Erweiterung von AML darstellt und viele Vorteile beider Ansätze vereint.

Zunächst werden in diesem Kapitel noch einmal ausführlich Hintergrund und Ziele der Spracherweiterung bzw. der Modellierungssprache AML/GT beschrieben (Abschnitt 6.1). Auch die technische Umsetzung der Spracherweiterung wird skizziert (Abschnitt 6.2). Danach werden spezifische Konstrukte und Besonderheiten von AML/GT behandelt (Abschnitt 6.3 bis Abschnitt 6.6).

6.1 Motivation

Wie in der Einleitung dieses Kapitels erwähnt, soll AML/GT die Vorteile der Ansätze aus Kapitel 4 und Kapitel 5 miteinander kombinieren. Zur besseren Lesbarkeit sollen für diese beiden Ansätze die Begriffe „AAS/GT-Ansatz“ und „AML-Ansatz“ verwendet werden.

In Kapitel 3 wird gezeigt, wie Hypergraphen Diagramme repräsentieren können. Auf dieser Basis ermöglicht der in Kapitel 4 beschriebene AAS/GT-Ansatz die Repräsentation von animierten Diagrammen. Da auf externe und

interne Ereignisse reagiert werden kann, ist auch die Umsetzung von Editoren für interaktive animierte Sprachen möglich.

Bei Verwendung des AAS/GT-Ansatzes müssen u. a. die Diagrammkomponenten in Form von Komponentenhyperkanten (inkl. Attribute und Aussehen) und die Sprachdynamik in Form von Ereignis-GTRs festgelegt werden. Unabhängig von einem konkreten Meta-Tool und ohne Erweiterung des AAS/GT-Ansatzes bedeutet dies in der Regel, dass Listen von Diagrammkomponenten und Ereignis-GTRs spezifiziert werden.

Auch wenn die Spezifikation einzelner GTRs in vielen Fällen intuitiv erfolgen kann, so ist es schwierig, mehrere Ereignis-GTRs in einem größeren Zusammenhang zu betrachten, zu verstehen, sie entsprechend abzustimmen und konsistent zu erstellen. Neben Schwierigkeiten bei der Planung ist auch die Übersichtlichkeit und Dokumentation in späteren Phasen mangelhaft, was wiederum zu Problemen bei Fehlerbehebung oder Spracherweiterung führen kann. Dieser Nachteil wurde bereits bei der Umsetzung einfacher Sprachen aus Abschnitt 2.3, S. 30, festgestellt. Eine Sprache wie Traffic mit dem AAS/GT-Ansatz ohne zusätzliche Hilfsmittel zur Planung umzusetzen, ist nur schlecht möglich (vgl. Abschnitt 2.3.4, S. 45).

Andererseits bietet sich auch die in Kapitel 5 beschriebene Modellierungssprache AML an, um Teilaspekte einer interaktiven animierten Sprache zu modellieren. Mit diesem AML-Ansatz fehlen für eine Sprachspezifikation allerdings wichtige Details, zumindest wenn Editoren generiert werden sollen und DiaMeta-Editoren als Maßstab gesehen werden. Dies ist vor allem darauf zurückzuführen, dass keine Graphen als Repräsentationsform für entsprechende Diagramme verwendet werden. So bieten Graphen bspw. die Möglichkeit, Reduktionsregeln zur Ableitung der abstrakten Syntax zu spezifizieren, was u. a. die Prüfung der Syntax ermöglicht oder die Erstellung von Layoutalgorithmen vereinfacht (vgl. Abschnitt 3.3, S. 88).

AML bietet mit Zustandsautomaten allerdings ein höheres strukturelles Konzept, was eine bessere Planung zusammenhängender Abläufe ermöglicht. Auch die Art und Weise, wie Struktur, Aussehen und Animationen von Komponenten spezifiziert werden, ist in AML genau festgelegt. Der AAS/GT-Ansatz und die darin sehr abstrakt definierte Darstellungsfunktion legen diesbezüglich nichts Genaueres fest. Daher können diese Aspekte in der konkreten Sprachspezifikation für ein bestimmtes Meta-Tool beliebig umgesetzt werden. Der DiaMeta-Designer wurde bspw. nicht erweitert, um Animationen direkt spezifizieren zu können, weshalb die Animationen nur mit angemessenen Kenntnissen über das DiaMeta-Framework umständlich ausprogrammiert werden können.

Für die Umsetzung von Traffic wurde AML erfolgreich zur Planung eingesetzt [SMPV10]. Danach wurde das Modell auf den AAS/GT-Ansatz übertragen und eine entsprechende (konkrete) Sprachspezifikation für DiaMeta händisch umgesetzt, was noch einmal sehr aufwändig war. Die Übersetzung in eine solche Sprachspezifikation kann jedoch automatisch durchgeführt werden, falls im AML-Modell genügend Informationen untergebracht werden, die eine derartige Übersetzung ermöglichen. Da die notwendigen Zusatzinformationen bestimmte Aspekte des AAS/GT-Ansatzes widerspiegeln, kann man hierbei von einer Kombination der beiden Ansätze sprechen.

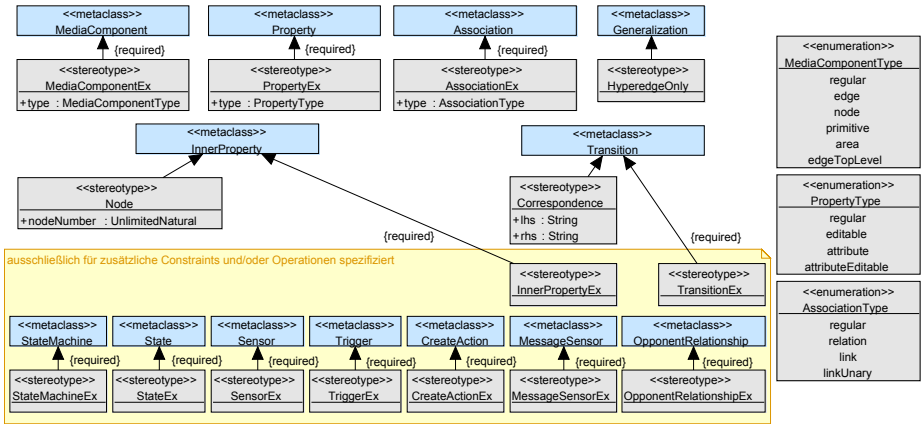


Abbildung 6.1: AML/GT-Profil

Eine solche Kombination war eines der Ziele dieser Arbeit, um die Vorteile beider Ansätze nutzen zu können. Entstanden ist dabei eine Sprache namens AML for Graph Transformations (AML/GT). Sie basiert vollständig auf AML und erweitert die Modellierungssprache um zusätzliche Konstrukte, welche die automatische Generierung konkreter Sprachspezifikationen für interaktive animierte Sprachen auf Basis von Hypergraphen ermöglichen.

6.2 Erweiterung von AML

Während AML eine Erweiterung von UML durch den sogenannten „middleweight extension“-Mechanismus darstellt (vgl. Abschnitt 5.3, S. 127), wird für die Erweiterung von AML durch AML/GT der „lightweight extension“-Mechanismus (vgl. [BH08]) genutzt. Dieser weniger aufwändige Mechanismus ist in diesem Fall vorteilhaft, da AML/GT keine neuen Sprachelemente definiert, sondern lediglich benutzt wird, um vorhandene Elemente detaillierter zu spezifizieren.

AML/GT wurde somit mittels UML-Profil realisiert. Die genaue Technik hierfür wird u. a. in [UML11] erläutert. Abb. 6.1 zeigt alle Stereotypen des Profils. Es wird dabei zwischen zwei Arten von Stereotypen unterschieden. Die eine Art umfasst die Stereotypen *Node*, *Correspondence* und *HyperedgeOnly*. Sie können optional einem entsprechenden Element zugeordnet werden. Die andere Art trägt das Suffix “Ex” (für engl. extension). Diese Stereotypen werden den Elementen obligatorisch zugeordnet, was auch durch die Eigenschaft *required* erkennbar ist. So wird bspw. jeder neu erzeugten Medienkomponente (*MediaComponent*) automatisch der Stereotyp *MediaComponentEx* hinzugefügt, der auch nicht entfernt werden kann. Dadurch ist es möglich, jedem passenden UML- oder AML-Element neue Zusicherungen, Methoden oder Attribute zuzuordnen. Letztere müssen jedoch über den angewendeten Stereotypen referenziert werden. Die

verfügbaren Operationen werden in Listing B.3, S. 278ff, aufgeführt und kurz beschrieben.

Einschränkungen durch das AML/GT-Profil

In AML/GT werden einige Modellierungsmöglichkeiten von UML und AML durch Zusicherungen innerhalb des AML/GT-Profils eingeschränkt. Dieser Abschnitt nennt einige dieser Einschränkungen. Die Begründung der Einschränkungen ist allerdings erst in den folgenden Abschnitten möglich, wenn die spezifischen Elemente von AML/GT erklärt werden. Die genauen Definitionen (inkl. weiterer Einschränkungen innerhalb von AML/GT) befinden sich in Listing B.4, S. 279ff.

Es folgt eine Kurzbeschreibung wichtiger Einschränkungen. Zum Verständnis dieser sollte zuvor erwähnt werden, dass es in AML/GT verschiedene Arten von Medienkomponenten gibt. Eine Art wird *Hyperkanten-Medienkomponente* genannt (siehe Abschnitt 6.3, S. 193).

- Inneren Merkmalen muss eine Multiplizität von 1 zugeordnet werden (siehe Listing B.4/14).
- Zustandsmaschinen dürfen nur für Hyperkanten-Medienkomponenten spezifiziert werden (siehe Listing B.4/79).
- Lediglich Hyperkanten-Medienkomponenten dürfen als Sensoreigentümer festgelegt werden (siehe Listing B.4/65).
- Lediglich Hyperkanten-Medienkomponenten dürfen als Verursacher festgelegt werden (siehe Listing B.4/70).
- Bei Nachrichtensensoren darf maximal ein Verursacher festgelegt werden (siehe Listing B.4/74). Falls dieser spezifiziert wird, darf eine entsprechende Nachricht nur von einer Medienkomponenteninstanz gesendet werden, deren Typ dem Verursacher entspricht (oder davon abgeleitet wird).
- Hyperkanten-Medienkomponenten dürfen lediglich einen parameterlosen Konstruktor (engl. default constructor) besitzen.
- Erstellte Medienkomponenten müssen Hyperkanten-Medienkomponenten sein (siehe Listing B.4/88). Es gelten dabei spezielle Regeln für *Create-Action*-Elemente (siehe Abschnitt 6.5.4, S. 209).

6.3 Spezifikation von Hyperkanten

Wie in Kapitel 3 beschrieben, können Diagramme intern durch (attributierte) Hypergraphen repräsentiert werden. Jede Diagrammkomponente entspricht dabei einer Komponentenhyperkante inkl. Knoten als Konnektoren. Beziehungen zwischen Diagrammkomponenten werden durch Verbindungshyperkanten repräsentiert. Sowohl Diagrammkomponenten als auch Beziehungen zwischen diesen Diagrammkomponenten können im statischen Teil eines AML-Modells

(Klassenmodell) modelliert werden. Es ist daher naheliegend, diesen Teil als Spezifikationsgrundlage für Komponentenhyperkanten (inkl. Knoten) und Verbindungshyperkanten zu nutzen.

Wichtig wird in diesem Zusammenhang das Attribut *type* der Stereotypen *MediaComponentEx* (bei Medienkomponenten) und *AssociationEx* (bei Assoziationen). Es hält fest, um welche *Art von Medienkomponente bzw. Assoziation* es sich handelt. Spezielle Arten, d. h., wenn *type* nicht auf *regular* gesetzt wird, werden in den nächsten Abschnitten vorgestellt.

Spezifikation von Komponentenhyperkanten und Knoten

Mittels AML-Medienkomponenten können die in einer Sprache verfügbaren Diagrammkomponenten modelliert werden.[1] AML/GT ermöglicht daher die Einordnung einer Medienkomponente als Diagrammkomponente, was zusätzlich mit der Spezifikation als Komponentenhyperkante gleichzusetzen ist. Die Attribute der Medienkomponente können dann als Vorlage für Hyperkantenattribute dienen (siehe Abschnitt 6.6, S. 210).

Um eine Medienkomponente als Diagrammkomponente und damit als Komponentenhyperkante zu spezifizieren, muss das Attribut *type* auf *edge* gesetzt werden. Eine solche Medienkomponente wird im Folgenden auch *Hyperkanten-Medienkomponente* genannt. Im AML/GT-Diagramm wird eine Hyperkanten-Medienkomponente mit einem blauen Rechteck in der oberen rechten Ecke markiert.

Diagrammkomponenten müssen mindestens einen Konnektor besitzen. Im internen Hypergraphen entsprechen Konnektoren den Knoten einer Komponentenhyperkante (vgl. Abschnitt 3.1.3, S. 55). Obwohl Konnektoren nicht visuell im Diagramm dargestellt werden, benötigen sie in vielen Fällen typische Eigenschaften von Medienkomponenten wie Position, Größe etc. Ein Konnektor kann daher ebenfalls als Medienkomponente – normalerweise ohne visuelle Darstellung – modelliert werden. Sie wird in diesem Fall auch *Knoten-Medienkomponente* genannt und ihr Attribut *type* muss auf *node* gesetzt werden. In der oberen rechten Ecke der Medienkomponente wird dann ein grüner Kreis dargestellt.

Es ist in AML üblich, bestimmte Medienkomponenten, die Teil von übergeordneten Medienkomponenten sind, als Typen von inneren Merkmalen einzusetzen. Da auch Knoten zu einer Komponentenhyperkante gehören, werden Knoten-Medienkomponenten als Typen für innere Merkmale einer Hyperkanten-Medienkomponente verwendet.[2] Dies ist gleichzusetzen mit der Spezifikation eines Knotens für die Komponentenhyperkante. Es ist allerdings die zusätzliche Angabe erforderlich, welchem Knoten das innere Merkmal entspricht. Das innere Merkmal wird daher mit Stereotyp *Node* versehen. Dessen Attribut *nodeNumber* erhält dabei die Nummer des Tentakels, der den damit spezifizierten Knoten besucht. Dies wird im AML/GT-Diagramm dargestellt, indem das innere Merkmal

[1] Im Zusammenhang mit generierten Editoren sind die Diagrammkomponenten gemeint, die in einer Werkzeugpalette angewählt und danach im Diagramm platziert werden können.

[2] Die Ebene in der Hierarchie der inneren Merkmale spielt dabei keine Rolle.

durch einen grünen Kreis mit entsprechender Nummer gezeichnet wird.[3] Jede Hyperkanten-Medienkomponente muss mindestens ein solches inneres Merkmal enthalten, da für jede Komponentenhyperkante auch mindestens ein Knoten spezifiziert werden muss.

Für die Spezifikation von Hyperkanten und Knoten können im AML/GT-Modell auch Vererbungsmechanismen eingesetzt werden, z. B. um die Hierarchie von Komponentenhyperkanten festzulegen. Eine abstrakte Hyperkanten-Medienkomponente kann ebenfalls modelliert werden. Allerdings stellt diese keine Diagrammkomponente der Sprache, sondern nur ein Grundgerüst einer solchen dar. Insbesondere bei der Spezifikation von Knoten können die Besonderheiten der Vererbung von inneren Merkmalen oder Neudefinition genutzt werden (vgl. Abschnitt 5.4.2, S. 137). Derartige Möglichkeiten werden auch in Beispiel 6.9, S. 212, für die Sprache Alligator Eggs gezeigt.

Beispiel 6.1 (Hyperkantenspezifikation für ein Avalanche-Bauteil)
Abb. 6.2 (unten) zeigt einen Auszug des AML/GT-Diagramms für Avalanche. Darin wird die gerade Bahn inkl. Konnektoren (oben links) als Medienkomponente mit dem Namen *Straight* modelliert (unten). Die Medienkomponente wird daher als Komponentenhyperkante markiert. Neben dem inneren Merkmal *bodyStraight*, welches die grafische Darstellung der Gerade in Form eines Bildes modelliert, besitzt *Straight* die inneren Merkmale *top* und *bot*. Diese modellieren die beiden Konnektoren der Geraden, weshalb die inneren Merkmale durch die Typen *NTop* und *NBot* – jeweils Knoten-Medienkomponenten – auch entsprechend als Knoten markiert sind. Die inneren Merkmale selbst tragen zudem Informationen über den Tentakel der Komponentenhyperkante. Der Tentakel 0 besucht den Knoten, der durch *top* spezifiziert wird, während der Tentakel 1 den Knoten besucht, der durch *bot* spezifiziert wird. Abb. 6.2 (oben rechts) zeigt die so spezifizierte Komponentenhyperkante *Straight* mit einigen Attributen (siehe *MediaComponent2D*) und ihren zwei Knoten. Das Beispiel zeigt ebenfalls, wie die Bereiche der Konnektoren durch *NTop* und *NBot* festgelegt werden. Details werden allerdings erst in Abschnitt 6.4, S. 197, erläutert. △

Top-Level-Komponentenhyperkante

Als Besonderheit erlaubt AML/GT auch die Spezifikation von sogenannten *Top-Level-Komponentenhyperkanten*. Als *type* einer Medienkomponente wird in diesem Fall *edgeTopLevel* gewählt. Im Diagramm wird dies durch ein blaues Rechteck mit einem „T“ im Inneren dargestellt.

Eine derartige Medienkomponente modelliert eine Diagrammkomponente, die in jedem Diagramm der spezifizierten Sprache immer und exakt einmal vorhanden sein muss. Gleiches gilt für die zugehörige Komponentenhyperkante im KSG. Meist ist die Diagrammkomponente unsichtbar, und für einen Editornutzer ist sie auch nicht relevant. Sie wird oft nur für interne Zwecke verwendet. Ein Beispiel dafür ist Komponentenhyperkante *Root* aus Alligator Eggs (siehe Abb. A.2,

[3] In allen Beispielen wird außerdem das Präfix „n“ vorangestellt, wodurch gleichzeitig der „Standardbezeichner“ des jeweiligen Knotens dargestellt wird.

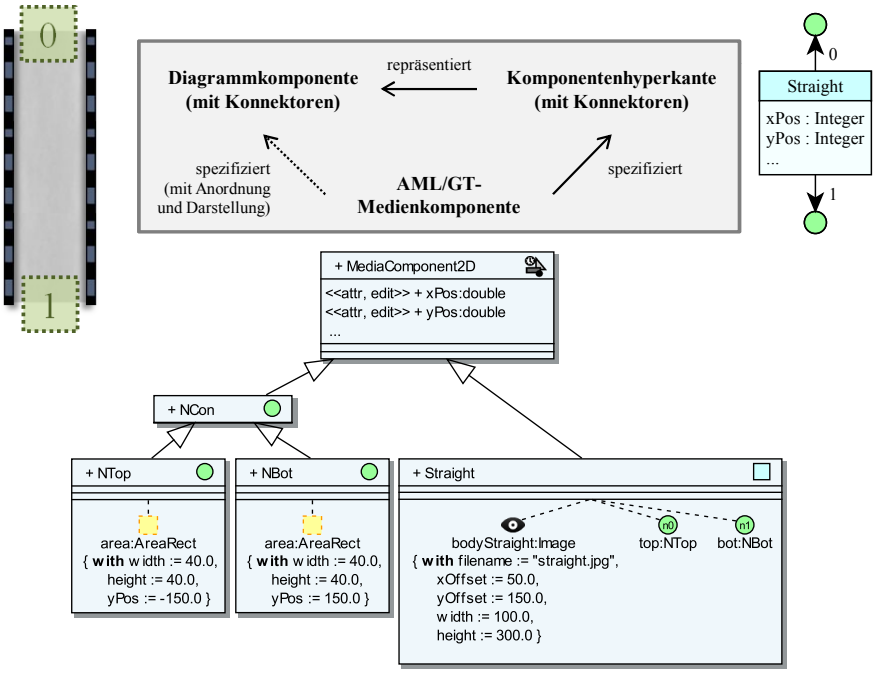


Abbildung 6.2: Hyperkantenspezifikation für Avalanche-Bauteil

S. 270). Mit dem Knoten dieser Komponentenhyperkante sind später alle anderen Diagrammelemente verbunden, wodurch die Reduktion des KSGs, die für die spätere Verarbeitung vorteilhaft ist, vereinfacht wird (vgl. Beispiel 3.12, S. 91).

Spezifikation von Verbindungshyperkanten

Im KSG eines Diagramms werden die Beziehungen zwischen Komponentenhyperkanten durch Verbindungshyperkanten repräsentiert, die Knoten von Komponentenhyperkanten miteinander verbinden können. In AML können mögliche Beziehungen zwischen Medienkomponenten mittels Assoziationen modelliert werden. Assoziationen, die bestimmte Bedingungen erfüllen, können in AML/GT-Modellen daher als Verbindungshyperkanten ausgezeichnet werden. Genauer gesagt, werden Assoziationen im Diagramm mit *«relation»* versehen, falls sie eine Relationshyperkante spezifizieren, oder mit *«link»*, falls sie eine Mehrzweck-Hyperkante spezifizieren. Im AML/GT-Modell wird hierfür das Attribut *type* des Stereotyps *AssoziationEx* entsprechend auf *relation* bzw. *link* gesetzt.

Die Anzahl der Enden einer jeweiligen Assoziation entspricht der Stelligkeit der spezifizierten Verbindungshyperkante. Daher sind für Relationshyperkanten nur binäre Assoziationen zulässig. Mehrzweck-Hyperkanten können hingegen auch durch n-äre Assoziationen spezifiziert werden. Da auch Mehrzweck-Hyperkanten mit Stelligkeit 1 erlaubt sind, in UML/AML allerdings eine Assoziation

mit nur einem Ende verboten ist, wird für diesen Fall eine Behelfslösung benötigt. Um eine Mehrzweck-Hyperkante mit Stelligkeit 1 zu spezifizieren, wird eine binäre Assoziation benutzt, die an beiden Enden mit derselben Medienkomponente verbunden ist und zusätzlich die Beschriftung *«link,unary»* trägt. Dies ist der Hinweis darauf, dass nur das erste Ende der Assoziation relevant ist. Ein Beispiel hierfür ist Assoziation *firing* des AML/GT-Modells für B/E-Netze (siehe Beispiel 6.6, S. 207).

Da eine Verbindungshyperkante immer die Knoten von Komponentenhyperkanten verbindet, darf eine Assoziation, die als Spezifikation einer Verbindungshyperkante dienen soll, auch nur mit Knoten-Medienkomponenten verbunden werden. An jedem Ende der Assoziation muss außerdem ein kleiner grüner Kreis mit Nummer notiert werden. Diese Nummer legt den Tentakel der Verbindungshyperkante fest, der mit dem entsprechenden Knoten verbunden werden muss. Somit wird nicht nur die Verbindungshyperkante spezifiziert, sondern auch festgelegt, wie sie verwendet werden bzw. welche Knoten sie verbinden darf.[4]

Die Navigierbarkeit einer Assoziation spielt im Zusammenhang mit der Hyperkanten-Spezifikation keine Rolle. Auch die Multiplizitäten werden ignoriert, wobei es durchaus denkbar wäre, dass dadurch zulässige Grenzen von Verbindungen mittels Verbindungshyperkanten spezifiziert werden können.[5]

Beispiel 6.2 (Spezifikation von Verbindungshyperkanten für Avalanche)
Die Spezifikation von Relations- und Mehrzweck-Hyperkanten innerhalb eines AML/GT-Diagramms wird in Abb. 6.3 veranschaulicht. Darin werden auch einige Knoten-Medienkomponenten gezeigt. Die Knoten-Medienkomponente *NMarb* wird für den Knoten der *Marble*-Komponentenhyperkante verwendet, *NTop* und *NBot* jeweils für Knoten der oberen und unteren Konnektoren von Bauteilen (z. B. *Straight*, vgl. Beispiel 6.1, S. 194). Die Knoten-Medienkomponente *NTopSwitch* spezialisiert *NTop* und wird nur für *Switch*-Komponentenhyperkanten verwendet.

Die Abbildung zeigt außerdem die Spezifikation von Relations- und Mehrzweck-Hyperkanten. Mehrzweck-Hyperkante *blocked* soll bspw. *NMarb*-Knoten mit *NTopSwitch*-Knoten verbinden können, d. h., es ist eine binäre Hyperkante, die immer von *NMarb* zu *NTopSwitch* gerichtet ist. Neben weiteren Mehrzweck-Hyperkanten wird auch eine Relationshyperkante namens *topBottom* spezifiziert. Eine solche Relationshyperkante verbindet (automatisch) *NBot*-Knoten mit *NTop*-Knoten, falls sich die zugehörigen Konnektoren (untere und obere Teile von Avalanche-Bauteilen) überlappen. Der Zweck einiger Verbindungshyperkanten wird in Beispiel 3.1, S. 58, und Beispiel 3.2, S. 58, erläutert. △

[4] DiaMeta erlaubt derzeit keine derartige Spezifikation für Mehrzweck-Hyperkanten. Jede Mehrzweck-Hyperkante darf daher zur Verbindung beliebiger Knoten verwendet werden. Trotzdem bietet AML/GT an dieser Stelle zumindest eine Dokumentation über die geplante Verwendung der Mehrzweck-Hyperkante.

[5] Es wurde nicht weiter analysiert, mit welchen Methoden der KSG gegen eine solche Spezifikation geprüft werden könnte. DiaMeta unterstützt weder die Spezifikation von derartigen Grenzen noch einen entsprechenden Prüfmechanismus. Auf der anderen Seite wird aber die Prüfung des ASGs auf Grundlage der abstrakten Sprachsyntax unterstützt, die ebenfalls durch ein Modell inkl. Assoziationen mit Multiplizitäten spezifiziert wird.

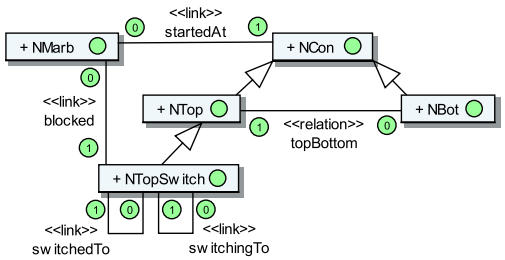


Abbildung 6.3: Spezifikation von Avalanche-Verbindungshyperkanten

Innerhalb von OCL-Ausdrücken im UML-Modell können Assoziationen bzw. die Namen der Assoziationsenden in der Regel verwendet werden, um zwischen Objekten zu navigieren und so Attribute verknüpfter Objekte auszuwerten. Auch in AML/GT sind an den Enden der Assoziationen für Verbindungshyperkanten Namen erlaubt, um einen derartigen Zugriffs-Mechanismus verwenden zu können (sei es mit OCL oder direkt mittels Java-Code, wie in dieser Arbeit). Allerdings müssen Informationen über Verknüpfungen zur Laufzeit aus dem KSG geholt werden, wodurch Probleme bzgl. Eindeutigkeit entstehen können. An dieser Stelle werden notwendige Einschränkungen aber nicht weiter ausgearbeitet.

Eine weitere Besonderheit ist, dass die durch Relations- und Mehrzweck-Hyperkanten assoziierten Medienkomponenten lediglich Knoten repräsentieren. Diese Knoten haben in der vorgestellten Klasse von Hypergraphen allerdings keine Merkmale wie Attribute etc. Es wird daher erwartet, dass die Methode *getEdgeComponent* (siehe Abschnitt 6.4, S. 199) direkt nach der Referenzierung aufgerufen wird, um Zugriff auf die Medienkomponenteninstanz der übergeordneten Hyperkanten-Medienkomponente zu erhalten.

6.4 Spezifikation von Darstellung und Konnektoren

Grundsätzlich kann das Aussehen von Diagrammkomponenten genauso durch Medienkomponenten modelliert werden, wie in Kapitel 5 beschrieben. Im Rahmen von AML/GT kann die Art einer Medienkomponente diesbezüglich aber noch genauer spezifiziert werden, was eine präzisere Codegenerierung für Editoren, und auch eine aussagekräftigere Dokumentation der modellierten Sprache ermöglicht. Neben Medienkomponenten für Hyperkanten und Knoten (vgl. Abschnitt 6.3, S. 192) gibt es zwei weitere Arten, die in den nächsten beiden Unterabschnitten vorgestellt werden. Danach wird beschrieben, wie das AML-Basisframework aufgrund der neuen Arten angepasst wurde.

Medienkomponentenart: *primitive*

Eine weitere Art von Medienkomponenten sind die sogenannten *primitiven Medienkomponenten*. Sie können durch Setzen des *type*-Attributs auf *primitive* modelliert werden. Ein kleines Augen-Symbol als Stereotyp-Icon der Medienkomponente stellt dies nach außen hin dar (vgl. Beispiel 6.1, S. 194).

Primitive Medienkomponenten sind in AML/GT die einzige Medienkomponentenart, die eine direkte grafische Repräsentation besitzen, d. h. eine grafische Repräsentation, die sich nicht (ausschließlich) durch Unterkomponenten bzw. innere Merkmale ableitet. Im Umkehrschluss bedeutet dies, dass andere Arten von Medienkomponenten (z. B. Hyperkanten-Medienkomponenten) keine direkte grafische Repräsentation besitzen. Ihr Aussehen kann also ausschließlich durch innere Merkmale spezifiziert werden.

Technisch gesehen, muss für eine primitive Medienkomponente individueller Code implementiert werden, der die Medienkomponente unter Berücksichtigung der verfügbaren Attribute zeichnet. Bei der Generierung von Editoren bedeutet dies, dass ein Grundgerüst erstellt wird, in dem der Code zum Zeichnen eingebettet werden kann. Davon wird allerdings in keinem Beispiel der Arbeit Gebrauch gemacht, da für die grafische Repräsentation ausschließlich primitive Medienkomponenten des Basisframeworks verwendet werden, deren Zeichenroutinen bereits im Framework enthalten sind (siehe Abschnitt 5.5, S. 182).

Medienkomponentenart: *area*

Für sogenannte *Bereichs-Medienkomponenten* wird *type* auf *area* gesetzt. Ein gelbes, gestricheltes Stereotyp-Icon zeigt dies an. Der einzige wesentliche Unterschied zu primitiven Medienkomponenten ist, dass Bereichs-Medienkomponenten zwar keine (direkte) visuelle Repräsentation besitzen, wohl aber eine spezifizierte Form. Für jede Bereichs-Medienkomponente ist also individueller Code erforderlich, der diese Form berechnet und zur Verfügung stellt. Ansonsten gilt die Beschreibung von primitiven Medienkomponenten analog auch für Bereichs-Medienkomponenten.

Die Verwendung einer Bereichs-Medienkomponente ist v. a. als Typ eines inneren Merkmals von Knoten-Medienkomponenten sinnvoll. Sie können so die genaue Lage und Form von Konnektoren einer Diagrammkomponente festlegen. Abb. 6.2, S. 195, zeigt, wie zwei rechteckige Konnektoren am oberen und unteren Ende der geraden Bahn festgelegt werden.

Angepasstes Basisframework

Das in Abschnitt 5.5, S. 175, beschriebene Basisframework wurde für AML/GT-Modelle angepasst und erweitert. Abb. 6.4 zeigt das entstandene AML/GT-Basisframework, das auch für alle AML/GT-Modelle dieser Arbeit verwendet wird. Allerdings ist auch dieses Basisframework nur als Beispiel zu verstehen, wie grundlegende Elemente verschiedenen AML/GT-Modellen zur Verfügung gestellt werden können.

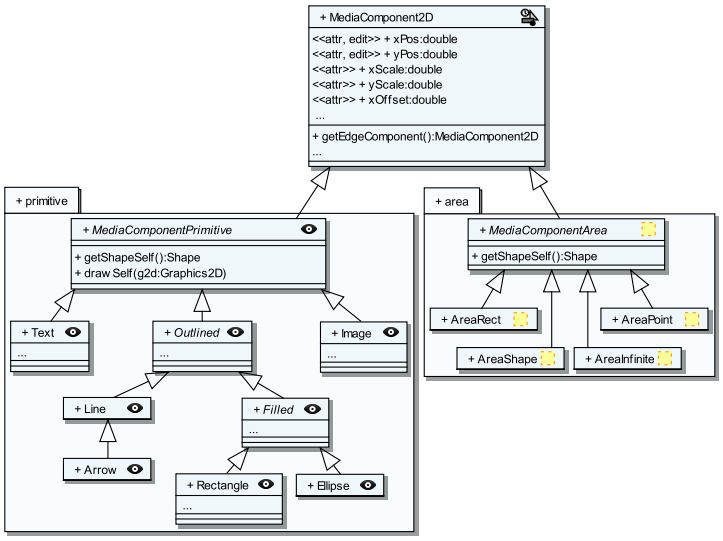


Abbildung 6.4: Erweiterung des Basisframeworks durch Verbindungsbereiche

Bei der Erstellung des AML/GT-Basisframeworks wurde darauf geachtet, dass die in den vorherigen Abschnitten beschriebenen Medienkomponenten-Arten gesetzt werden. Außerdem wurden zusätzliche Medienkomponenten für die Spezifikation von Bereichs-Medienkomponenten hinzugefügt:

- *AreaRect* modelliert einen rechteckigen Bereich, der genau wie sein grafisches Pendant *Rectangle* positioniert und transformiert werden kann.
- *AreaPoint* modelliert einen dimensionslosen, punktförmigen Bereich.
- *AreaShape* benötigt keinerlei Attribute und ermöglicht die Modellierung eines Bereichs, der exakt die gleiche Form hat, wie der grafische Anteil der zugehörigen Diagrammkomponente.
- *AreaInfinite* modelliert einen unendlich großen Bereich, d. h. einen Bereich, der das komplette Diagramm umfasst.

Ansonsten ist in Abb. 6.4 erkennbar, dass die Operationen *drawSelf* und *getShapeSelf* zum Zeichen bzw. zur Bestimmung der Form nur von Medienkomponenten der Art *primitive* angeboten werden. Die Operation *getShapeSelf* wird zusätzlich auch für die Art *area* angeboten. Beide Operationen beachten dabei keine Unterkomponenten (vgl. Abschnitt 5.5, S. 179).

Eine weitere Besonderheit ist, dass die Operationen *getParent* und *getChildren* nicht unterstützt werden, da bei der später beschriebenen Strategie zur Editor-Generierung die Hierarchie der inneren Merkmale zur Laufzeit nicht mehr existiert. Einen gewissen Ersatz bietet die Operation *getEdgeComponent* (vgl. Abschnitt 6.3, S. 197). Mit dieser Operation kann jede Medienkomponente (z. B.

eine Knoten-Medienkomponente) auf die übergeordnete Hyperkanten-Medienkomponente zugreifen. Die Verwendung dieser Operation wird in Beispiel 6.6, S. 207, erläutert.

6.5 Verwendung von Graphmustern

Im letzten Abschnitt wurde beschrieben, wie Medienkomponenten als Komponentenhyperkanten bzw. Knoten und Assoziationen als Verbindungshyperkanten spezifiziert werden können. Dadurch ist es möglich, an verschiedenen Stellen von AML/GT-Diagrammen Graphmuster zu verwenden. Zum einen können viele Situationen durch (grafische) Graphmuster anschaulich beschrieben werden. Zum anderen ermöglicht die Verwendung von Graphmustern auch eine eindeutige und automatische Übersetzung in GTRs, wie in Kapitel 7 noch gezeigt wird.

Die angesprochenen Graphmuster modellieren in allen Fällen eine Art Bedingung. Ein Einsatzgebiet ist die Verwendung der Graphmuster als Invarianten für Zustände (Abschnitt 6.5.1). AML/GT unterstützt aufgrund dieser Möglichkeit auch einen speziellen Trigger (Abschnitt 6.5.2). Außerdem können die Graphmuster als Bedingungen innerhalb von Sensoren oder als Wächter eingesetzt werden (Abschnitt 6.5.3). Und auch für Aktionssequenzelemente gibt es Besonderheiten (Abschnitt 6.5.4).

6.5.1 Graphmuster als Zustandsinvarianten

Die Kernidee des Ansatzes aus Kapitel 4 ist, dass nicht nur statische Diagramme durch Hypergraphen beschrieben werden können. Auch die Informationen über aktuell ablaufende Animationen und zugehörige Diagrammzustände können im KSG untergebracht werden. Animationen können dabei dargestellt werden, ohne dass sich der Hypergraph ändert. Auch AML basiert auf einer ähnlichen Idee: für jede Medienkomponente kann ein Zustandsautomat modelliert werden, wobei ein AML-Animationszustand den Zustand einer Medienkomponente während einer Animation darstellt. Trotz Animation ändert sich dieser Zustand nicht.

Um beide Ansätze kombinieren zu können, wird davon ausgegangen, dass ein Hypergraph, der wie oben erläutert eine animierte Situation beschreibt, in Teilhypergraphen zerlegt werden kann. Dabei repräsentiert jeder Teilhypergraph den Zustand einer Diagrammkomponente, und es liegt nahe, AML-Zustände mit diesen Teilhypergraphen zu verknüpfen. Genau dies ist in AML/GT-Zustandsdiagrammen möglich.

In AML/GT wird jedem modellierten Zustand ein (Teil-)Hypergraph bzw. Graphmuster zugeordnet, das mindestens die *Zustandsautomaten-Eigentümerhyperkante* inkl. zugehörige Knoten enthält. Damit ist die Komponentenhyperkante gemeint, die dem Zustandsautomateneigentümer zugeordnet wird. Zusätzlich dürfen Mehrzweck-Hyperkanten und damit verbundene Knoten Teil des Graphmusters sein. Der Tentakel 0 einer solchen Mehrzweck-Hyperkante muss allerdings mit einem Knoten der Zustandsautomaten-Eigentümerhyperkante verbunden sein. Das so gebildete Graphmuster bestimmt, welcher Teilhypergraph im KSG

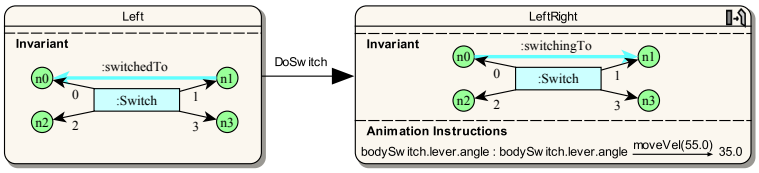


Abbildung 6.5: Invarianten-Graphmuster in Avalanche

bezogen auf die Diagrammkomponente existieren muss, während sich diese im modellierten Zustand befindet. Das Graphmuster stellt somit eine Invariante des Zustands dar und wird daher auch *Invarianten-Graphmuster* (engl. *invariant pattern*) genannt. Solche Invarianten-Graphmuster sind auch im Rahmen zusammengesetzter Zustände verwendbar.

Visuell wird ein Invarianten-Graphmuster zusammen mit dem Schlüsselwort *Invariant* innerhalb der Zustandsbox notiert. Im Modell wird das Invarianten-Graphmuster für einen Zustand textuell als UML-Zusicherung *stateInvariant* mit eingebetteter *uml::OpaqueExpression* untergebracht, deren Attribut *language* auf „*graph_pattern*“ gesetzt wird. Wie ein Graphmuster textuell kodiert werden kann, soll an dieser Stelle nicht beschrieben werden und ist im Rahmen von AML/GT auch nicht festgelegt.

An dieser Stelle sei noch einmal angemerkt, dass jeder Zustand ein Invarianten-Graphmuster und jedes Invarianten-Graphmuster die Zustandsautomaten-Eigentümerhyperkante inkl. Knoten enthält. Darüber hinaus ist festgelegt, dass die Zustandsautomaten-Eigentümerhyperkante immer den Bezeichner *owner* und die deren Knoten immer die Bezeichner *n0*, *n1*, ..., tragen müssen. Invarianten-Graphmuster oder Zustandsautomaten-Eigentümerhyperkante können daher auch weggelassen werden (visuell und intern im Modell). In den vollständigen Diagrammen im Anhang A ist dies bspw. der Fall, um Platz zu sparen.

Beispiel 6.3 (Invarianten-Graphmuster in Avalanche)
In Abb. 6.5 werden zwei Zustände einer Weiche (*Switch*) der Sprache Avalanche gezeigt. Im Zustand *Left* liegt der Hebel der Weiche auf seiner linken Seite. Die Invariante des Zustands legt fest, dass eine *switchedTo* Mehrzweck-Hyperkante die Knoten *n1* und *n0* der Komponentenhyperkante der Weiche miteinander verbinden muss, während der Zustand aktiv ist. Im Animationszustand *LeftRight* kippt der Hebel gerade von der linken zur rechten Seite. Dies wird während des Zustands durch Mehrzweck-Hyperkante *switchingTo* von *n0* zu *n1* angezeigt. △

Dass die Invarianten-Graphmuster im KSG existieren, wenn ein zugehöriger Zustand eingenommen wird, müssen die Zustandsübergänge selbst sicherstellen. Daher werden Mehrzweck-Hyperkanten der Zielzustände während des Zustandsübergangs erzeugt. Auf der anderen Seite werden die Mehrzweck-Hyperkanten der Quellzustände aus dem KSG entfernt.

Wichtig für ein gültiges AML/GT-Modell ist es daher, dass aus dem Zustandsübergang eindeutig hervorgeht, wie die Mehrzweck-Hyperkanten des Invarianten-Graphmusters des Zielzustands erzeugt werden, d. h., mit welchen Knoten diese

verbunden werden. Daher müssen alle im Invarianten-Graphmuster des Zielzustands verwendeten Knoten im Rahmen eines Zustandsübergangs bekannt sein durch: (a) das Invarianten-Graphmuster des Quellzustand, (b) zusätzliche Bedingungs-Graphmuster (siehe Abschnitt 6.5.3, S. 204) oder (c) Komponentenhyperkanten, die während des Zustandsübergangs erzeugt werden (siehe Abschnitt 6.5.4, S. 209). Diese Anforderung wird auch bei Beschreibung des Generierungsmechanismus in Kapitel 7 deutlich.

Im Rahmen eines Zustandsübergangs müssen manche Knoten verschiedener Graphmuster daher als übereinstimmend betrachtet werden. Im Allgemeinen ist dies der Fall, wenn identische Bezeichner verwendet werden. Es ist allerdings nicht in allen Fällen möglichen, übereinstimmenden Knoten identische Bezeichner zu geben. Daher können an der Transition sogenannte *Übereinstimmungs-Anweisungen* (engl. correspondence statements) notiert werden. Die Syntax einer Übereinstimmungs-Anweisung lautet „*rhs node* $x := y$“ mit x und y als Knotenbezeichner, die übereinstimmen sollen. Dabei wird Bezeichner x innerhalb des Invarianten-Graphmusters des Zielzustands und Bezeichner y innerhalb der übrigen Graphmuster (z. B. des Invarianten-Graphmusters des Quellzustands) verwendet. Im Modell wird die Übereinstimmungs-Anweisung im Attribut *rhs* des zu verwendenden Stereotyps *Correspondence* untergebracht. In einem späteren Unterabschnitt wird eine solche Anweisung verwendet (siehe Beispiel 6.6, S. 207).

Die in Kapitel 7 beschriebene Vorgehensweise zeigt außerdem, dass eine Mehrzweck-Hyperkante, die beim Verlassen eines Zustands entfernt wird, nicht genau der Mehrzweck-Hyperkante entsprechen muss, die beim Eintritt des Zustands erzeugt wurde. Dies ist möglich, falls mehrere Mehrzweck-Hyperkanten mit derselben Markierung so mit der Zustandsautomaten-Eigentümerhyperkante verbunden sind, wie durch das Invarianten-Graphmuster beschrieben. Welche dieser Mehrzweck-Hyperkanten dann entfernt wird, ist nicht bestimmt.

6.5.2 *inv broken*-Ereignis und -Trigger

Auch wenn die Zustandsübergänge von und zu Zuständen mit Invarianten-Graphmustern verantwortlich für die Erzeugung und Entfernung von Mehrzweck-Hyperkanten des Graphmusters sind, so ist es möglich, dass derartige Mehrzweck-Hyperkanten auch entfernt werden, während der zugehörige Zustand aktiv ist. Zum einen kann es dazu kommen, wenn Komponentenhyperkanten inkl. Knoten aufgrund ihres spezifizierten Verhaltens entfernt werden. Auch Mehrzweck-Hyperkanten, die mit diesen Komponentenhyperkanten (bzw. deren Knoten) verbunden sind, müssen in diesem Fall entfernt werden (vgl. Abschnitt 3.2.3, S. 73). Zum anderen soll die Nutzung von AML/GT generell nicht verhindern, dass innerhalb eines Systems auch GTs durchgeführt werden, die nicht durch das AML/GT-Modell bedingt sind. Durch solche GTs können beliebige Hyperkanten entfernt und Invarianten somit verletzt werden. Dies kann bspw. in Editoren geschehen, die „freies Editieren“ unterstützen, d. h., beliebige Diagrammkomponenten können jederzeit gelöscht werden.

Falls die Mehrzweck-Hyperkanten eines Invarianten-Graphmusters entfernt und damit die Invariante des Zustands verletzt wird (während dieser aktiv

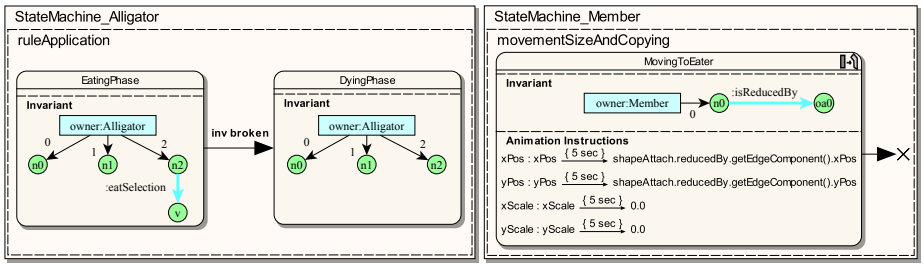


Abbildung 6.6: *inv broken*-Trigger in Alligator Eggs

ist), wird an die Zustandsautomaten-Eigentümerhyperkante ein sogenanntes *inv broken-Ereignis* („invariant broken“) geschickt. Für die Verarbeitung dieser Ereignisse kann eine Transition mit *inv broken-Trigger* modelliert werden.

Für eine entsprechende Transition, oder besser gesagt das Invarianten-Graphmuster des Zielzustands, gilt allerdings die Einschränkung, dass es sich nicht auf die Knoten des Invarianten-Graphmuster des Quellzustands beziehen darf (die Knoten der Zustandsautomaten-Eigentümerhyperkante ausgenommen). Im Rahmen der Verletzung der Invariante könnte ein solcher Knoten nämlich bereits entfernt worden und zur Laufzeit daher nicht verfügbar sein.

Ein *inv broken*-Trigger wird visuell mit dem Schlüsselwort *inv broken* notiert. Im Modell selbst muss für den zu erstellenden Trigger ein Ereignis (*uml::ExecutionEvent*) mit diesem Schlüsselwort als Name existieren.

Falls keine Transition mit *inv broken*-Trigger für einen Quellzustand mit Invarianten-Graphmuster modelliert wird, darf das Graphmuster zur Laufzeit unter keinen Umständen verletzt werden. Ist dies trotzdem der Fall, sollten Meldungen zur Laufzeit auf einen entsprechenden Fehler hinweisen.

Beispiel 6.4 (*inv broken*-Trigger in Alligator Eggs)
Abb. 6.6 (links) illustriert zwei wichtige Zustände von Alligator Eggs-Alligatoren, während die Fressregel ausgeführt wird. Beide haben im vollständigen Modell einige Unterzustände, die für das Beispiel allerdings nicht relevant sind. Der Zustand *EatingPhase* ist während des eigentlichen Fressvorgangs aktiv. Die Invariante des Zustands legt fest, dass während des Zustands eine Mehrzweck-Hyperkante *eatSelection* existiert. Sie verbindet einen Knoten der Alligator-Komponentenhyperkante mit dem Opfer, das gefressen wird. Ist eine ganze Familie das Opfer, so ist dies das oberste Element der Familie (ein alter oder hungriger Alligator).

Während sich der fressende Alligator in Zustand *EatingPhase* befindet, gehen alle Opfer in den Animationszustand *MoveToEater* über. Dieser Animationszustand ist für die Basismedienkomponente *Member* definiert, von der alle Medienkomponenten (Alligatoren und Eier) abgeleitet sind, und wird ebenfalls in Abb. 6.6 (rechts) gezeigt. Durch die Invariante des Animationszustands wird festgelegt, dass jedes Opfer durch eine Mehrzweck-Hyperkante *isReducedBy* mit dem fressenden Alligator verbunden wird. Außerdem soll sich jedes Opfer in-

nerhalb von fünf Sekunden auf den fressenden Alligator, der mit *reducedBy*[6] referenziert werden kann, zubewegen und dabei schrumpfen.

Sobald ein Opfer am Ziel ankommt bzw. die Animationsanweisungen in *MoveToEater* ihr Ende erreicht haben, wird die dargestellte Transition geschaltet, was den Übergang zum Terminierungsknoten zur Folge hat. Deshalb wird das entsprechende Opfer aus dem Diagramm entfernt. Wird allerdings die zugehörige Komponentenhyperkante aus dem Graphen entfernt, müssen auch alle mit ihren Knoten verbundenen Verbindungshyperkanten wie bspw. *eatSelection* entfernt werden. Dadurch wird wiederum die Invariante des Zustands *EatingPhase* des fressenden Alligators verletzt. Der Zustand *EatingPhase* muss in diesem Fall über die Transition mit *inv broken*-Trigger verlassen werden. Die Transition wird konsequent dazu benutzt, den Ablauf der Fressregel fortzusetzen. Der Alligator wechselt dadurch in den Zustand *DyingPhase*, in dem er sich zunächst auf den Rücken dreht und später ebenfalls entfernt wird. Details für diesen Zustand werden in Abb. A.4, S. 272, dargestellt. △

6.5.3 Graphmuster als Bedingungen

Viele Transitionen dürfen nur in bestimmten Situationen ausgelöst werden. Dies kann bspw. durch Bedingungen innerhalb von Sensoren erreicht werden. Diese bestimmen, wann ein Sensor aktiviert wird. Eine weitere Möglichkeit ist die Verwendung von Wächtern an den Transitionen. OCL ist ein mächtiges Mittel, um solche Bedingungen zu modellieren. In AML/GT können allerdings auch Graphmuster verwendet werden, um Bedingungen anschaulich zu modellieren.

Innerhalb von Sensoren oder als Wächter können daher sogenannte *Bedingungs-Graphmuster* genutzt werden. Diese legen fest, welcher Teilhypergraph existieren muss, damit die geforderte Bedingung als erfüllt gilt. Dabei dürfen sich Bedingungs-Graphmuster auch auf Invarianten-Graphmuster von Quellzuständen beziehen, indem für übereinstimmende Hyperkanten und Knoten identische Bezeichner verwendet werden.

Da ein einzelner Sensor für viele Transitionen (mit unterschiedlichen Quellzuständen und ggf. anderen Bezeichnern) eingesetzt werden kann, ist es in manchen Fällen nicht möglich, Knoten für alle Fälle passend zu bezeichnen. Daher lässt sich die Übereinstimmung von Knoten in Bedingungs-Graphmustern von Sensoren parametrisieren. Erreicht werden kann dies erneut durch Übereinstimmungs-Anweisungen, die an den Transitionen modelliert werden (vgl. Abschnitt 6.5.1, S. 202). In diesem Fall lautet die Syntax jedoch „*lhs node* $x := y$“. Dabei sind x und y die Knotenbezeichner, die übereinstimmen sollen. Der Bezeichner x wird innerhalb des Bedingungs-Graphmusters eines Sensors verwendet, während Bezeichner y innerhalb der übrigen Graphmuster „der linken Seite“ (Invarianten-Graphmusters des Quellzustands oder das Bedingungs-Graphmusters des Wächters) verwendet wird. Im Modell wird für die Übereinstimmungs-Anweisung das Attribut *lhs* des Stereotyps *Correspondence* gesetzt.

[6] Dies ist der Name eines Endes der Assoziation *isReducedBy*, die durch eine gleichnamige Mehrzweck-Hyperkante im KSG umgesetzt wird. Derartige Details können dem vollständigen (strukturellen) Modell aus Abb. A.2, S. 270, entnommen werden.

Bedingungs-Graphmuster innerhalb der Sensoren enthalten immer mindestens die Komponentenhyperkanten des Sensoreigentümers (mit Bezeichner *owner* und den Knoten *n0*, *n1*, ...) und aller Verursacher (mit dem jeweiligen Verursachernamen als Bezeichner). Die Knoten der Verursacher müssen die Bezeichner *oa0*, *oa1*, ... (erster Verursacher), *ob0*, *ob1*, ... (zweiter Verursacher), usw., tragen. Wie bei den Invarianten-Graphmustern können diese Elemente in den Bedingungs-Graphmustern weggelassen werden.

Die obigen Regeln spiegeln die Einschränkung wider, dass neben dem Sensoreigentümer auch Verursacher eine Repräsentation als Komponentenhyperkante besitzen müssen. Es können daher nicht beliebige Medienkomponenten als Sensoreigentümer oder Verursacher gewählt werden, sondern nur Medienkomponenten, die als Komponentenhyperkanten markiert sind. Dies stellt eine Einschränkung ggü. AML dar (vgl. Abschnitt 6.2, S. 192).

Eine Besonderheit gibt es im Falle von Nachrichtensensoren. Die Komponentenhyperkante des Verursachers (maximal einer, vgl. Abschnitt 6.2, S. 192) inkl. dessen Knoten werden im Bedingungs-Graphmuster gestrichelt dargestellt. Dies hat den Hintergrund, dass diese Elemente bei der Matchsuche mit anderen Elementen der in Frage kommenden Graphmuster zusammenfallen dürfen (vgl. Beispiel 3.5, S. 68). Ansonsten könnten Hyperkanten-Medienkomponenten keine Nachrichten an sich selbst schicken, da *owner* (der Sender) und die Komponentenhyperkante des Empfängers verschieden sein müssten.

Neben Bedingungs-Graphmustern können für einen Sensor oder Wächter sowohl funktionale als auch negative Anwendungsbedingungen modelliert werden. Diese Anwendungsbedingungen können Hyperkanten und Knoten des Bedingungs-Graphmusters und des Invarianten-Graphmusters des Quellzustands einbeziehen. Nur falls diese Anwendungsbedingungen erfüllt werden, wird ein Sensor aktiviert bzw. kann eine Transition geschaltet werden. Funktionale Anwendungsbedingungen können dabei in einer systemabhängigen, textuellen Sprache erfasst werden. In dieser Arbeit werden Java-Ausdrücke verwendet, die später in angepasster Form direkt in GTRs und anschließend im resultierenden Java-Code eingebettet werden. Eine solche Anwendungsbedingung wurde bereits in Beispiel 3.7, S. 72, präsentiert. Die Syntax für Pfadausdrücke wird in Beispiel 3.9, S. 77, vorgestellt. Es sei außerdem angemerkt, dass im Kontext der Anwendungsbedingung alle Hyperkanten und Knoten der Graphmuster durch deren Bezeichner referenziert werden können. Zusätzlich ist die (optionale) *this*-Referenz verfügbar, die mit der *owner*-Referenz gleichzusetzen ist.

Visuell werden Bedingungs-Graphmuster und zusätzliche Anwendungsbedingungen innerhalb des „[]“-Blocks (Wächter) bzw. des Sensors dargestellt. Das Graphmuster selbst wird mit dem Schlüsselwort *pattern* versehen, negative Anwendungsbedingungen mit *nac* und zusätzliche funktionale Anwendungsbedingungen mit *cond* („condition“) bzw. *where*, wenn sie im Rahmen einer negativen Anwendungsbedingung verwendet wird. Im Modell werden sie textuell als UML-Zusicherung mit eingebetteter *uml::OpaqueExpression* untergebracht. Das Attribut *language* der *uml::OpaqueExpression* muss dabei auf *amlgt_constraint* gesetzt werden. Die Kodierung ist weder für die Graphmuster noch für die zusätzlichen Anwendungsbedingungen festgelegt.

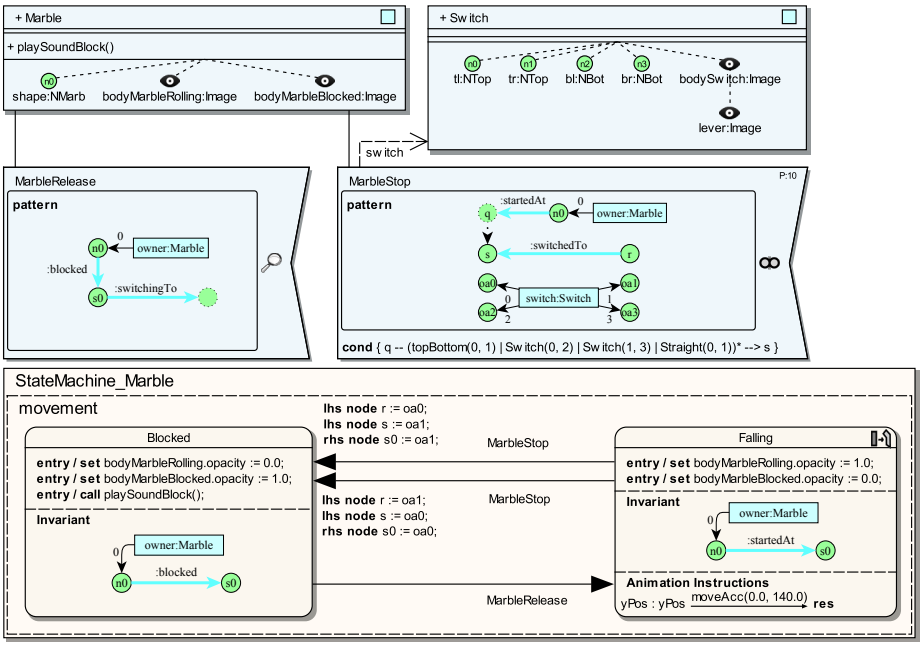


Abbildung 6.7: Bedingungs-Graphmuster von Sensoren in Avalanche

Beispiel 6.5 (Bedingungs-Graphmuster von Sensoren in Avalanche)
Innerhalb des Modells der Sprache Avalanche werden Bedingungs-Graphmuster häufig genutzt. Abb. 6.7 zeigt die zwei Zustände *Falling* und *Blocked* von Murmeln und wie Zustandsübergänge zwischen diesen Zuständen modelliert werden können. Außerdem werden oben die beiden Medienkomponenten *Marble* (die Murmel) und *Switch* (die Weiche) inkl. der wichtigsten strukturellen Merkmale veranschaulicht.

Für den Übergang von Zustand *Falling* zu *Blocked* ist Kollisionssensor *MarbleStop* verantwortlich. Dieser ist an den beiden Transitionen notiert und wird aktiviert, sobald die Murmel auf einen Hebel trifft, welcher die Bahn der Kugel blockiert. Die implizite Medienkomponente des inneren Merkmals *lever* (der eigentliche Hebel) als Verursacher zu wählen, wie in Beispiel 5.11, S. 167, ist in AML/GT nicht erlaubt, da diese Medienkomponente nicht durch eine Komponentenhyperkante repräsentiert wird (vgl. Abschnitt 6.2, S. 192). Im konkreten Fall ist die Angabe von *Switch* als Verursacher aber kein Problem, da die Berechnung des Kollisionszeitpunkts ohnehin die Erstellung von zusätzlichem Programmcode erfordert (siehe Abschnitt 7.3.4, S. 257) und Code zur Berechnung der Fallzeit bis zum Hebel entsprechend umgesetzt werden kann.

Mit dem Bedingungs-Graphmuster in *MarbleStop* soll geprüft werden, ob sich die Murmel auf der Bahn befindet, die vom Hebel der Weiche blockiert wird. Die Komponentenhyperkanten *owner* und *switch* sind dabei automatisch Teil des Bedingungs-Graphmusters. Die *startedAt*-Verbindungshyperkante zeigt auf den Knoten, der die Startposition der Murmel repräsentiert. Die *switchedTo*-

Verbindungshyperkante gibt an, ob der Hebel von *switch* auf die linke oder rechte Seite gekippt ist. Dabei wird Parametrisierung genutzt, denn anstelle der Knoten *r* und *s* sollen eigentlich die Knoten *oa0* und *oa1* (die Knoten von *switch*) verwendet werden. Welche Knoten übereinstimmen sollen, wird durch die Übereinstimmungs-Anweisung („lhs node“) an den Transitionen angegeben, die den Sensortrigger *MarbleStop* nutzen. Dadurch kann der Sensor einmal modelliert, aber in zwei Fällen genutzt werden: nach links oder rechts gekippt. Zusätzlich wird ein Pfadausdruck als funktionale Anwendungsbedingung verwendet. Dadurch wird die Aktivierung der Sensorinstanz an eine weitere Bedingung gekoppelt. Der Pfadausdruck stellt sicher, dass sich die Murmel auf der Bahn befindet, die vom Hebel blockiert wird (vgl. *path_down* in Beispiel 4.3, S. 107).

Für den Übergang von Zustand *Blocked* in Zustand *Falling* wird der Bedingungssensor *MarbleRelease* verwendet. Der Zustandsübergang soll demnach ausgelöst werden, sobald im KSG eine *switchingTo*-Kante an Knoten *s0* existiert, wobei sich *s0* auf denselben Knoten bezieht wie das Invarianten-Graphmuster des Zustands *Blocked*. Der Knoten repräsentiert dabei ein abstraktes Element, das den Fall der Murmel gestoppt hat und danach blockiert. Es handelt sich also immer um den Knoten einer *Switch*-Komponentenhyperkante (oben links oder oben rechts). Sobald der Hebel der Weiche seine Position wechselt, wird eine *switchingTo*-Kante mit diesem Knoten verbunden (vgl. Beispiel 6.3, S. 201). Dadurch wird der Bedingungssensor *MarbleRelease* (sofort) aktiviert, was zum beschriebenen Zustandsübergang führt.

Das Beispiel zeigt zusätzlich die Nutzung einer Übereinstimmungs-Anweisung, die genauer festlegt, wie das Invarianten-Graphmuster eines Zielzustands durch eine Transition erzeugt werden muss („rhs node“). Die dargestellten Zustände *Blocked* und *Falling* verwenden beide einen Knoten mit Bezeichner *s0* innerhalb ihrer Invarianten-Graphmuster. Beim Übergang von *Blocked* zu *Falling* wird im Gegensatz zum umgekehrten Fall keine Übereinstimmungs-Anweisung benötigt, da davon ausgegangen wird, dass die entsprechenden Knoten der beiden Zustände übereinstimmen. Anschaulich formuliert, wird der Knoten der Weiche, welcher die Murmel zuvor geblockt hat, zu dem Knoten, an dem die Murmel beginnt zu fallen. Umgekehrt wird eine Übereinstimmungs-Anweisung verwendet, um auszudrücken, dass *s0* des Zielzustands *Blocked* je nach Transition entweder *oa0* oder *oa1* entspricht (und somit nicht dem Knoten *s0* von *Falling*). Dies ist in jedem Fall ein Knoten des Bedingungs-Graphmusters im Kollisionssensor *MarbleStop*. Durch die Transition und zur Erzeugung des Invarianten-Graphmusters in *Blocked* muss also eine *blocked*-Kante zu einem solchen Knoten erzeugt werden, d. h., die Murmel wird „an dieser Stelle“ vom Hebel blockiert. △

Beispiel 6.6 (Bedingungs-Graphmuster von Wächtern in B/E-Netzen)
Auch für die Umsetzung eines Editors für B/E-Netze werden Bedingungs-Graphmuster verwendet. Abb. 6.8 zeigt ein ähnliches (Teil-)Modell wie bereits in Beispiel 5.12, S. 168, gezeigt wurde, allerdings unter Verwendung von AML/GT. Oben im Diagramm werden alle Hyperkanten-Medienkomponenten der Sprache für B/E-Netze dargestellt. Beim Klick auf eine *Transition* wird der Benutzersensor *Fire* flüchtig aktiviert. Dies soll dazu führen, dass die Transition von

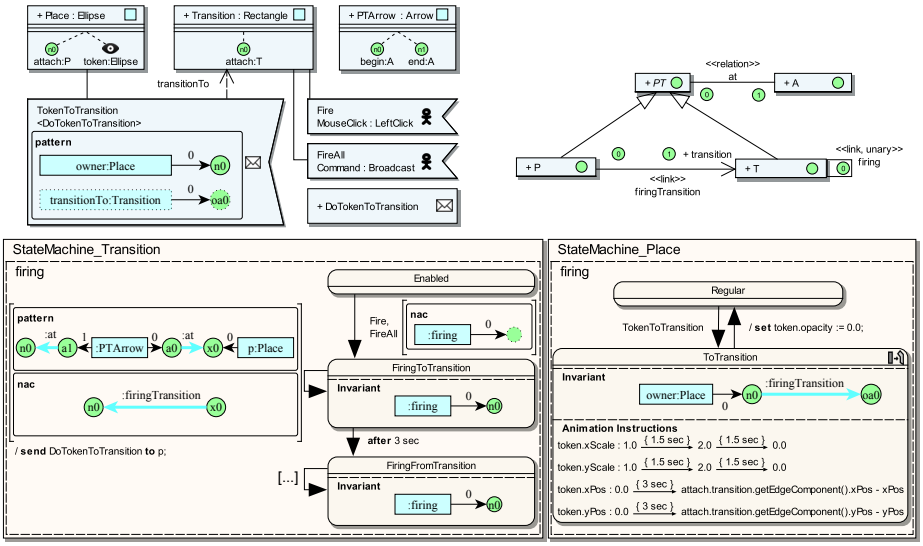


Abbildung 6.8: Bedingungs-Graphmuster von Wächtern in B/E-Netzen

Zustand *Enabled*, d. h., die Transition ist aktiv, in den Zustand *FiringToTransition* wechselt. Der Zustandsübergang setzt allerdings einen Wächter ein. Für ihn wird eine negative Anwendungsbedingung angegeben, die sicherstellt, dass keine unäre *firing*-Hyperkante im KSG existiert. Eine solche Mehrzweck-Hyperkante existiert, solange irgendeine Transition schaltet, was auch durch die Invarianten-Graphmuster der entsprechenden Zustände *FiringToTransition* und *FiringFromTransition* ersichtlich ist.

Nach Erreichen des Zustands *FiringToTransition* wird der nächste Zustandsübergang durchgeführt. Dies ist die Schleife, falls das angegebene Bedingungs-Graphmuster des Wächters erfüllt wird. Das Bedingungs-Graphmuster prüft dabei die Stellen des Vorbereichs der Transition. Aufgrund des Bedingungs-Graphmusters trägt eine potentiell gefundene Stelle bzw. deren Komponentenhyperkante den Bezeichner p. Durch die negative Anwendungsbedingung wird sichergestellt, dass mit dem Knoten *x0* dieser Stelle keine Mehrzweck-Hyperkante *firingTransition* verbunden ist, was bedeutet, dass diese Stelle noch nicht in den Schaltprozess eingebunden ist. Wird die Bedingung erfüllt, wird der Zustandsübergang durchgeführt, wodurch die Nachricht *DoTokenToTransition* an die Stelle p gesendet wird. Die Stelle wechselt nach Erhalt der Nachricht aufgrund des Nachrichtensensors *TokenToTransition* in den Animationszustand *ToTransition*, in dem sich die Marken in Richtung der Transition bewegen sollen. Um das Invarianten-Graphmuster des Animationszustands mit der Mehrzweck-Hyperkante *firingTransition* erzeugen zu können, muss der Knoten *oa0* während des Zustandsübergangs bekannt sein. Dieser ist durch den Sender der Nachricht (die Transition) bekannt, da das Bedingungs-Graphmuster des Nachrichtensensors den Sender (bzw. Verursacher) in Form der Komponentenhyperkante mit Bezeich-

ner *transitionTo* und Knoten *oa0* enthält. Bei Erreichen von *TokenToTransition* wird somit Stelle und Transition mittels *firingTransition* miteinander verbunden.

Die Schleife an Zustand *FiringToTransition* wird so oft verwendet, bis alle Stellen des Vorbereichs per *firingTransition* mit der Transition verbunden sind. Nach einer Wartezeit von drei Sekunden wird der Vorgang analog für die Stellen des Nachbereichs wiederholt. Dies wird aber nicht mehr detailliert dargestellt.

Alternativ zum Benutzersensor *Fire* kann auch *FireAll* zum Schalten einer Transition verwendet werden. Dies ist ein spezieller Benutzersensor, der die zusätzlichen Angabe „Broadcast“ trägt.[7] Derartige Benutzersensoren können eingesetzt werden, wenn Benutzernachrichten durch eine Aktion (z. B. ein Klick auf einen Button) nicht nur an eine spezielle Diagrammkomponente, sondern an alle Diagrammkomponenten des zugehörigen Typs geschickt werden sollen. In Bezug zu *FireAll* bedeutet dies, dass keine Transition gewählt werden muss. Bei Ausführung des Kommandos schaltet eine aktive Transition irgendwo im Diagramm. Das Schalten mehrerer Transitionen wird durch den zwischen *Enabled* und *FiringToTransition* modellierten Wächter verhindert. △

6.5.4 Möglichkeiten in Aktionssequenzen

Die meisten Arten von Aktionssequenzelementen können in AML/GT genauso verwendet werden wie in AML. Eine Besonderheit ist, dass darin Hyperkanten-Medienkomponenten referenziert werden können, die in den Graphmustern vorkommen. Hierfür werden die Bezeichner der Komponentenhyperkanten aus den Graphmustern eingesetzt, was in Beispiel 6.6, S. 207, bereits gezeigt wurde.

Eine weitere Besonderheit stellen „CreateAction“-Elemente dar. Mit ihnen können in AML/GT neben regulären Klassen nur Hyperkanten-Medienkomponenten erzeugt werden. In letzterem Fall wird nach dem Schlüsselwort *create* das Graphmuster der zu erzeugenden Komponentenhyperkante inkl. Knoten angegeben. Da bei dieser Art der Erzeugung keine Argumente für einen Konstruktor übergeben werden, müssen Hyperkanten-Medienkomponenten einen parameterlosen Konstruktor besitzen (vgl. Abschnitt 6.2, S. 192), der aufgerufen wird, falls entsprechender Code genutzt wird. Die erzeugte Komponentenhyperkante kann durch ihren Bezeichner in späteren Aktionssequenzelementen referenziert werden. Außerdem dürfen im Invarianten-Graphmuster des Zielzustands die erzeugten Knoten vorkommen.

Im AML/GT-Modell werden die Bezeichner der zu erzeugenden Elemente anstelle der Konstruktorargumente eingetragen (*constructorArgs*). Der Bezeichner der Komponentenhyperkante kommt dabei an die erste Stelle. Danach folgen die Bezeichner der Knoten.

Beispiel 6.7 (Erzeugung von Medienkomponenten in Avalanche)
In Avalanche kann der Benutzer auf das *Start*-Bauteil einer Bahn klicken, um dort eine neue Murmel zu platzieren. Dies kann in AML/GT modelliert werden,

[7] Die Verwendung von „Broadcast“ innerhalb von Benutzersensoren ist speziell für die Generierung von Editoren und den Generierungsansatz im folgenden Kapitel zugeschnitten und genau genommen kein Element von AML/GT (siehe Abschnitt 7.3.4, S. 252).

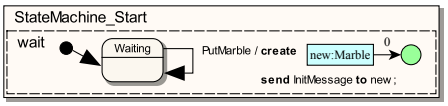


Abbildung 6.9: Erzeugung von Hyperkanten-Medienkomponenten in Avalanche

wie in Abb. 6.9 dargestellt. Die Abbildung zeigt eine Schleife innerhalb des Zustandsautomaten für das *Start*-Bauteil. Durch den Benutzersensor *PutMarble* kann der Zustandsübergang ausgelöst werden. Dabei wird zunächst eine *Marble*-Hyperkanten-Medienkomponente bzw. *Marble*-Komponentenhyperkante erzeugt. Direkt im Anschluss wird an die erzeugte Murmel (*new*) eine *InitMessage*-Nachricht gesendet. Diese Nachricht wird im Zustandsautomaten der Murmel verarbeitet und führt dazu, dass die Murmel an die richtige Stelle des Avalanche-Spielbretts gesetzt und eine *startedAt*-Mehrzweck-Hyperkante erzeugt wird (siehe vollständiges Modell in Abb. A.6, S. 274). △

6.6 Weitere Stereotypen

In diesem Abschnitt werden noch einige zusätzliche Stereotypen des AML/GT-Profils erläutert, die v. a. auch in Bezug auf die Generierung animierter Editoren nützlich sein können.

«edit» und *«attr»*

Attribute von Medienkomponenten können bei Verwendung des AML/GT-Profils durch zusätzliche Angaben genauer charakterisiert werden. Durch Angabe von *«edit»* kann bspw. spezifiziert werden, dass ein Attribut einer Medienkomponente direkt im Editor geändert werden darf. Typischerweise können solche Attributwerte durch Textfelder, Auswahlfelder etc. angezeigt und verändert werden. Allerdings können für spezielle Attribute auch andere Möglichkeiten umgesetzt werden. Sind *xPos* und *yPos* mit *«edit»* spezifiziert (vgl. Abb. 6.4, S. 199), dann ist eine Komponente per Maus frei platzier- und verschiebbar. Im Falle von *xScale* und *yScale* kann die Größe der Komponente per Maus beliebig modifiziert werden etc. Ist ein Attribut in einer Basisklasse zunächst nicht mit *«edit»* gekennzeichnet, wie bei *xScale* und *yScale* der Fall, kann dies durch Neudefinition des Attributs in einer Unterklasse geändert werden. In der Sprache Alligator Eggs müssen alle Komponenten (ihre gemeinsame Basisklasse ist *Member*) z. B. in ihrer Größe beliebig angepasst werden können, weshalb eine solche Neudefinition notwendig ist.

Weiterhin ist es möglich, Attribute einer Medienkomponente mit dem Stereotyp *«attr»* zu versehen. In diesem Fall wird festgelegt, dass dieses Attribut auch ein Attribut der Komponentenhyperkante darstellt, falls die zugehörige Medienkomponente als Hyperkanten-Medienkomponente spezifiziert wird.

Technisch betrachtet sind die beiden Angaben *«attr»* und *«edit»* nicht als *uml*-Stereotypen im Profil untergebracht (vgl. Abb. 6.1, S. 191). Stattdessen wird

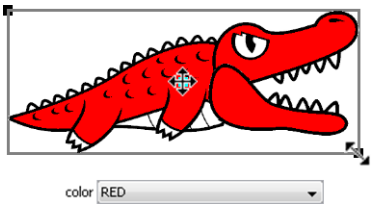


Abbildung 6.10: Editierbare Attribute

der Stereotyp *PropertyEx* eingesetzt, dessen Attribut *type* entsprechend gesetzt wird. Sowohl für die Verwendung von *«edit»* als auch *«attr»* gilt außerdem die Einschränkung, dass sie nicht für statische oder schreibgeschützte Attribute verwendet werden können (siehe Listing B.4/28).

Beispiel 6.8 (Attribute von Komponentenhyperkanten in Alligator Eggs)
Im Diagramm in Abb. 6.11 wird bspw. das Attribut *color* in Medienkomponente *ColoredMember* mit *«attr»* und *«edit»* markiert. Zum einen ist dadurch die Farbe eines *ColoredMember*, d. h. von Alligatoreiern und Alligatoren, im Editor veränderbar. Zum anderen ist *color* ein Attribut (mit identischem Typ) einer Komponentenhyperkante des Typs *ColoredMember* bzw. der abgeleiteten Typen *Egg* und *Alligator* (siehe Abb. 6.12). Abb. 6.10 zeigt, wie eine Medienkomponente *Alligator* aufgrund der verwendeten Stereotypen in einem Editor modifiziert werden kann (Änderung der Position, der Größe und der Farbe). △

«hypergraphonly»

AML verbietet nicht die Verwendung von Mehrfachvererbung für Medienkomponenten. Allerdings ist Mehrfachvererbung oftmals problematisch (vgl. [Sin95]), weshalb viele Programmiersprachen, u. a. auch Java, dieses Konzept nicht unterstützen. Da AML/GT als Transformationsbasis für eben solche Programmiersprachen eingesetzt werden kann, z. B. zur Generierung von Klassencode für Java, sollte vor der Modellierung entsprechend festgelegt werden, ob Mehrfachvererbung im AML/GT-Modell zum Einsatz kommen darf oder nicht.

Unabhängig von der Unterstützung von Mehrfachvererbung auf der Ebene des Programmcodes, kann auch das mit den Hyperkanten verbundene Typsystem (in diesem Abschnitt auch "Kantentypsystem" genannt) Mehrfachvererbung unterstützen oder nicht. Im Folgenden wird davon ausgegangen, dass Mehrfachvererbung vom Kantentypsystem unterstützt wird.

Für den Fall, dass Vererbung innerhalb des Kantentypsystems verwendet werden soll, nicht aber innerhalb des generierten Codes, existiert der Stereotyp *«hypergraphonly»*. Er kann an einem Generalisierungspfeil notiert werden, was bedeutet, dass im generierten Code zwar keine Klassenvererbung angewendet werden soll, wohl aber im Kantentypsystem. In Abschnitt 7.2.1, S. 219, wird auf dieses Konzept noch einmal näher eingegangen.

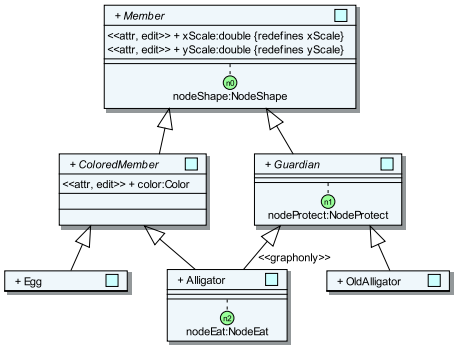


Abbildung 6.11: Stereotypen «attr», «edit» und «hypergraphonly»

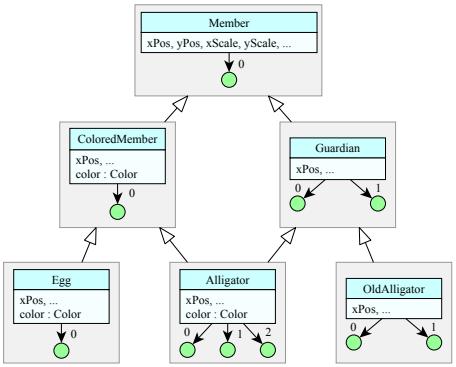


Abbildung 6.12: Hierarchie der Hyperkanten (mit Attributen)

Beispiel 6.9 (Typhierarchie in Alligator Eggs)
Für Alligator Eggs ist es sinnvoll, eine Hierarchie wie in Abb. 6.11 zu modellieren, da die Klasse für Alligatoren (*Alligator*) sowohl Aspekte eines eingefärbten Elements (*ColoredMember*) als auch Aspekte eines Beschützers (*Guardian*) in sich vereint. Im Gegensatz dazu unterstützen Alligatoreier (*Egg*) und alte Alligatoren (*OldAlligator*) nur jeweils einen dieser Aspekte. Durch die Verwendung des Stereotyps *«hypergraphonly»* kann eine Hyperkantenhierarchie abgeleitet werden, die dieser Sachlage entspricht (siehe Abb. 6.12). △

Kapitel 7

Modellgetriebene Generierung von animierten Editoren

Mittels AML/GT (Kapitel 6) können Modelle für interaktive animierte Sprachen entwickelt werden. Um das gesetzte Ziel einer modellgetriebenen Entwicklung bis zur Generierung von Editoren ohne komplexen Programmieraufwand zu erreichen, fehlt allerdings noch die Brücke zu einem Formalismus, der eine Generierung ermöglicht. Ein solcher Formalismus ist AAS/GT (Kapitel 4). Mit einem Meta-Tool wie DiaMeta, das Spezifikationen gemäß AAS/GT unterstützt, können auf diese Weise Editoren erstellt werden.

In den folgenden Abschnitten werden daher *Übersetzungsregeln* definiert, mit denen ein AML/GT-Modell in den AAS/GT-Formalismus übersetzt werden kann. Ansatzweise werden diese Übersetzungsregeln bereits in [SM12] erläutert. Hinzu kommt die direkte Generierung von Klassencode für Medienkomponenten und Zustände. Neben einer strukturellen Basis für die Editoren kann dieser Code v. a. für die (animierte) Visualisierung eingesetzt werden.

Die Übersetzungsregeln werden zum Teil formal beschrieben, wobei für die Darstellung OCL-Ausdrücke [OCL12] eingesetzt werden. Durch diese Ausdrücke werden einzelne AML/GT-Elemente, Mengen oder Folgen referenziert und abgefragt. Dies hat den Vorteil, dass keine zusätzlichen mathematischen Symbole und Konstrukte eingeführt werden müssen und eine konkrete Implementierung die OCL-Abfragen unter Umständen direkt nutzen kann. Gleichzeitig versteht sich OCL als formale Spezifikationssprache, für die – von einigen Lücken abgesehen (vgl. [CK01, BW02, Fla04]), die im verwendeten Kontext eher unwichtig sind – auch eine formale Semantik existiert.

Der Rest des Kapitels ist wie folgt aufgebaut. Zunächst wird veranschaulicht, wie AML, AML/GT und die Übersetzungsregeln für die Generierung von animierten Editoren eingesetzt werden und wie dieses Konzept eingeordnet werden kann

(Abschnitt 7.1). Danach beschreibt jeweils ein größerer Abschnitt, wie statische Teile des AML/GT-Modells (Abschnitt 7.2) und dynamische Teile des AML/GT-Modells (Abschnitt 7.3) übersetzt werden können. Ziel der Übersetzung sind dabei sowohl Code (wichtig für den statischen Teil) als auch die einzelnen Komponenten eines AAS/GTs (wichtig für den dynamischen Teil). Anschließend wird auf die konkrete Implementierung und den angepassten Generierungsprozess für das DiaMeta-Framework eingegangen (Abschnitt 7.4). Zum Abschluss werden einige verwandte Arbeiten in Bezug zu Modellierung, GTRs und Codegenerierung beschrieben (Abschnitt 7.5), womit der Abschnitt gleichzeitig die Verweise auf verwandte Arbeiten für das vorausgegangene AML/GT-Kapitel enthält.

7.1 Einordnung des Generierungskonzepts

Wie einleitend beschrieben, soll der präsentierte Ansatz eine modellgetriebene Entwicklung von Editoren für interaktive animierte Sprachen ermöglichen. Bisher wurde allerdings noch nicht genau erläutert, was es bedeutet, Editoren „modellgetrieben“ zu entwickeln. Begriffe und Konzepte rund um modellgetriebene Entwicklung wurden maßgeblich von der Object Management Group (OMG) unter dem Namen *Model Driven Architecture (MDA)* geprägt (siehe [MM03]). Die Ziele der MDA sind u. a. Interoperabilität, Plattformunabhängigkeit, Wartbarkeit, Qualität und effiziente Entwicklung von Softwaresystemen. Außerdem verbergen sich hinter dem Namen eine Menge von Standards, welche ebenfalls von der OMG definiert wurden, wie bspw. UML, MOF (das Metamodell, mit der die UML definiert wurde [MOF11]) oder Modelltransformationssprachen, die später noch erwähnt werden.

Es existieren aber auch Initiativen und Strategien anderer Organisationen und Firmen, wie z. B. das Konzept der *Software-Factory* [GS03] von Microsoft. Dieses Konzept fokussiert im Gegensatz zur MDA keine Plattformunabhängigkeit, sondern Effizienz von Entwicklung und Wartung. Zusätzlich setzt es sich teilweise recht deutlich von den Standards der OMG ab, da textuelle oder visuelle DSLs anstatt UML eingesetzt werden sollen. Insgesamt ähneln andere Strategien aber den Konzepten der MDA. Aufgrund des bereits fortgeschrittenen Stadiums der OMG-„Standards“, wurde die MDA in dieser Arbeit ggü. anderen Strategien favorisiert und diente als Basis für die Entwicklung des folgenden Ansatzes.

In der Literatur existieren viele, leicht unterschiedliche Definitionen für den Begriff „Modell“. Manche Definitionen erlauben es sogar, Programmcode einer General Purpose Language (GPL), wie Java, als Modell zu bezeichnen, was dazu führen würde, dass die Programmiersprache als Modellierungssprache bezeichnet werden darf. In Zusammenhang mit der MDA definiert die OMG den Begriff folgendermaßen:

> A *model* of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language. ([MM03])

Der Begriff „modellgetrieben“ wird definiert durch:

> MDA is an approach to system development, which increases the power of models in that work. It is *model-driven* because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification. ([MM03])

Demnach kann eine Sprachspezifikation, wie sie durch den DiaMeta-Designer (vgl. Abschnitt 3.3.2, S. 94) oder für viele andere Meta-Tools (vgl. Abschnitt 3.4, S. 95) erstellt werden kann, als Modell bezeichnet werden, auch wenn diese nicht der „klassischen“ Vorstellung eines Modells entsprechen. Die Sprach- und Editorspezifikation (das Modell) für DiaMeta unterstützt bspw. auch die direkte Einbettung von Java-Code oder die Einbettung von GTRs, die mit einer eigenen (textuellen) DSL verfasst werden müssen. Die Form der Spezifikation – eine XML-Datei – wird hauptsächlich durch eine Document Type Definition (DTD) [XML08] bestimmt (das Metamodell). Letztlich wird durch den Editor-Generator der Programmcode für den Editor erstellt. Auch die Spezifikation der abstrakten Sprachsyntax erfolgt durch ein Modell. Aus dem hierfür spezifizierten Ecore-Modell kann anschließend weiterer Code für den Editor generiert werden. Abgesehen von einigen Aspekten wie bspw. Animation, die zusätzlich programmiert werden muss, kann die Generierung von Editoren also bereits an dieser Stelle als „modellgetrieben“ bezeichnet werden.

Der in dieser Arbeit präsentierte Ansatz geht im Sinne der MDA noch einen Schritt weiter und unterstützt weitere Ebenen zur Modellierung bzw. Sprachspezifikation. Nach [MM03] gehören zur MDA folgende Modelle. Das Abstraktionsniveau des jeweiligen Modells sinkt dabei zunehmend:

- das *Computation Independent Model (CIM)* beschreibt das System formlos (meist mittels natürlicher Sprache),
- das *Platform Independent Model (PIM)* beschreibt das System unter Verwendung einer Modellierungssprache, wobei es unabhängig von einer konkreten Plattform (Programmiersprache, Technologie, Betriebssystem etc.) dargestellt werden soll, und
- das *Platform Specific Model (PSM)* beschreibt das System unter Berücksichtigung einer konkreten Plattform und ist daher bereits sehr nahe an der Implementierung bzw. dem Programmcode.

In der MDA sollen durch Modelltransformationen die Modelle einer höheren Abstraktionsebene in Modelle einer niedrigeren Ebene transformiert werden können, wobei die Modelle zunehmend verfeinert werden (vertikale Transformationen, vgl. [CE00]). Dies geht bis zur Generierung von Quellcode auf Basis des PSMs. An dieser Stelle wird die Vorgehensweise häufig auch als „generativer Ansatz“ bezeichnet. Oftmals muss das Modell der höheren Ebene dabei mit zusätzlichen Informationen versehen werden, z. B. durch Stereotypen bei UML-

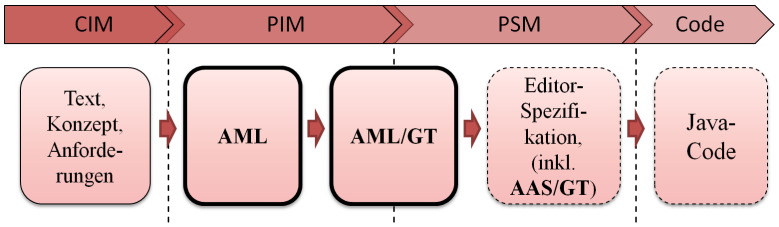


Abbildung 7.1: MDA-Schema zur Generierung von animierten Editoren

Modellen. Nach der Transformation ist im Zielmodell oder im Code teilweise Nacharbeit erforderlich, d. h., es entstehen durch die Transformation nicht zwangsläufig vollständige Modelle bzw. vollständiger Code.

Abb. 7.1 zeigt, wie AML und AML/GT innerhalb dieses Konzepts eingesetzt werden können. AML kann zur Erstellung des PIMs genutzt werden, d. h. als Mittel zur Spezifikation einer animierten Sprache, ohne auf plattformspezifische Details einzugehen. Unter Verwendung von AML/GT wird bereits festgelegt, dass animierte Diagramme durch Hypergraphen repräsentiert werden sollen. Dabei werden auch zusätzliche Details modelliert, z. B. wie Teilzustände der Komponenten durch den Hypergraphen repräsentiert werden, wie Bedingungen für Zustandsübergänge mittels Graphmustern geprüft werden können etc. Das AML/GT-Modell legt jedoch die restlichen zu verwendenden Plattformen und Technologien (z. B. DiaMeta, Java etc.) oder Editor-Merkmale noch nicht fest, weshalb das AML/GT-Modell sowohl unter PIM als auch unter PSM einzuordnen ist. Schließlich folgt nach einer weiteren Modelltransformation und dem Hinzufügen zusätzlicher Details (z. B. Icons für die Knöpfe des Editors) die plattformspezifische Sprach- und Editorspezifikation, welche den Animationsansatz gemäß AAS/GT unterstützt. Diese soll abschließend in den Programmcode des Editors transformiert werden. Im Falle von DiaMeta ist dies Java-Code. In der Regel enthält der resultierende Code nur noch wenige Abschnitte, in denen nachträglich weiterer Code eingebettet werden muss. Ein Beispiel sind Operationen, die im AML/GT-Modell zwar modelliert, aber nicht implementiert werden.

Dieser Prozess gilt in erster Linie für die Spezifikation der konkreten Sprachsyntax. Es soll offen gelassen werden, inwieweit sich die AML-Modelle und Modelltransformationen auch zur Spezifikation der abstrakten Syntax eignen. Das Klassenmodell für die abstrakte Syntax ist den AML-Modellen in vielen Fällen sehr ähnlich, z. B. auch für B/E-Netze, Avalanche oder Alligator Eggs. Eine mögliche Ableitung der abstrakten Syntax zur vollständigen Generierung eines Editors soll an dieser Stelle aber nicht weiter betrachtet werden.

7.2 Übersetzung des statischen Anteils eines AML/GT-Modells

Mit dem statischen Anteil eines AML/GT-Modells (auch *Klassenmodell*) kann spezifiziert werden, aus welchen Diagrammkomponenten die konkrete Syntax einer Sprache besteht und welche Beziehungen zwischen einzelnen Komponenten bestehen können. Zusätzlich können visuelle Strukturen und Eigenschaften festgelegt werden.

Insgesamt kann das Klassenmodell damit einen wichtigen Teil der konkreten Sprachsyntax abdecken. Nicht abgedeckt wird bspw. Verhalten und Animation von Diagrammkomponenten im Falle von animierten Sprachen. Zusätzlich wird nicht genau festgelegt, welche Diagramme oder Anordnungen von Diagrammteilen im Rahmen der konkreten Sprachsyntax gültig sind. Hier gibt es auf der Grundlage von Hypergraphen bessere Spezifikationsmöglichkeiten, als AML/GT sie bietet, z. B. Graphgrammatiken (vgl. [Min01]) oder die Reduktion des KSGs (vgl. Abschnitt 3.3.1, S. 90). Dies ist gleichzeitig einer der Gründe, warum AML/GT-Modelle Informationen über zugrunde liegende Graphen bereitstellen (vgl. Abschnitt 6.1, S. 189). Der beschriebene modellgetriebene Ansatz klammert daher bestimmte Aspekte wie bspw. die Syntaxprüfung oder die abstrakte Sprachsyntax aus. Um einen Editor zu generieren, müssen diese Aspekte separat neben dem AML/GT-Modell betrachtet werden (vgl. Abb. 1.1, S. 4). In der präsentierten Form schließt dies auch Layout und Layoutalgorithmen für Diagramme der Sprache ein.

Im Rahmen des modellgetriebenen Ansatzes wird zunächst beschrieben, wie aus dem Klassenmodell in direkter Weise Code erzeugt werden kann (Abschnitt 7.2.1). Danach wird gezeigt, wie die einzelnen Abschnitte einer auf AAS/GT-basierenden Sprachspezfikation aus dem Klassenmodell hergeleitet werden können (Abschnitt 7.2.2).

7.2.1 Generierung des Klassencodes

Beim verwendeten Ansatz muss davon ausgegangen werden, dass Diagrammkomponenten und Diagramme nicht ausschließlich auf Komponentenhyperkanten und Hypergraphen basieren. Der KSG repräsentiert zwar das Diagramm, und mittels GTR lassen sich Diagrammmodifikationen durchführen, aber einige grundlegende Editor-Bestandteile und viele in AML/GT modellierten Aspekte lassen sich nicht direkt mit diesen Techniken umsetzen: Attribute, die nicht zu Komponentenhyperkanten gehören, gewöhnliche UML-Klassen, UML-Schnittstellen, UML-Aufzählungen und einige andere UML-Elemente. Zusätzlich muss bedacht werden, dass durch das AML/GT-Modell einige Elemente nicht vollständig spezifiziert werden. Es muss bspw. die Möglichkeit geben, Methoden und Animatoren zu implementieren, eigene Zeichenroutinen für primitive Medienkomponenten festzulegen etc. Es ist daher naheliegend, aus einem AML/GT-Modell Code zu erzeugen, der Teile des Modells entsprechend umsetzt und auch um eigene Implementierungen erweitert werden kann. Im Regelfall ist dies Code der

Programmiersprache, die das *Grundsystem* (Editor-Framework, GTS, Widget-Toolkit etc.) des zu generierenden Editors bildet.

Im untersuchten Ansatz wird von einem objektorientierten Grundsystem ausgegangen, d. h., aus dem AML/GT-Modell wird Code einer objektorientierten Programmiersprache generiert, z. B. Java [Ull11] im Falle von DiaMeta. Neben gewöhnlichen Klassen werden auch Medienkomponenten in Klassen übersetzt. Dies ist auch für Hyperkanten-Medienkomponenten notwendig, die primär genutzt werden, um das Markierungsalphabet für den KSG zu spezifizieren. Dies ist notwendig, da einige Sprachaspekte in Form von Programmcode umgesetzt werden. Als Beispiel kann die animierte Visualisierung genannt werden, aber auch die folgenden Punkte.

Im Falle einer Hyperkanten-Medienkomponente sollte ein Objekt der generierten Klasse zur Laufzeit mit einer entsprechenden Komponentenhyperkante aus dem KSG verknüpft werden. Dadurch kann das Objekt die Attribute der Komponentenhyperkante auslesen. Auch der Zugriff auf andere Komponentenhyperkanten, die durch Verbindungshyperkanten verknüpft sind, kann so im Code erfolgen. Umgekehrt sollte eine Komponentenhyperkante zur Laufzeit mit „seinem“ Objekt verbunden werden. Falls das GTS entsprechende Möglichkeiten unterstützt, können dadurch im Rahmen von GTRs (v. a. in Attributberechnungsregeln, siehe Definition 3.17, S. 70) bspw. Methoden des Objekts aufgerufen werden etc.

Zusammengefasst hat es sich bewährt, Klassencode für folgende Aspekte in Bezug auf Medienkomponenten zu erzeugen. Dabei wird für die erwähnten Methoden (mit Ausnahme der Zugriffsmethoden und der Methoden des letzten Punkts) nur ein leerer Rumpf generiert, der nachträglich ausprogrammiert werden muss:

- Attribute und deren Zugriffsmethoden, basierend auf modellierten UML-Attributen[1] und UML-Assoziationen[2]
- Methoden für die modellierten UML-Operationen,
- Methoden zum Erstellen von Animatorinstanzen, basierend auf modellierten AML-Animatoren (siehe S. 219),
- Methode *drawSelf* zum Zeichnen der Medienkomponente ohne Unterkomponenten, falls die Medienkomponentenart auf *primitive* gesetzt wurde (vgl. Abschnitt 6.4, S. 199),
- Methode *getShapeSelf* zur Bestimmung der Form der Medienkomponente ohne Unterkomponenten, falls die Medienkomponentenart auf *area* oder *primitive* gesetzt wurde (vgl. Abschnitt 6.4, S. 199) und
- Methoden zum vollständigen Zeichnen der Medienkomponente und zur vollständigen Bestimmung ihrer Form (siehe S. 221).

[1] Auch zeitabhängige Werte von Attributen müssen abgefragt werden können (vgl. Definition 5.7, S. 148).

[2] Es sollten navigierbare Assoziationsenden berücksichtigt werden.

Darüber hinaus müssen auch gewöhnliche Modellelemente wie UML-Schnittstellen, UML-Aufzählungen etc. in Code umgesetzt werden. Allerdings soll auf diese Art der Codegenerierung nicht weiter eingegangen werden. Codegenerierung auf der Grundlage von UML-Klassendiagrammen ist inzwischen eine häufig eingesetzte Technik und wird auch in anderen Arbeiten beschrieben (siehe Abschnitt 7.5, S. 261). Um einen Eindruck über die Codegenerierung und die darin umgesetzten Elemente zu vermitteln, wird in Listing C.1, S. 284, ein beispielhafter Code-Ausschnitt präsentiert.

Das beschriebene Vorgehen setzt prinzipiell die Verwendung einer objektorientierten Programmiersprache zur Implementierung des Editors voraus, was in Anbetracht der Modellart und aktuell vorherrschender Systeme empfehlenswert ist. Wird aber auf diverse Möglichkeiten verzichtet (Spezifikation eines eigenen Zeichencodes etc.) und werden bestimmte AML/GT-Elemente vermieden (gewöhnliche UML-Klassen etc.), so kann der Ansatz auch mit Grundsystemen verwendet werden, die nicht objektorientiert sind und lediglich ein passendes GTS bieten.

Klassenhierarchie

Im Modell für Alligator Eggs wird Mehrfachvererbung eingesetzt, was für einige objektorientierte Programmiersprachen ein Problem darstellt (vgl. Abschnitt 6.6, S. 211). In [CMR02, TKH99] werden verschiedene Lösungsstrategien für Mehrfachvererbung besprochen.

Folgendes Beispiel veranschaulicht, wie die modellierte Typhierarchie im präsentierten Ansatz ansatzweise bewahrt werden kann, indem Schnittstellen für alle Klassen generiert werden. Diese Schnittstelle sollten dann anstelle der Klassen als Typen verwendet werden.

Im Klassencode können durch die Codegenerierung problemlos Attribute und Methoden dupliziert werden, wenn eine zusätzliche Basisklasse nicht verwendet werden darf. Dabei werden die Elemente dupliziert, die aufgrund von *«hypergraphonly»*-Generalisierungen Teil der Klasse sein müssen. Auch für EMF [EMF12] wird eine ähnliche Strategie bei der Codegenerierung eingesetzt.

Beispiel 7.1 (Klassenhierarchie und Schnittstellen in Alligator Eggs)
Alligator wird im Modell von *Guardian* abgeleitet (vgl. Abb. 6.11, S. 212). Diese Ableitung wird allerdings nicht generiert, da zwischen *Alligator* und *Guardian* ein Generalisierungspfeil mit *«hypergraphonly»* modelliert wird und Java-Klassen keine Mehrfachvererbung unterstützen. Um in Java dennoch mit einer Typhierarchie gemäß AML/GT-Modell arbeiten zu können, werden entsprechende Schnittstellen mit vollständiger Hierarchie generiert (siehe Abb. 7.2). Außerdem müssen die Elemente der Klasse *Guardian* (Attribute, Methoden etc.) neben der Klasse selbst auch für die Klasse *Alligator* erzeugt werden. △

Animatoren

Die genaue Semantik eines konkreten Animators wird nicht im AML/GT-Modell festgehalten. Animatoren werden erst nach der Codegenerierung in Form einer

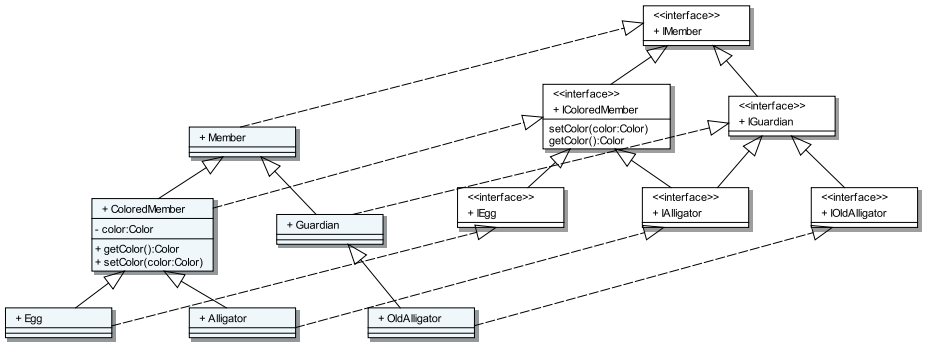


Abbildung 7.2: Generierte Klassen und Interfaces (Code)

```
1  public interface IAnimator {
2      public class AnimatorConfiguration {
3          public Object startVal;
4          public Object targetVal;
5          public double startTime;
6          public Double duration;
7          public java.util.List parameters;
8      }
9      AnimatorConfiguration getConfiguration();
10     Object getValueTimed(double time);
11     Double getDuration();
12 }
```

Listing 7.1: Schnittstelle für Animatoren (Java)

Implementierung genau spezifiziert. Allerdings muss ein passendes Grundgerüst generiert werden, in dem der Code eingebettet werden kann. Zusätzlich müssen bei der Implementierung Richtlinien eingehalten werden.

Eine Möglichkeit besteht darin, Animatoren mittels Klassen zu implementieren, die eine bestimmte Schnittstelle realisieren. Eine beispielhafte Schnittstelle wird in Listing 7.1 gezeigt. Die Methode *getConfiguration* kann genutzt werden, um die Konfiguration einer Animatorinstanz nach deren Erzeugung abzufragen. Die Methoden *getValueTimed* und *getDuration* liefern den zeitabhängigen Wert und die Animationsdauer in Abhängigkeit von der Konfiguration (und dem Zeitpunkt der Abfrage im Falle von *getValueTimed*).

Falls eine Medienkomponente im AML/GT-Modell einen Animator enthält, wird für diesen eine Methode mit leerem Rumpf generiert, in dem anschließend manueller Code eingebettet werden kann (siehe Listing C.1/17). Dabei soll die Methode eine Animatorinstanz gemäß gegebener Schnittstelle erstellen und zurückgeben, wobei die zu verwendende Animatorkonfiguration als Argument an die Methode übergeben wird. Eine derartig implementierte Methode wird auch Fabrikmethode genannt [GHJJ96].

Visualisierung

Ein Algorithmus zur Visualisierung von Medienkomponenten wurde bereits in Abschnitt 5.5, S. 180, präsentiert. Der Algorithmus basiert allerdings auf der Verfügbarkeit der Methode *getChildren*. Diese Methode wird im AML/GT-Basisframework nicht angeboten (vgl. Abschnitt 6.4, S. 199). Dies ist durch eine Besonderheit bei der Übersetzung begründet, denn die Hierarchie der zusammengesetzten Medienkomponenten wird aufgelöst und ist zur Laufzeit nicht verfügbar. Attribute von Unterkomponenten werden bspw. in der übergeordneten Medienkomponente integriert (siehe S. 223).

Aus diesem Grund muss ein individueller Zeichenalgorithmus bereits zum Zeitpunkt der Codegenerierung für jede Medienkomponente erstellt werden. Der wesentliche Unterschied zum bereits gezeigten Algorithmus ist, dass keine durch *getChildren* angeforderte Liste der untergeordneten Medienkomponenten verwendet wird, sondern eine Liste, die (nur) zum Zeitpunkt der Generierung bekannt ist. Analog gilt dies für den Algorithmus zur Bestimmung der Form.

7.2.2 Markierungsalphabet mit Attributen

In Abschnitt 6.3, S. 192, und Abschnitt 6.6, S. 210, wird beschrieben, wie AML/GT genutzt werden kann, um Hyperkanten zu spezifizieren, wobei für Komponentenhyperkanten zusätzlich Knoten und Attribute bestimmt werden. An dieser Stelle wird im eigentlichen Sinne keine Übersetzung benötigt, da die wesentlichen Aspekte direkt im AML/GT-Modell modelliert werden. Trotzdem soll in diesem Abschnitt noch einmal formal dargestellt werden, wie sich das Markierungsalphabet mit Attributen eines AAS/GTs auf Basis des AML/GT-Modells zusammensetzen lässt.

In den Hilfsdefinitionen und Übersetzungsregeln dieses Kapitels werden OCL-Abfragen verwendet, um einzelne Elemente, Mengen (*Set*), geordnete Mengen (*OrderedSet*) und Folgen (*Sequence*) basierend auf den Elementen eines gegebenen AML/GT-Modells zu bilden. Zum Einsatz kommen in diesem Zusammenhang auch wirkungsfreie Operationen des UML-Metamodells (vgl. [UML11]), des AML-Metamodells (vgl. Listing B.1, S. 275) und des AML/GT-Profils (vgl. Listing B.3, S. 278).

Hilfsdefinitionen
Einige häufig verwendete OCL-Ausdrücke sind:

- *getModel()* gibt das zu übersetzende AML/GT-Modell vom Typ *uml::Model* zurück. Dabei ist die Operation *getModel()* in nahezu jedem Kontext verfügbar, da jedes UML-Element diese Operation unterstützt.
- *objectsOfType(T)* wird als Kurzform für den OCL-Ausdruck *getModel().allOwnedElements()->select(oclIsKindOf(T)).oclAsType(T)* geschrieben, d. h., es werden alle Modellelemente eines bestimmten Typs als Menge zurückgeliefert.[3]

[3] Der Name der Operation orientiert sich dabei an einer ähnlichen Operation mit identischem Namen, die in QVT verfügbar ist [QVT11].

Mit diesen Ausdrücken können folgende Mengen von UML-Elementen definiert werden:

- $\mathcal{AO}$ = *objectsOfType(uml::Association)*
- $\mathcal{MC}$ = *objectsOfType(aml::MediaComponent)* △

Markierungsalphabet

Um das Markierungsalphabet $\mathcal{L}$ des AAS/GTs zu erhalten, müssen alle Medienkomponenten und Assoziationen des AML/GT-Modells hinsichtlich ihres *type*-Attributs (Stereotyp *MediaComponentEx* bzw. *AssociationEx*) analysiert werden.

Übersetzungsregel 7.1 (Markierungsalphabet)
Das Markierungsalphabet $\mathcal{L} = \mathcal{CMP} \cup \mathcal{REL} \cup \mathcal{GEN}$ ist gegeben durch

- $\mathcal{CMP} = \{mc \in \mathcal{MC} \mid mc.extension\_MediaComponentEx.isEdgeComponent()\}$,
- $\mathcal{GEN} = \{ao \in \mathcal{AO} \mid ao.extension\_AssociationEx.isLink()\}$ und
- $\mathcal{REL} = \{ao \in \mathcal{AO} \mid ao.extension\_AssociationEx.isRelation()\}$.

Die Stelligkeit der Hyperkanten ist gegeben durch

$$arity : lbl \mapsto \begin{cases} lbl.extension\_MediaComponentEx.numberOfNodes() \\ \quad \text{, falls } lbl \in \mathcal{CMP} \\ lbl.memberEnd\text{->}size() \\ \quad \text{, falls } lbl \in \mathcal{GEN} \wedge lbl.extension\_AssociationEx \\ \qquad .type = amlgtp::AssociationType::link \\ 1 \quad \text{, falls } lbl \in \mathcal{GEN} \wedge lbl.extension\_AssociationEx \\ \qquad .type = amlgtp::AssociationType::linkUnary \\ 2 \quad \text{, falls } lbl \in \mathcal{REL} \end{cases} . \quad \triangle$$

In dem einfachen Modell für Markierungsalphabete gehen die Informationen über Typhierarchien oder erlaubte Beziehungen verloren, z. B. welche Knoten durch Verbindungshyperkanten verbunden werden dürfen. Wenn ein Meta-Tool wie DiaMeta solche Aspekte unterstützt, müssen diese natürlich ebenfalls aus dem AML/GT-Modell hergeleitet werden. Während DiaMeta bspw. Typhierarchien unterstützt, wird die genaue Verwendung von Mehrzweck-Hyperkanten nicht geprüft und es existieren keine entsprechenden Spezifikationsmöglichkeiten (vgl. Abschnitt 6.3, S. 196).

Attribute

Das Markierungsalphabet mit Attributen $\mathcal{LV}$ des AAS/GT entsteht neben dem bereits beschriebenen Markierungsalphabet aus den Attributen der Hyperkanten-Medienkomponenten, wobei innere Merkmale in besonderer Weise berücksichtigt werden müssen. Da der Aufbau von $\mathcal{LV}$ nicht genau definiert ist (vgl. Abschnitt 3.1.2, S. 53ff), soll eine entsprechende Übersetzung nachfolgend nur schematisch beschrieben werden.

Zunächst müssen alle für eine Hyperkanten-Medienkomponente modellierten Attribute, die mit Stereotyp *«attr»* markiert sind, als Attribut der entsprechenden Markierung festgelegt werden (vgl. Abschnitt 6.6, S. 210). Dies schließt auch die Attribute ein, die innerhalb von Basismedienkomponenten modelliert werden.

Auch die Unterkomponenten von Hyperkanten-Medienkomponenten dürfen Attribute mit Stereotyp *«attr»* enthalten. Da Unterkomponenten aber keine Komponentenhyperkante spezifizieren, können entsprechende Attribute auch keiner Markierung zugeordnet werden. Auch im Falle von Knoten-Medienkomponenten können keine Attribute zugeordnet werden, da in der vorgegebenen Klasse von Hypergraphen Knoten keine Attribute besitzen. Aus diesem Grund wird die modellierte Komponentenhierarchie im präsentierten Ansatz und im Falle einer Hyperkanten-Medienkomponente nicht in den generierten Teil übertragen. Die Attribute von Unterkomponenten sollen bspw. als Attribute ihrer übergeordneten Hyperkanten-Medienkomponente betrachtet werden.

Um Namenskonflikte zu vermeiden, sollte ein erweiterter Name für das jeweilige Attribut vergeben werden, der die Herkunft einbezieht, z. B. *b_c*, falls ein inneres Merkmal *b* ein Attribut *c* besitzt. Bei der Übersetzung von Ausdrücken des AML/GT-Modells in Code muss dies entsprechend beachtet werden. Der Aufruf der Methode *getEdgeComponent* (vgl. Abschnitt 6.4, S. 199) ist lediglich als Platzhalter zu verstehen, um ein konsistentes AML/GT-Modell erstellen zu können. Prinzipiell gilt diese Vorgehensweise nicht nur für Attribute, sondern auch für modellierte Operationen.

Ein solches Schema setzt voraus, dass sich die Anzahl von Unterkomponenten nicht ändert. Daher ist die Modellierung von Multiplizitäten (ungleich 1) bei inneren Merkmalen in AML/GT nicht erlaubt (vgl. Abschnitt 6.2, S. 192).[4]

Beispiel 7.2 (Attribute von inneren Merkmalen)
Die Komponentenhyperkante, die durch Hyperkanten-Medienkomponente *Switch* spezifiziert wird (siehe Abb. 6.7, S. 206), besitzt aufgrund ihrer Basismedienkomponente (*MediaComponent2D*) und den darin modellierten Attributen mit Stereotyp *«attr»* folgende Attribute: *xPos*, *yPos*, *xScale* etc. Hinzu kommen weitere Attribute durch das innere Merkmal *bodySwitch*: *bodySwitch_xPos*, *bodySwitch_yPos*, *bodySwitch_filename* etc. Auch das innere Merkmal *lever* führt zur Spezifikation von Attributen für die Komponentenhyperkante: *bodySwitch_lever_xPos*, *bodySwitch_lever_yPos*, *bodySwitch_lever_filename* etc. Es folgen die Attribute für die übrigen inneren Merkmale *tl*, *tr*, *bl* und *br*. △

[4] Dies wurde vereinfachend festgelegt, obwohl ein derartiges Schema auch dann verwendet werden könnte, wenn Multiplizitäten mit konstanten Werten größer 1 erlaubt sind.

Konnektoren

AML/GT ermöglicht die Spezifikation von Konnektoren mittels spezieller Medienkomponenten, deren Attribut *type* auf *area* gesetzt wird (vgl. Abschnitt 6.4, S. 198). Eine exakte Übersetzung kann jedoch nicht dargestellt werden, da das Konzept von Konnektoren im Formalismus AAS/GT nicht berücksichtigt wird.

Für DiaMeta ist die Angabe eines Konnektors für jeden Knoten im Rahmen der konkreten Sprachspezifikation allerdings notwendig. Dabei wird jeder Konnektor durch einen bestimmten Bereich festgelegt. Ein solcher Bereich kann auch dynamisch spezifiziert werden, d. h., zur Laufzeit wird die Form der entsprechenden (unsichtbaren) Diagrammkomponente verwendet. Alternativ unterstützt DiaMeta spezielle Konnektoren, z. B. für einen unendlich großen Bereich. Diese werden ebenfalls im AML/GT-Basisframework abgebildet (vgl. Abschnitt 6.4, S. 198). Sie stellen Spezialfälle für die Übersetzung dar.

7.3 Übersetzung des dynamischen Anteils eines AML/GT-Modells

Im dynamischen Anteil eines AML/GT-Modells (dem *Zustandsautomaten*) können Verhalten und Animationen von Diagrammkomponenten modelliert werden. Er deckt somit weitere Aspekte der konkreten Sprachspezifikation ab, die nicht im statischen Teil verfügbar sind.

Die folgenden Übersetzungsregeln umfassen nicht alle Modellierungsmöglichkeiten. Daher wird zunächst die zulässige Teilmenge von AML/GT beschrieben (Abschnitt 7.3.1). Danach wird die Generierung von Code auf Grundlage der Zustandsautomaten skizziert (Abschnitt 7.3.2). Abschließend wird der wichtigste Teil beschrieben, und zwar die Übersetzung in GTRs und Ereignisse gemäß AAS/GT (Abschnitt 7.3.3 und Abschnitt 7.3.4).

7.3.1 Annahmen und Einschränkungen

Die in diesem Abschnitt formulierten Einschränkungen werden nicht durch AML oder AML/GT (bzw. dessen UML-Profil) vorgegeben. Diese wurden bereits in Abschnitt 5.3.2, S. 129, und Abschnitt 6.2, S. 192, beschrieben. Vielmehr handelt es sich um Einschränkungen in Bezug auf den präsentierten Generierungsansatz. Betroffen sind davon v. a. Sprachelemente von UML, die eher selten verwendet werden. Es ist davon auszugehen, dass einige der ausgesparten Elemente durch weitere Ausarbeitung des Generierungsansatzes trotzdem einsetzbar sind. Zusätzlich werden einige Annahmen bzgl. des zu übersetzenden AML/GT-Modells getroffen, um die Darstellung der Übersetzungsregeln zu vereinfachen.

Einschränkungen

Die Übersetzung von folgenden Elementen und Konzepten innerhalb von AML-Zustandsautomaten wird nicht beschrieben und wurde zum Teil nicht untersucht:

- flache/tiefe Historie,
- interne Transitionen,
- verzögerte Ereignisse,
- Neudefinition von Zuständen und Transitionen,
- Inter-Level-Transitionen[5],
- Ein- und Austrittspunkte,
- Gabelung/Vereinigung (engl. fork/join),
- Kreuzung/Entscheidung (engl. junction/choice) und
- der Einsatz von OCL in jeglicher Form.

Einige dieser Elemente werden deshalb nicht beschrieben, weil die Übersetzungsregeln und resultierenden GTRs zum einen unnötig komplex werden würden. Zum anderen gibt es meist einfachere, alternative Darstellungsformen. Daher ist es im Rahmen der präsentierten Übersetzungsregeln bspw. nicht möglich, Inter-Level-Transitionen zu modellieren. Werden zusammengesetzte Zustände verwendet, so müssen sich Quell- und Zielzustand in derselben Region befinden, d. h., es sind keine direkten Zustandsübergänge in untergeordnete oder übergeordnete Zustände erlaubt. Auch die Nutzung von Gabelung und Vereinigung oder Kreuzung und Entscheidung ist nicht möglich. Stattdessen kann ein Konzept verwendet werden, das in der Literatur manchmal auch „Self-Start“ und „Self-Termination“ bezeichnet wird (vgl. [Bee94]).

Abb. 7.3 zeigt einen Automaten mit Elementen, die nicht eingesetzt werden dürfen (oben) und einen alternativen Automaten (unten). Der alternative Automat nutzt „Self-Start“ sowie „Self-Termination“ und verzichtet auf Entscheidung, Gabelung/Vereinigung und Inter-Level-Transitionen. Bei Erreichen des Zustands B werden automatisch die UML-Regionen *regionB1* und *regionB2* betreten, was der Gabelung entspricht. Werden beide finalen Zustände erreicht, wird Zustand B verlassen, da ein Abschlussereignis dazu führt, dass die Transition mit Abschlussereignis-Trigger schaltet, und Zielzustand C wird eingenommen. Dies entspricht der Vereinigung. Die Entscheidung wird durch einen zusätzlichen Zustand P innerhalb von *regionB2* realisiert, der nach Aktivierung ebenfalls durch Abschlussereignis wieder verlassen wird und lediglich der Abfrage der Variable dient. Semantisch ist der alternative Zustandsautomat nicht absolut äquivalent, denn der Ablauf besteht dadurch aus mehreren „Run-to-completion“-Schritten – allerdings spielt dies meist eine untergeordnete Rolle.

[5] Inter-Level-Transitionen sind Transitionen, bei denen sich Quell- und Zielzustand in unterschiedlichen UML-Regionen befinden (vgl. [LRV94]).

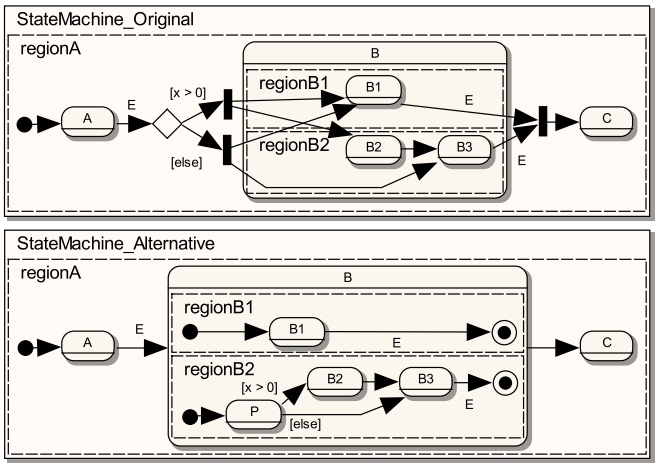


Abbildung 7.3: Alternative UML-Zustandsdiagramme

Annahmen

Bei der vorgestellten Vorgehensweise wird im Allgemeinen jede Transition in eine GTR übersetzt. Allerdings hat auch der Trigger der Transition einen Einfluss auf die erzeugte GTR, z. B. das Graphmuster eines Sensortriggers bzw. des zugehörigen Sensors. Aus diesem Grund ist es problematisch, wenn mehrere (alternative) Trigger für eine Transition modelliert werden, was von UML unterstützt wird. Es soll daher angenommen werden, dass für jede Transition maximal ein Trigger modelliert wird. Praktisch ist dies aber kein Problem, da jede Transition mit mehreren Triggern äquivalent auf mehrere UML-Transitionen mit jeweils einem Trigger und ansonsten identischen Eigenschaften (Wächter, Effekte etc.) abgebildet werden kann.

Als *uml::Behavior*, d. h. Effekte von Transitionen und „entry“/„exit“-Verhalten von Zuständen, sind lediglich AML-Aktionssequenzen erlaubt (eine Einschränkung). Die darin spezifizierte Reihenfolge, auch in Verbindung mit „entry“-/„exit“-Verhalten, wird im Rahmen der generierten GTRs aber nicht genau eingehalten. Die Verarbeitung der modellierten Aktionssequenzelemente beginnt immer mit den *CreateAction*-Elementen und endet mit den *SendMessageAction*-Elementen, wobei die Reihenfolge innerhalb der einzelnen Gruppen aber unverändert bleibt. Da GTP-GTRs genutzt werden, könnte zwar eine beliebige Reihenfolge realisiert werden, allerdings wären die entstehenden GTPs unnötig komplex. Sollte dennoch eine andere Reihenfolge benötigt werden, so besteht in der Regel auch die Möglichkeit, Zwischenzustände zu modellieren. Ebenso wären Pseudozustände wie die „Entscheidung“ denkbar. Wie weiter oben erwähnt, werden diese im präsentierten Ansatz zur Übersetzung aber nicht berücksichtigt.

Aufgrund der später vorgestellten Übersetzung im Falle von zusammengesetzten Zuständen (siehe Abschnitt 7.3.3, S. 244), sind *SendMessageAction*-Elemente in „entry“- und „exit“-Verhalten von Zuständen verboten. Gleiches gilt für die

Effekte von Transitionen, die von einem Initialzustand ausgehen oder an einem Terminierungsknoten enden. Zum einen kann es ohne diese Einschränkungen zu Unstimmigkeiten bei der Verarbeitungsreihenfolge von Zustandsübergängen kommen.[6] Zum anderen ist die Nachrichtenverarbeitung von der Existenz des Senders abhängig. Im Falle eines Terminierungsknotens als Ziel würde der Sender allerdings zerstört werden, bevor die Nachricht verarbeitet wird.

Darüber hinaus dürfen *CreateAction*-Elemente nicht im „exit"-Verhalten von Zuständen modelliert werden. Dadurch können einige Übersetzungsregeln in Verbindung mit zusammengesetzten Zuständen kompakter dargestellt werden.

In den Übersetzungsregeln werden Graphmuster verwendet, die im AML/GT-Modell als Zeichenkette eingebettet sind, z. B. in Attribut *stateInvariant* eines Zustands. Dabei wird davon ausgegangen, dass die abgefragten Graphmuster auch die impliziten Teile enthalten, d. h. die Teile, die feststehen und daher nicht angegeben werden müssen, z. B. die Zustandsautomaten-Eigentümerhyperkante mit Bezeichner *owner*. Die impliziten Teile werden in Abschnitt 6.5.1, S. 201, und Abschnitt 6.5.3, S. 204, beschrieben.

Abschließend sei kurz erwähnt, dass der KSG durch mögliche Aufrufe von Methoden nicht verändert werden darf, zumindest nicht, wenn dies die aktuelle Verarbeitung eines Ereignisses beeinträchtigt. Auch eine mögliche Optimierung der Neuberechnung von internen Ereignissen (vgl. Abschnitt 4.3, S. 111) kann davon abhängig sein.

7.3.2 Generierung von Zustandsklassen

Auch für die modellierten Zustandsautomaten muss Code generiert werden. Als Grundlage für die Generierung kann das Zustands-Entwurfsmuster [GHJJ96] verwendet werden, das vorsieht, für jeden Zustand eine Klasse zu erzeugen. Zusätzlich können Klassen für Regionen generiert werden. Um einen Eindruck über die Beispiel-Implementierung zu vermitteln, wird in Listing C.2, S. 285, ein Code-Ausschnitt für eine generierte Zustandsklasse gezeigt.

Dadurch ist es möglich, die modellierte Struktur in das Laufzeitsystem zu übertragen, d. h., zur Laufzeit kann geprüft werden, welche untergeordneten Regionen und Zustände ein zusammengesetzter Zustand besitzt. Dadurch ist bspw. auch die Abfrage möglich, ob alle untergeordneten Regionen bereits den finalen Zustand erreicht haben (vgl. *isSubStateActive*, Abschnitt 5.5, S. 178). Diese Funktionalität wird im Folgenden für die Prüfung benötigt, ob ein Abschlussereignisses eintritt.

Wichtig sind v. a. auch die Klassen für Animationszustände, denn es kann Code generiert werden, der die modellierten Animationsanweisungen implementiert. In den generierten Code-Abschnitten müssen entsprechende Animatorinstanzen erzeugt und den angegebenen Attributen zugewiesen werden (Aktivierung).

[6] Genauer gesagt, kann es zur Verarbeitung von Ereignissen kommen, ausgelöst durch die Aktivierung von Nachrichtensensorinstanzen, obwohl noch nicht alle Initialzustände von neu betretenen Regionen verlassen wurden.

7.3.3 Generierung von Graphtransformationsregeln

Für die Erzeugung der Ereignis-GTRs eines AAS/GTs müssen die Transitionen der Zustandsautomaten analysiert werden. Nach einigen Hilfsdefinitionen und Anmerkungen, wird ein Überblick über die Entstehung und Zusammensetzung der GTRs präsentiert. Danach werden die grundlegenden Übersetzungsregeln und spezielle Aspekte (*inv broken*-Trigger, zusammengesetzte Zustände etc.) beschrieben.

Hilfsdefinitionen
Um die Übersetzungsregeln kompakter darstellen zu können, seien folgende Operationen für Modellelemente verfügbar. Sie sind als OCL-Ausdrücke definiert und werden verwendet, als wären sie Teil des UML- bzw. AML-Metamodells. Abschnitt B.3, S. 281, zeigt die genaue Spezifikation der Operationen:

- *uml::Vertex::smMed()* gibt den *Zustandsautomateneigentümer* des gegebenen Knotenpunkts zurück (Listing B.5/2).
- *uml::Transition::actSeq()* gibt die Folge der Aktionssequenzelemente der Transition zurück, d. h. die Aktionssequenzelemente (in gegebener Reihenfolge) aus dem „exit“-Verhalten des Quellzustands, aus dem Effekt der Transition und aus dem „entry“-Verhalten des Zielzustands (Listing B.5/8).
- *uml::Region::initTr()* gibt die *initiale Transition* einer Region zurück, d. h. die Transition, die den Initialzustand der Region verlässt (Listing B.5/24).
- *uml::Vertex::getDepth()* gibt die Verschachtelungstiefe eines Knotenpunkts zurück (Listing B.5/30).
- *aml::CreateAction::isForEdge()* gibt zurück, ob ein *CreateAction*-Element eine Hyperkanten-Medienkomponente erstellt (Listing B.5/37).
- *aml::CreateAction::getLabel()* gibt den Bezeichner der zu erstellenden Komponentenhyperkante zurück (Listing B.5/43).
- *aml::Region::isValidFor(mediaComp : aml::MediaComponent)* gibt zurück, ob die Zustandsautomaten der übergebenen Hyperkanten-Medienkomponente *mediaComp* die Region enthält, wobei Zustandsautomaten der Basismedienkomponenten und der abgeleiteten Medienkomponenten berücksichtigt werden (Listing B.5/48).
- *aml::MediaComponent::allRegions()* gibt alle Regionen einer Hyperkanten-Medienkomponente zurück, wobei die Regionen von Zustandsautomaten der Basismedienkomponenten ebenfalls berücksichtigt werden (Listing B.5/57).

Zusätzliche Mengen von UML-Elementen, die benötigt werden, sind:

- $\mathcal{CST}$ = *objectsOfType(uml::Constraint)*,
- $\mathcal{TE}$ = *objectsOfType(uml::TimeEvent)*,

- $\mathcal{US}$ = *objectsOfType(aml::UserSensor)*,
- $\mathcal{MS}$ = *objectsOfType(aml::MessageSensor)*,
- $\mathcal{SA}$ = *objectsOfType(aml::SendMessageAction)*,
- $\mathcal{CA}$ = *objectsOfType(aml::CreateAction)->select(isForEdge())*,
- $\mathcal{RG}$ = *objectsOfType(uml::Region)*,
- $\mathcal{VX}$ = *objectsOfType(uml::Vertex)*,
- $\mathcal{ST}$ = *objectsOfType(uml::State)*,
- $\mathcal{TR}$ = *objectsOfType(uml::Transition)*.

Außerdem werden identische Abbildungen benötigt:

- id_X bezeichne für eine gegebene Menge X die identische Abbildung auf X, die definiert ist durch: $id_X : X \to X\ ,\ x \mapsto x\ .$ △

Zur Verwendung von Graphmustern, Triggern, Aktionssequenzelementen und Anwendungsbedingungen stehen die Funktionen der folgenden Hilfsdefinitionen zur Verfügung. Diese Funktionen wandeln im Modell befindliche UML-Zusicherungen (Zeichenketten innerhalb von *uml::OpaqueExpression*-Elementen) in eine verwendbare Form um. Zurückgegeben wird entweder ein Hypergraph (bzw. ein Graphmuster) oder ein sogenannter *Bedingungs-Code* im Falle von funktionalen Anwendungsbedingungen.

Ein Bedingungs-Code gehöre zu einer systemabhängigen, textuellen Sprache (vgl. Abschnitt 6.5.3, S. 205). Zulässige Wörter der Sprache seien durch die Menge $\mathcal{STRC}$ beschrieben, wobei ein Bedingungs-Code $strc \in \mathcal{STRC}$ auch mehrere funktionale Anwendungsbedingungen enthalten kann. Die leere Zeichenkette $\varepsilon \in \mathcal{STRC}$ bedeute, dass keine funktionale Anwendungsbedingung spezifiziert wurde.

Hilfsdefinitionen
Bei gegebenem Markierungsalphabet mit Attributen $\mathcal{LV} = (\mathcal{L}, \mathcal{V})$ werden folgende Funktionen verwendet:

- $gpattern : \mathcal{CST} \to \mathcal{H}_{\mathcal{L}}$ ordnet jeder UML-Zusicherung $cst \in \mathcal{CST}$ das dadurch modellierte Graphmuster $H \in \mathcal{H}_{\mathcal{L}}$ zu. Dies ist entweder das Invarianten-Graphmuster bei UML-Zuständen oder das Bedingungs-Graphmuster von Wächtern bzw. Sensoren.
  $gpatternR : \mathcal{CST} \times \mathcal{TR} \to \mathcal{H}_{\mathcal{L}}$ und $gpatternL : \mathcal{CST} \times \mathcal{TR} \to \mathcal{H}_{\mathcal{L}}$ sind ähnlich definiert. Allerdings berücksichtigen diese Funktionen zusätzlich eine Transition $tr \in \mathcal{TR}$. Sie werden verwendet, um die Bezeichner des Graphmusters gemäß zugeordneter Übereinstimmungs-Anweisungen anzugleichen. Die Funktion $gpatternR$ bzw. $gpatternL$ verwendet dabei Attribut *rhs* bzw. *lhs* der Transitionserweiterung *Correspondence* (vgl. Abschnitt 6.5.1, S. 202, und Abschnitt 6.5.3, S. 204).

- $gcons : \mathcal{CST} \to \mathcal{STRC}$ ordnet jeder UML-Zusicherung $cst \in \mathcal{CST}$ den dadurch modellierten Bedingungs-Code $strc \in \mathcal{STRC}$ zur Spezifikation von funktionalen Anwendungsbedingungen zu. Diese dürfen für Wächter von Transitionen oder Bedingungen von Sensoren modelliert werden.

- $gnacs : \mathcal{CST} \to \mathbf{P}(\mathcal{H}_\mathcal{L} \times \mathcal{STRC})$ ordnet jeder UML-Zusicherung $cst \in \mathcal{CST}$ die Menge der dadurch modellierten Spezifikationen von negativen Anwendungsbedingungen zu. Die Elemente der Menge bestehen aus jeweils einem Graphmuster $H \in \mathcal{H}_\mathcal{L}$ mit zugehörigem Bedingungs-Code $strc \in \mathcal{STRC}$ zur Spezifikation von funktionalen Anwendungsbedingungen. Diese dürfen für Wächter von Transitionen oder Bedingungen von Sensoren modelliert werden.

- $gcreate : \mathcal{CA} \to \mathcal{H}_\mathcal{L}$ ordnet jedem *CreateAction*-Element $ca \in \mathcal{CA}$ den dadurch modellierten elementaren Hypergraphen $H \in \mathcal{H}_\mathcal{L}$ zu, der aus der von ca zu erzeugenden Komponentenhyperkante und ihren Knoten besteht.

- $\nearrow : \mathcal{H}_\mathcal{L} \to \mathcal{H}_\mathcal{L}$ ordnet jedem Hypergraphen $H \in \mathcal{H}_\mathcal{L}$ einen Hypergraphen $H' \in \mathcal{H}_\mathcal{L}$ zu, der dieselben Elemente wie H, allerdings keine Mehrzweck-Hyperkanten enthält:

$$H' = (E_H|_\mathcal{X}, V_H, vis_H|_{E_H|_\mathcal{X}}, lab_H|_{E_H|_\mathcal{X}}) \text{ mit } \mathcal{X} = \mathcal{CMP} \cup \mathcal{REL} \ .$$

Bei gegebener LHS $L \in \mathcal{H}_\mathcal{L}$ einer einfachen GTR wird außerdem folgende Funktion verwendet:

- $cscons_L : \mathcal{STRC} \to \mathbf{P}(\mathcal{CON}_{L,\mathcal{LV}})$ ordnet jedem Bedingungs-Code $strc \in \mathcal{STRC}$ die dadurch spezifizerte Menge funktionaler Anwendungsbedingungen zu. △

Auf die Erstellung der funktionalen Anwendungsbedingungen auf Grundlage von Bedingungs-Codes soll nicht näher eingegangen werden. Wie bereits erwähnt, ist die Übersetzung abhängig von dem verwendeten GTS und von der im AML/GT-Modell gewählten Sprache, die für funktionale Anwendungsbedingungen eingesetzt wird. Für gewöhnliche funktionale Anwendungsbedingungen wird in der Beispiel-Implementierung (unter Einsatz von DiaMeta) und in den Beispielen dieser Arbeit eine an Java angelehnte Syntax eingesetzt, mit der boolesche Ausdrücke formuliert werden können. Die Zeichenketten können daher direkt als Code verwendet werden, wobei der Code für Zugriffe auf Attribute, Referenzen und Methoden meist angepasst werden muss. Die Syntax für Pfad-Anwendungsbedingungen orientiert sich an der in Abschnitt 3.2.4, S. 75, vorgestellten Syntax.

In einigen der folgenden Übersetzungsregeln werden $\langle\!\langle \ldots \rangle\!\rangle$-Blöcke dargestellt. Darin sind Ausdrücke enthalten, welche die Funktionsweise des generierten Elements zur Laufzeit veranschaulichen. Verwendet wird einfacher Java-Code, wie er auch im Rahmen der Beispiele genutzt wird. Die darin häufig verwendete Variable *owner* referenziert immer das Objekt, das der Komponentenhyperkante $m_E(owner)$ zugeordnet wird, wobei Match m im jeweiligen Kontext immer

gegeben ist. Die Blöcke können zudem zusätzliche {{...}}-Blöcke enthalten. Diese enthalten Code-Anteile, die zum Zeitpunkt der Übersetzung aus dem angegebenen Konstrukt generiert werden müssen.

Vereinigung und Schnitt von Graphmustern

Für die Vereinigung oder den Schnitt der einzelnen Graphmuster im AML/GT-Modell ist entscheidend, welche Knoten und Hyperkanten einander entsprechen. Zwei Elemente entsprechen dann einander, wenn der gleiche Bezeichner (ungleich der leeren Zeichenkette) verwendet wird. Im Falle von Hyperkanten müssen beide Elemente außerdem dieselbe Markierung haben. Andernfalls gilt dies als Modellierungsfehler. Eine Ausnahme von dieser Regel wird weiter unten behandelt. Elemente ohne Bezeichner entsprechen keinem Element eines anderen Graphmusters. Nur im Falle von Knoten gilt diese Regel nicht immer: falls zwei Hyperkanten aus unterschiedlichen Graphmustern aufgrund ihres Bezeichners einander entsprechen müssen, so müssen auch besuchte Knoten einander entsprechen, unabhängig von den Bezeichnern der Knoten. Falls die Knoten in den beiden Graphmustern dennoch unterschiedlich bezeichnet werden, sind diese Bezeichner im gesamten Kontext als identisch zu betrachten.

Da Typhierarchien unterstützt werden, was im verwendeten Formalismus nicht abgebildet wird (vgl. Abschnitt 3.1.2, S. 54), ist es unter bestimmten Umständen möglich, dass unterschiedliche Typen für Hyperkanten mit identischem Bezeichner, und damit evtl. auch unterschiedlicher Anzahl von Knoten, verwendet werden dürfen. Die Typen müssen allerdings hierarchisch miteinander in Beziehung stehen. Obwohl auch andere, komplexere Fälle möglich sind, soll nur ein Spezialfall erlaubt sein: befindet sich eine Hyperkante mit Bezeichner x sowie Markierung A in einem Invarianten-Graphmuster und eine Hyperkante mit Bezeichner x sowie Markierung B in einem zugehörigen Bedingungs-Graphmuster, so muss gelten $A = B$ oder $A \sqsubset B$. Im Rahmen der Vereinigung bzw. des Schnitts soll A verwendet werden, da dieser als „konkreter Typ“ der Hyperkante angenommen wird.

Elemente, die keinen injektiven Match benötigen, werden in den Graphmustern gestrichelt hervorgehoben. Auch dieses Konzept wird nicht in den gegebenen Formalismen abgebildet (vgl. Abschnitt 3.2.1, S. 67). Müssen derartige Elemente aus verschiedenen Graphmustern vereint werden und ist die Forderung nach Injektivität unterschiedlich, so wird im vereinigten Graphmuster Injektivität gefordert.

Überblick

Im folgenden Überblick werden die einzelnen Teilschritte aufgelistet, die bei einem Zustandsübergang in gezeigter Reihenfolge durchgeführt werden sollen. Einige dieser Teilschritte sind jedoch an Bedingungen geknüpft (siehe Tab. 7.1), die später noch genauer erläutert werden:

Schritt	GTR / GTP	Bedingung
(1)	gtp_{tr}^{leav}	zusammengesetzter Zustand ist Quelle
(2)	gtp_{tr}^{rmv}	*inv broken*-Trigger
(3) – (8)	$fgtr_{tr}^{main}$	(3) ist im Falle eines *inv broken*-Triggers speziell aufgebaut, (5) und (8) nur bei Animationszuständen als Quelle/Ziel
(9)	gtp_{tr}^{term}	Terminierungsknoten ist Ziel
(10)	gtp_{tr}^{init}	Komponentenhyperkanten wurden erzeugt oder ein zusammengesetzter Zustand ist Ziel
(11)	gtp_{tr}^{send}	*SendMessageAction*-Elemente

Tabelle 7.1: Abarbeitung der GT-Teilschritte und Bedingungen

(1) Verlassen untergeordneter Zustände des Quellzustands,

(2) Entfernen der Mehrzweck-Hyperkanten des Invarianten-Graphmusters des Quellzustands aufgrund einer Verletzung der Invarianten,

(3) Entfernen/Hinzufügen der Mehrzweck-Hyperkanten des Invarianten-Graphmusters des Quellzustands/Zielzustands,

(4) Erzeugung von Hyperkanten-Medienkomponenten aufgrund der *CreateAction*-Elemente aus dem Effekt der Transition (a) und aus dem „entry“-Verhalten des Zielzustands (b), d. h., es werden entsprechende Komponentenhyperkanten inkl. Knoten hinzugefügt,

(5) Deaktivierung der Animatorinstanzen des Quellzustands inkl. Aktualisierung der Attributwerte,

(6) Änderung der sogenannten Zustandsattribute und Zeitpunktattribute,

(7) Ausführung aller *SetAttributeAction*-Elemente, *CallMethodAction*-Elemente und *CreateAction*-Elemente, die keine Hyperkanten-Medienkomponenten erzeugen, aus dem „exit“-Verhalten des Quellzustands (a), aus dem Effekt der Transition (b) und aus dem „entry“-Verhalten des Zielzustands (c),

(8) Erzeugung und Aktivierung der Animatorinstanzen des Zielzustands,

(9) Entfernen der Zustandsautomaten-Eigentümerhyperkante,

(10) Schalten von allen sogenannten initialen Transitionen in den Regionen des (zusammengesetzten) Zielzustands und den Regionen erzeugter Hyperkanten-Medienkomponenten,

(11) Ausführung aller *SendMessageAction*-Elemente, was zu weiteren Zustandsübergängen führt.[7]

[7] Dies stellt bereits den nächsten Schritt im Sinne der *Run-to-completion*-Semantik dar (vgl. Abschnitt 5.3.2, S. 130).

Die angegebenen Teilschritte werden in einer GTP-GTR für den jeweiligen Zustandsübergang gekapselt. Folgende Übersetzungsregel zeigt die vollständige Zusammensetzung einer solchen GTR. Einige Teile, wie bspw. die Aktivierung und Deaktivierung von Animatorinstanzen (vgl. Abschnitt 5.4.3, S. 146ff), werden im Rahmen von Methoden vollzogen, die von der GTR aufgerufen werden.

Übersetzungsregel 7.2 (GTP-GTR für eine Transition)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Die *GTP-GTR* $pgtr_{tr} \in \mathcal{PG}_{\mathcal{LV}}$ *der Transition* tr ist gegeben durch

$$pgtr_{tr} = (fgtr_{tr}, gtp_{tr}) = ((egtr^{\emptyset}_{L_{tr}}, acr^{\emptyset}, \emptyset, \emptyset), gtp_{tr}) \ .$$

Die darin verwendete *GTP* $gtp_{tr} \in \mathcal{GTP}_{R_{tr},\mathcal{LV}}$ *zur Ausführung der Teilschritte der Transition* tr, wobei definitionsgemäß gilt $R_{tr} = L_{tr}$, hat den grundlegenden Aufbau:

$$\begin{aligned}
gtp_{tr} &= gtp^{pre}_{tr} \ ; \ pgtr^{main}_{tr}(id_{L_{tr}}) && \text{mit} \\
pgtr^{main}_{tr} &= (fgtr^{main}_{tr}, gtp^{post}_{tr}) && \text{und} \\
id_{L_{tr}} &= (id_{V_{L_{tr}}}, id_{E_{L_{tr}}}) && \text{, wobei} \\
fgtr^{main}_{tr} &= (egtr^{main}_{tr}, acr^{main}_{tr}, CONS^{main}_{tr}, NACS^{main}_{tr}) && \text{mit} \\
egtr^{main}_{tr} &= (L^{main}_{tr}, K^{main}_{tr}, R^{main}_{tr}) = (L_{tr}, K^{main}_{tr}, R^{main}_{tr}) && .
\end{aligned}$$

Die enthaltenen GTPs sind wie folgt aufgebaut:

$$\begin{aligned}
gtp^{pre}_{tr} &= (gtp^{leav}_{tr} \ ; \ gtp^{rmv}_{tr}) && \text{und} \\
gtp^{post}_{tr} &= (gtp^{term}_{tr} \ ; \ gtp^{init}_{tr} \ ; \ gtp^{send}_{tr}) && .
\end{aligned}$$

Die Zustandsautomaten-Eigentümerhyperkante im Rahmen von tr werde in den Hypergraphen außerdem mit *owner* bezeichnet. Dabei gilt $owner \in E_{L_{tr}}$.△

Die in GTP-GTR $pgtr_{tr}$ enthaltene erweiterte GTR $fgtr_{tr}$ führt keinerlei Änderungen am KSG durch. Die eigentlichen Änderungen werden erst durch das eingebettete GTP gtp_{tr} vollzogen. Vereinfacht dargestellt, wird durch gtp_{tr} (und damit $pgtr_{tr}$) zunächst gtp^{pre}_{tr}, danach $pgtr^{main}_{tr}$ (der Kern der GT) und schließlich gtp^{post}_{tr} angewendet. Dabei sind die LHS der übergeordneten GTP-GTR $pgtr_{tr}$ und der internen GTP-GTR $pgtr^{main}_{tr}$ identisch, d. h. $L_{tr} = L^{main}_{tr}$. Die Anwendung von $pgtr^{main}_{tr}$ muss außerdem mit demselben Match erfolgen, wie durch Anwendung von $pgtr_{tr}$ vorgegeben, wofür im Rahmen des GTPs der Identitätsmorphismus $id_{L_{tr}}$ genutzt wird.

Tab. 7.1 zeigt die Zuordnung der Teilschritte zu den einzelnen GTRs und GTPs innerhalb von gtp_{tr}. In der Spalte „Bedingung“ wird ein Kriterium genannt, das erfüllt werden muss, damit durch den entsprechenden Schritt überhaupt GTs durchgeführt werden. Wird das Kriterium nicht erfüllt, kann für das jeweilige GTP auch $gtp^{\emptyset}$ angenommen werden. Weitere Details zu den einzelnen Komponenten von gtp_{tr} werden in den folgenden Unterabschnitten behandelt.

Entstehung einer einfachen Graphtransformationsregel

Den Kern der durchzuführenden GT bildet die einfache GTR $egtr_{tr}^{main}$. Ihre LHS enthält die Zustandsautomaten-Eigentümerhyperkante und weitere Elemente. Diese stellen neben ggf. zusätzlichen Anwendungsbedingungen die Situation dar, in denen der Zustandsübergang erfolgen kann. Durch die Anwendung der einfachen GTR sollen die Mehrzweck-Hyperkanten aus dem Invarianten-Graphmuster des Quellzustands entfernt und entsprechende Mehrzweck-Hyperkanten des Zielzustands hinzugefügt werden. Insgesamt werden die Bestandteile der einfachen GTR aus den Invarianten-Graphmustern von Quell- und Zielzustand (inkl. der Invarianten-Graphmuster übergeordneter Zustände) und den Bedingungs-Graphmustern von Wächter und dem Sensor zusammengesetzt, der als Sensortrigger eingesetzt wird. Zusätzlich werden durch die einfache GTR auch neue Komponentenhyperkanten inkl. Knoten hinzugefügt, falls passende *CreateAction*-Elemente in den Effekten der Transition enthalten sind.

Hilfsdefinitionen

Gegeben sei eine Transition $tr \in \mathcal{TR}$. Die Hypergraphen H_{tr}^{base}, H_{tr}^{grd}, H_{tr}^{sens}, $H_{tr}^{statetrg} \in \mathcal{H}_{\mathcal{L}}$ sind gegeben durch

$$H_{tr}^{base} = H_{tr}^{grd} \cup H_{tr}^{sens} \cup H_{tr.source}^{statesup} \quad ,$$

$$H_{tr}^{grd} = gpattern(tr.guard) \quad ,$$

$$H_{tr}^{sens} = \begin{cases} gpatternL(tr.trigger.event.oclAsType(aml::Sensor) \\ \qquad .constraint, tr) \\ \quad \text{, falls } tr.\text{trigger} <> \text{null and} \\ \qquad tr.trigger.event.oclIsKindOf(aml::Sensor) \\ H^{\emptyset} \quad \text{, sonst} \end{cases} \text{ und}$$

$$H_{tr}^{statetrg} = \begin{cases} gpatternR(tr.target.oclAsType(uml::State) \\ \qquad .stateInvariant, tr) \\ \qquad \text{, falls } tr.target \\ \qquad\quad .oclIsKindOf(uml::State) \\ H_{owner:tr.target.smMed()} \quad \text{, sonst} \end{cases} .$$

Gegeben sei ein Knotenpunkt $vx \in \mathcal{VX}$. Die Hypergraphen H_{vx}^{state}, $H_{vx}^{stateself}$, $H_{vx}^{statesup} \in \mathcal{H}_{\mathcal{L}}$ sind gegeben durch

$$H_{vx}^{state} = H_{vx}^{stateself} \cup H_{vx}^{statesup} \quad ,$$

$$H_{vx}^{stateself} = \begin{cases} gpattern(vx.oclAsType(uml::State).stateInvariant) \\ \qquad \text{, falls } vx.oclIsKindOf(uml::State) \\ H_{owner:vx.smMed()} \quad \text{, sonst} \end{cases} \text{ und}$$

$$H_{vx}^{statesup} = \begin{cases} H_{vx.container.state}^{state} \quad \text{, falls } vx.container.state <> null \\ H^{\emptyset} \quad \text{, sonst} \end{cases} . \quad \triangle$$

Übersetzungsregel 7.3 (Einfache GTR für eine Transition)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Falls Transition tr keinen *inv broken*-Trigger verwendet, d. h.

$$tr.trigger.extension\_TriggerEx.isInvBrokenTrigger() = false \ ,$$

ist die *einfache GTR* $egtr_{tr}^{main} \in \mathcal{EG}_{\mathcal{L}}$ *der Transition* tr gegeben durch

$$\begin{aligned}
egtr_{tr}^{main} &= (L_{tr}^{main}, K_{tr}^{main}, R_{tr}^{main}) && \text{mit}\\
L_{tr}^{main} &= H_{tr}^{base} \cup \quad H_{tr.source}^{stateself} \ \cup \nrightarrow (H_{tr}^{statetrg}) && ,\\
K_{tr}^{main} &= H_{tr}^{base} \cup \nrightarrow (H_{tr.source}^{stateself}) \cup \nrightarrow (H_{tr}^{statetrg}) && \text{und}\\
R_{tr}^{main} &= H_{tr}^{base} \cup \nrightarrow (H_{tr.source}^{stateself}) \cup \quad H_{tr}^{statetrg} \ \cup H_{tr}^{crea} && \text{, wobei}\\
H_{tr}^{crea} &= \bigcup_{ca \in \mathcal{CA}_{tr}^{crea}} gcreate(ca) && .
\end{aligned}$$

Die dabei verwendete Menge der *CreateAction*-Elemente $\mathcal{CA}_{tr}^{crea} \subseteq \mathcal{CA}$ der Transition tr für die Erzeugung von Komponentenhyperkanten ist gegeben durch

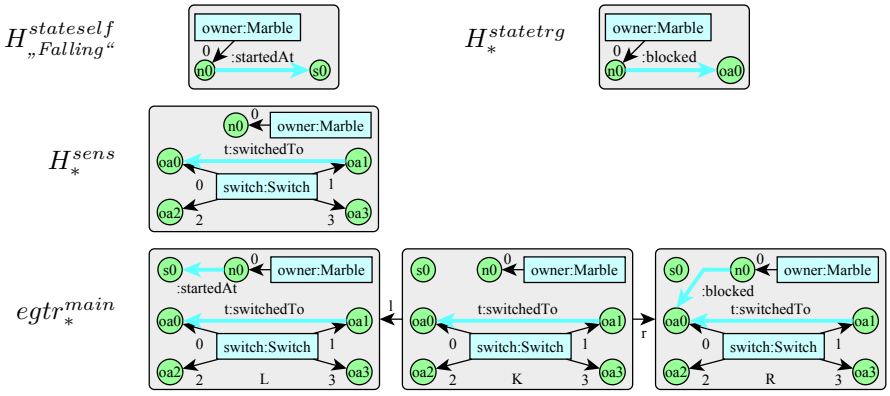
$$\begin{aligned}
\mathcal{CA}_{tr}^{crea} = tr.actSeq()&\text{->select(oclIsKindOf(aml::CreateAction) and}\\
&\text{oclAsType(aml::CreateAction).isForEdge())->asSet()} \ . \qquad \triangle
\end{aligned}$$

Für die Menge der *CreateAction*-Elemente ist wichtig, dass die darin angegebenen Bezeichner voneinander verschieden sein müssen und in den übrigen Graphmustern (abgesehen vom Invarianten-Graphmuster des Zielzustands) auch nicht vorkommen dürfen, d. h.

$$\begin{aligned}
gcreate(ca) \cap H_{tr.source}^{state} &= H^{\emptyset} \quad \wedge\\
gcreate(ca) \cap H_{tr}^{grd} &= H^{\emptyset} \quad \wedge\\
gcreate(ca) \cap H_{tr}^{sens} &= H^{\emptyset} \quad \text{für alle } ca \in \mathcal{CA}_{tr}^{crea} \qquad \text{und}\\
gcreate(ca_1) \cap gcreate(ca_2) &= H^{\emptyset} \quad \text{für alle } ca_1, ca_2 \in \mathcal{CA}_{tr}^{crea} \wedge ca_1 \neq ca_2 \ .
\end{aligned}$$

In den folgenden konkreten Beispielen dieses Kapitels werden Zustände, Regionen, Benutzersensoren und Transitionen eines AML/GT-Modells mit ihrem Namen referenziert: „Name“. Da Namen in den verwendeten Modelle normalerweise eindeutig sind, wird auf die Verwendung vollständig qualifizierter Namen verzichtet. Um für Transitionen einen Namen zu bilden, werden die Namen des Quell- und des Zielzustands in Form von „Quelle/Ziel“ benutzt, wobei Initialzustände den Namen „•“ tragen. Im Falle von Hyperkanten wird der Bezeichner und/oder die Markierung verwendet: „Bezeichner:Markierung“. Werden diese innerhalb eines *CreateAction*- oder *SendMessageAction*-Elements benutzt, wird das Schlüsselwort *create* bzw. *send* angegeben. Zur Abkürzung wird in vielen Beispielen außerdem das $*$-Symbol eingesetzt, dessen Bedeutung zuvor für den jeweiligen Kontext festgelegt wird.

Beispiel 7.3 (Einfache GTR für Transition mit Sensortrigger)
Als Ausgangsmodell der folgenden einfachen GTR einer Transition mit Sensortrigger dienen die AVALANCHE-Diagramme im Anhang A und der darin abgebildete Zustandsautomat *StateMachine_Marble* (S. 274). Betrachtet wird die untere Transition *„Falling/Blocked“* ($= *$). Sie wird durch Sensor *MarbleStop* ausgelöst. Details zu diesem Zustandsübergang werden in Beispiel 6.5, S. 206, genauer erläutert.



Neben der Vereinigung der einzelnen Graphmuster zur einfachen GTR $egtr^{main}_{*}$, zeigt das Beispiel auch die Ergebnisse der modellierten Übereinstimmungs-Anweisungen. Unter Verwendung einer Übereinstimmungs-Anweisung („rhs node“) wird aus dem Invarianten-Graphmuster des Zielzustands der Hypergraph $H^{statetrg}_{*}$. Mit den anderen beiden Übereinstimmungs-Anweisungen („lhs node“) wird aus dem Bedingungs-Graphmuster des Sensors der Hypergraph H^{sens}_{*}. △

Attributänderungen und Anwendungsbedingungen

Die im vorherigen Unterabschnitt vorgestellte einfache GTR $egtr^{main}_{tr}$ ist in einer erweiterten GTR $fgtr^{main}_{tr}$ mit der Attributberechnungsregel acr^{main}_{tr} und den zusätzlichen Anwendungsbedingungen $CONS^{main}_{tr}$ und $NACS^{main}_{tr}$ eingebettet. Wichtig zur Erzeugung der genannten Komponenten sind spezielle Attribute, die für jede Komponentenhyperkante verfügbar sein müssen.

Übersetzungsregel 7.4 (Zustandsattribute und Zeitpunktattribute)
Gegeben sei die Markierung einer Komponentenhyperkante $mc \in \mathcal{CMP}$. Ihr müssen durch das Markierungsalphabet mit Attributen $\mathcal{LV}$ zwei zusätzliche Attribute für jede Region rg ihrer Zustandsautomaten zugeordnet werden, d. h. $rg \in mc.allRegions()$. Durch diese Attribute werden jeweils der aktuelle Zustand innerhalb der Region und der Zeitpunkt gespeichert, an dem dieser Zustand eingenommen wurde. Die Attribute werden daher jeweils *Zustandsattribut* und *Zeitpunktattribut* der Region rg genannt. Als mögliche Werte eines Zustandsattributs gelten alle Zustände der Region (ohne Pseudozustände oder untergeordnete Zustände). Zusätzlich können die Werte *Init*, *NotInRegion*

und *Exceptional* gespeichert werden. Der Wertebereich eines Zeitpunktattributs entspricht $\mathbb{T}$. △

Zustandsattribut und Zeitpunktattribut dienen der Steuerung von Abläufen im Zustandsautomaten und können als Grundlage zahlreicher Berechnungen verwendet werden. Ein Beispiel stellt Attribut *changeTime* aus Beispiel 4.3, S. 107, dar. Es spiegelt die Idee hinter den Zeitpunktattributen wider.

Der Wert *Init* wird gesetzt, falls der Initialzustand der Region eingenommen wurde, was lediglich temporär der Fall sein darf. *NotInRegion* bedeutet, dass kein Zustand der Region aktiv ist. Und *Exceptional* wird gesetzt, falls ein Laufzeitfehler aufgetreten ist, und somit kein korrekter Zustand gesetzt werden konnte.

In Java kann *Class* als Typ für ein Zustandsattribut verwendet werden. In diesem Fall können Klassenobjekte der generierten Zustandsklassen (vgl. Abschnitt 7.3.2, S. 227) als Wert hinterlegt werden (*Zustandsklasse.class*). Dementsprechend müssen auch spezielle Klassen für die Werte *Init*, *NotInRegion* und *Exceptional* existieren.

Bei Erzeugung einer Komponentenhyperkante müssen alle Zustandsattribute und Zeitpunktattribute auf bestimmte Werte gesetzt werden. Für jede Region rg einer neu erzeugten Komponentenhyperkante gilt, dass das zugehörige Zustandsattribut auf den Wert *Init* gesetzt werden muss, falls rg auf oberster Ebene ist, d. h. $rg.stateMachine <> null$. Andernfalls muss der Wert auf *NotInRegion* gesetzt werden. Alle Zeitpunktattribute müssen auf den Wert der Animationszeit *at* zum Zeitpunkt der Erzeugung gesetzt werden. Das Setzen der Attribute von neuen Komponentenhyperkanten muss auch beim „freien Editieren“ geschehen, wenn keine GTRs eingesetzt werden, die im Rahmen dieses Kapitels erläutert werden. Aus diesem Grund sollen keine weiteren Details genannt werden, wie eine entsprechende Erzeugung sichergestellt werden kann.

Im Folgenden werden Zustandsattribute und Zeitpunktattribute nicht direkt referenziert. Stattdessen wird entweder ausführlich beschrieben, welches Attribut abgefragt bzw. verändert wird, oder es werden Methoden des AML-Basisframeworks genutzt. So liefert *getCurrentState* den aktuellen Zustand innerhalb einer Region und *getStateEnteredTime* den Zeitpunkt, wann der Zustand eingenommen wurde (vgl. Abschnitt 5.5, S. 178). Beide Methoden verwenden intern die jeweiligen Attribute, um die entsprechenden Informationen zurückzuliefern.

Übersetzungsregel 7.5 (Attributberechnungsregel für eine Transition)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Die *Attributberechnungsregel* $acr_{tr}^{main} \in \mathcal{ACR}_{R_{tr}^{main},\mathcal{LV}}$ *der Transition* tr muss folgende Schritte durchführen, wobei m den im Kontext von acr_{tr}^{main} verfügbaren Match bezeichne:

(1) Deaktivierung der Animatorinstanzen des Quellknotens, falls der Knoten ein Animationszustand ist, d. h. *tr.source.oclIsKindOf(aml::AnimationState)*.

(2) Setzen des Zustandsattributs der Komponentenhyperkante $m_E(owner)$ für Region *tr.target.container* auf den Zustand *tr.target*.

(3) Setzen der Zustandsattribute der Komponentenhyperkante $m_E(owner)$ für die Regionen *tr.target.oclAsType(uml::State).region* auf den Wert *Init*, falls der Zielknoten ein Zustand ist, d. h. *tr.target.oclIsKindOf(uml::State)*.

(4) Setzen des Zeitpunktattributs der Komponentenhyperkante $m_E(owner)$ für die Region *tr.target.container* auf den Zeitpunkt des Zustandsübergangs.

(5) Setzen der Zeitpunktattribute der Komponentenhyperkante $m_E(owner)$ für die Regionen *tr.target.oclAsType(uml::State).region* auf den Zeitpunkt des Zustandsübergangs, falls der Zielknoten ein Zustand ist, d. h. *tr.target.oclIsKindOf(uml::State)*.

(6) Setzen von Attributen sowie Aufruf von Methoden der Komponentenhyperkante $m_E(owner)$ bzw. des zugeordneten Objekts und Erzeugen gewöhnlicher Objekte gemäß folgender Sequenz von *SetAttributeAction*-, *CallMethodAction*- und *CreateAction*-Elementen (ohne die *CreateAction*-Elemente, die Hyperkanten-Medienkomponenten erzeugen):

```
tr.actSeq()->select( oclIsKindOf(aml::SetAttributeAction) or
  oclAsType(aml::CallMethodAction) or
  (oclIsKindOf(aml::CreateAction) and
    not(oclAsType(aml::CreateAction).isForEdge())))
```

(7) Erzeugung und Aktivierung von Animatorinstanzen für die Animationsanweisungen des Zielzustands, falls dieser ein Animationszustand ist, d. h. *tr.target.oclIsKindOf(aml::AnimationState)*. △

Die genaue Übersetzung der *SetAttributeAction*-, *CallMethodAction*- und *CreateAction*-Elemente (6) in eine Attributberechnungsregel kann dabei nicht angegeben werden, da diese stark von den Möglichkeiten und der konkreten Umsetzung des GTS abhängt. Voraussetzung ist allerdings, dass im Rahmen der Attributberechnungsregel beliebige Methoden von Objekten, die zu den Komponentenhyperkanten gehören, aufgerufen werden können.

Die Deaktivierung, Erzeugung und Aktivierung von Animatoren (1) und (7) kann ebenfalls mit dem Aufruf geeigneter Methoden durchgeführt werden. Dabei kann bspw. die Funktionalität des Basisframeworks in Verbindung mit dem Code verwendet werden, der für die Animationszustände und ihre Animationsanweisungen (vgl. Abschnitt 7.3.2, S. 227) generiert wurde.

Übersetzungsregel 7.6 (Funktionale Anwendungsbedingungen)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Die *funktionalen Anwendungsbedingungen* $CONS_{tr}^{main} \in \mathbf{P}(\mathcal{CON}_{L_{tr}^{main},\mathcal{LV}})$ *der Transition* tr sind gegeben durch

$$CONS_{tr}^{main} = \{con_{tr}^{src}\} \cup \{con_{tr}^{fin}\} \cup CONS_{tr}^{grd} \cup CONS_{tr}^{sens} \quad \text{, wobei}$$
$$con_{tr}^{src}(m, H_A) = \langle\!\langle\ owner.getCurrentState(\{\{tr.source.container\}\}) = \{\{tr.source\}\}\ \rangle\!\rangle$$

für alle attributierten Hypergraphen $H_A \in \mathcal{H}_{\mathcal{LV}}$ und Matches $L_{tr}^{main} \xrightarrow{m} H$,

$$con_{tr}^{fin}(m, H_A) = \begin{cases} \langle\!\langle\ !owner.isSubStateActive(\{\{tr.source\}\})\ \rangle\!\rangle \\ \qquad\qquad \text{, falls } tr.trigger = null \\ wahr \qquad \text{, sonst} \end{cases}$$

für alle attributierten Hypergraphen $H_A \in \mathcal{H}_{\mathcal{LV}}$ und Matches $L_{tr}^{main} \xrightarrow{m} H$,

$$CONS_{tr}^{grd} = cscons_{L_{tr}^{main}}(gcons(tr.guard)) \qquad \text{und}$$
$$CONS_{tr}^{sens} = \begin{cases} cscons_{L_{tr}^{main}}(gcons(tr.trigger.event \\ \quad .oclAsType(aml::Sensor).constraint)) \\ \quad \text{, falls } tr.trigger <> null \text{ and } tr.trigger \\ \quad\quad .event.oclIsKindOf(aml::Sensor) \\ \emptyset \qquad \text{, sonst} \end{cases} . \qquad \triangle$$

Die funktionale Anwendungsbedingung con_{tr}^{src} hat den Zweck, dass die GTP-GTR $pgtr_{tr}^{main}$ (und damit auch $pgtr_{tr}$) nur dann angewendet werden kann, falls sich die Zustandsautomaten-Eigentümerhyperkante im Quellzustand befindet, was durch das abgefragte Zustandsattribut entschieden wird.[8]

Die funktionale Anwendungsbedingung con_{tr}^{fin} stellt sicher, dass der Zustandsübergang im Falle eines Abschlussereignis-Triggers nur durchgeführt werden kann, wenn keine nicht-finalen Zustände innerhalb des Quellzustands (im Falle eines zusammengesetzten Zustands) aktiv sind. Hierfür kann die angegebenen Methode *isSubStateActive* verwendet werden, sofern strukturelle Informationen für Zustände und Animationsanweisungen zur Systemlaufzeit zur Verfügung stehen (vgl. Abschnitt 7.3.2, S. 227). Alternativ kann die notwendige Abfrage auch direkt erzeugt und für con_{tr}^{fin} eingesetzt werden.

Die Mengen $CONS_{tr}^{grd}$ und $CONS_{tr}^{sens}$ stellen die für Wächter und Sensor-Bedingungen modellierten funktionalen Anwendungsbedingungen dar.

[8] Ein verwendetes Invarianten-Graphmuster des Quellzustands ist in den meisten Fällen kein hinreichendes Kriterium zur Bestimmung des Zustands.

Übersetzungsregel 7.7 (Negative Anwendungsbedingungen)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Die *negativen Anwendungsbedingungen* $NACS^{main}_{tr} \in \mathbf{P}(\mathcal{NAC}_{L^{main}_{tr},\mathcal{LV}})$ *der Transition* tr sind gegeben durch

$$NACS^{main}_{tr} = NACS^{grd}_{tr} \cup NACS^{sens}_{tr} \cup NACS^{brk}_{tr} \quad \text{, wobei}$$

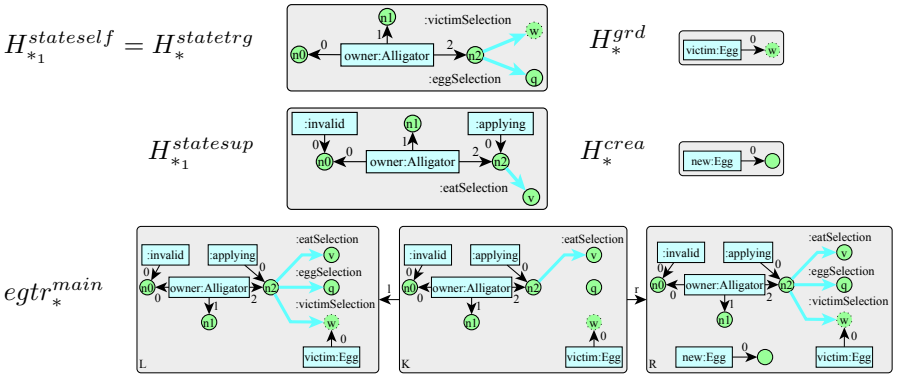
$$NACS^{grd}_{tr} = \{(N, cscons_N(gcons(strc))) \mid N = L^{main}_{tr} \cup N^{part} \ \wedge \ (N^{part}, strc) \in gnacs(tr.guard)\} \quad \text{und}$$

$$NACS^{sens}_{tr} = \begin{cases} \{(N, cscons_N(gcons(strc))) \mid N = L^{main}_{tr} \cup N^{part} \ \wedge \\ \quad (N^{part}, strc) \in gnacs(tr.trigger.event \\ \qquad .oclAsType(aml::Sensor).constraint)\} & \\ \quad \text{, falls } tr.trigger <> null \text{ and } tr.trigger \\ \qquad .event.oclIsKindOf(aml::Sensor) & \\ \emptyset \quad \text{, sonst} & \end{cases} . \qquad \triangle$$

Die Menge der negativen Anwendungsbedingungen $NACS^{brk}_{tr}$ wird in der Übersetzungsregel für Transitionen mit *inv broken*-Trigger erläutert (siehe Übersetzungsregel 7.8, S. 241).

Beispiel 7.4 (Erweiterte GTR für Transition mit „create“-Element)
Als Ausgangsmodell der folgenden erweiterten GTR einer Transition mit *CreateAction*-Element dienen die Alligator Eggs-Diagramme im Anhang A und der darin abgebildete Zustandsautomat *StateMachine_Alligator* (S. 272). Betrachtet wird die linke Transition *„VictimAndHatchingEggSelected/VictimAndHatchingEggSelectedCopyCreated“* $(= * = *_1/*_2)$.

$H^{stateself}_{*_1} = H^{statetrg}_{*}$ H^{grd}_{*} $H^{statesup}_{*_1}$ H^{crea}_{*}



$egtr^{main}_{*}$

$$fgtr^{main}_{*} = (egtr^{main}_{*}, acr^{main}_{*}, CONS^{main}_{*}, NACS^{main}_{*})$$

Die Entscheidung, welche der drei Transitionen mit gleicher Quelle und Ziel zu verwenden ist, wird aufgrund des angegebenen Wächters getroffen, der im

gegebenen Fall durch das Graphmuster H_*^{grd} repräsentiert wird. Durch diesen wird vorausgesetzt, dass eine *Egg*-Komponentenhyperkante als (aktuelles) Opfer der *Alligator*-Hyperkante gewählt wurde (vgl. Knoten w in $H_{*_1}^{stateself}$).

Infolge der gewählten Transition wird durch ein *CreateAction*-Element eine zusätzliche *Egg*-Komponentenhyperkante erzeugt, die in H_*^{crea} dargestellt wird. Zusätzlich ist durch den übergeordneten Zustand *EatingPhase* ein weiteres Graphmuster gegeben: $H_{*_1}^{statesup}$. Die einfache GTR $egtr_*^{main}$ vereint diese Graphmuster und damit Bedingungen und Resultat.

Durch die funktionale Anwendungsbedingung con_*^{src} ist die Anwendung von $fgtr_*^{main}$ und damit $pgtr_*$ nur dann erlaubt, wenn das Zustandsattribut für Region *„reducing“* den Wert *VictimAndHatchingEggSelected.class* enthält:

$$con_*^{src} = \langle\!\langle\ owner.getCurrentState(\text{„reducing“}) = *_1\ \rangle\!\rangle\ .$$

Da der Abschlussereignis-Trigger verwendet wird, ist con_*^{fin} gegben durch

$$con_*^{fin} = \langle\!\langle\ !owner.isSubStateActive(*_1)\ \rangle\!\rangle\ .$$

Diese Bedingung darf allerdings auch weggelassen werden, da der Quellzustand kein zusammengesetzter Zustand ist und somit kein Unterzustand aktiv sein kann.

Die Attributberechnungsregel acr_*^{leav} ändert das oben erwähnte Zustandsattribut auf *VictimAndHatchingEggSelectedCopyCreated*. Für die erstellte *Egg*-Komponentenhyperkante müssen außerdem die zugehörigen Zustandsattribute der Regionen *movementSizeAndCopying*, *hatching* und *recoloring* auf *Init* gesetzt werden, was aber nicht zwangsweise durch acr_*^{leav} geschehen muss (vgl. Abschnitt 7.3.3, S. 237). Alle Zeitpunktattribute, die zu den genannten Zustandsattributen gehören, werden auf den Zeitpunkt des Zustandsübergangs gesetzt, welcher dem aktuellen Wert der Animationszeit *at* entspricht. Weitere Aktionen müssen nicht durchgeführt werden, da keine Animationszustände betroffen sind und keine relevanten Aktionssequenzelemente modelliert wurden. △

inv broken-Trigger

Transitionen mit *inv broken*-Trigger müssen in spezielle GTRs übersetzt werden. Eine solche GTR muss angewendet werden, sobald das Invarianten-Graphmuster des Quellzustands nicht mehr im KSG existiert. Diese Situation kann durch (zusätzliche) negative Anwendungsbedingungen erkannt werden.

Übersetzungsregel 7.8 (Transition mit *inv broken*-Trigger)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Falls Transition tr einen *inv broken*-Trigger verwendet, d. h.

$$tr.trigger.extension\_TriggerEx.isInvBrokenTrigger() = true\ ,$$

ist die einfache GTR $egtr_{tr}^{main} \in \mathcal{EG}_{\mathcal{L}}$ der Transition tr gegeben durch

$$\begin{aligned} egtr_{tr}^{main} &= (L_{tr}^{main}, K_{tr}^{main}, R_{tr}^{main}) && \text{mit} \\ L_{tr}^{main} &= H_{tr}^{base} \cup \nearrow (H_{tr}^{statetrg}) && , \\ K_{tr}^{main} &= H_{tr}^{base} \cup \nearrow (H_{tr}^{statetrg}) && \text{und} \\ R_{tr}^{main} &= H_{tr}^{base} \cup \quad H_{tr}^{statetrg} \cup H_{tr}^{crea} && . \end{aligned}$$

Die Menge der negativen Anwendungsbedingungen $NACS_{tr}^{brk}$ ist in diesem Fall gegeben durch

$$\begin{aligned} NACS_{tr}^{brk} &= \{\ (N_{tr}^{brk}, \emptyset)\ \} && \text{mit} \\ N_{tr}^{brk} &= L_{tr}^{main} \cup H_{tr.source}^{stateself} && . \end{aligned}$$

Das *GTP* $gtp_{tr}^{rmv} \in \mathcal{GTP}_{R_{tr},\mathcal{LV}}$ *zum Entfernen des Invarianten-Graphmusters des Quellzustands der Transition* tr ist gegeben durch

$$\begin{aligned} gtp_{tr}^{rmv} = \ & (egtr_{tr,e_1}^{rmv}(owner \to owner))^{\geq 0} \\ & ;\ \ldots && , \text{wobei} \\ & ;\ (egtr_{tr,e_n}^{rmv}(owner \to owner))^{\geq 0} \\ e_1 \ldots e_n = \ & E_{H_{tr.source}^{stateself}}\text{->}excluding(owner)\text{->}asSequence() \ . \end{aligned}$$

Gegeben sei eine Transition $tr \in \mathcal{TR}$ und eine Mehrzweck-Hyperkante $e \in E_{H_{tr.source}^{stateself}}\text{->}excluding(owner)$. Die einfache *GTR* $egtr_{tr,e}^{rmv} \in \mathcal{EG}_{\mathcal{L}}$ *zum Entfernen der Mehrzweck-Hyperkante* e *bzgl. Transition* tr ist gegeben durch

$$\begin{aligned} egtr_{tr,e}^{rmv} &= (L_{tr,e}^{rmv}, K_{tr,e}^{rmv}, R_{tr,e}^{rmv}) && \text{mit} \\ L_{tr,e}^{rmv} &= (E_{L_{tr,e}^{rmv}}, V_{H_{tr.source}^{stateself}}, vis_{H_{tr.source}^{stateself}}|_{E_{L_{tr,e}^{rmv}}}, lab_{H_{tr.source}^{stateself}}|_{E_{L_{tr,e}^{rmv}}}) && , \\ K_{tr,e}^{rmv} &= (E_{K_{tr,e}^{rmv}}, V_{H_{tr.source}^{stateself}}, vis_{H_{tr.source}^{stateself}}|_{E_{K_{tr,e}^{rmv}}}, lab_{H_{tr.source}^{stateself}}|_{E_{K_{tr,e}^{rmv}}}) && , \\ R_{tr,e}^{rmv} &= K_{tr,e}^{rmv} && , \text{wobei} \\ E_{L_{tr,e}^{rmv}} &= \{owner\} \cup \{e\} && \text{und} \\ E_{K_{tr,e}^{rmv}} &= \{owner\} && . \end{aligned}$$

Falls Transition tr keinen *inv broken*-Trigger verwendet, d. h.

$$tr.trigger.extension\_TriggerEx.isInvBrokenTrigger() = false \ ,$$

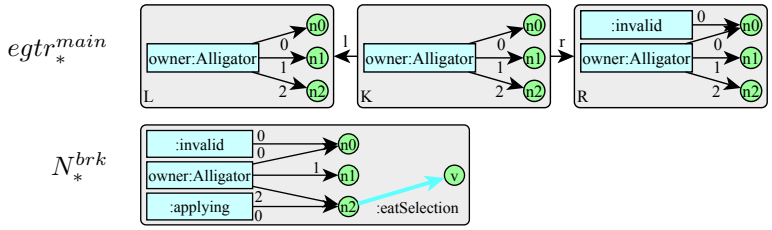
dann gilt $gtp_{tr}^{rmv} = gtp^{\emptyset}$ und $NACS_{tr}^{brk} = \emptyset$. △

Die in diesem Fall entstehende GTP-GTR kann angewendet werden, wenn sich die Zustandsautomaten-Eigentümerhyperkante zwar in einem bestimmten Zustand befindet, was durch das Zustandsattribut festgestellt werden kann, aber das modellierte Invarianten-Graphmuster des Quellzustands nicht im KSG existiert. Dies bedeutet, dass die Invariante verletzt wird. Durch Verwendung der negativen Anwendungsbedingung in $NACS_{tr}^{brk}$ kann eine derartige Verletzung im KSG gefunden werden. Hypergraph N_{tr}^{brk} enthält dabei (im Gegensatz zur LHS L_{tr}^{main}) das gesamte Invarianten-Graphmuster des Quellzustands, d. h., die

GTR ist anwendbar, sobald auch nur eine Mehrzweck-Hyperkante der Invariante fehlt. Bei Anwendung der GTR werden außerdem die verbliebenen Mehrzweck-Hyperkanten der Invariante entfernt. Dies geschieht durch das GTP gtp_{tr}^{rmv}.

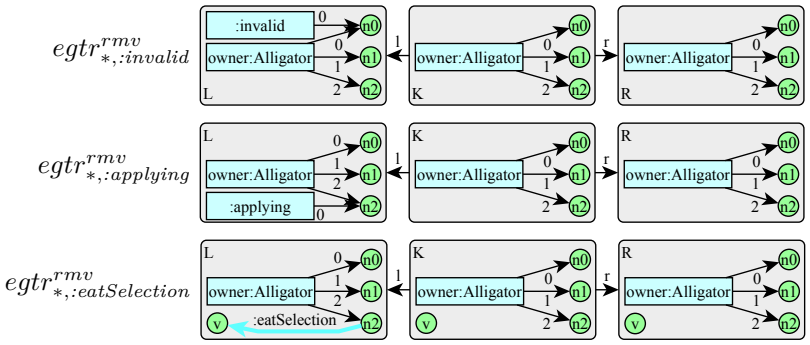
Eine GTP-GTR für eine Transition mit *inv broken*-Trigger wird immer durch interne Ereignisse ausgelöst. Die zugeordnete TCR, die in einer späteren Übersetzungsregel noch beschrieben wird, sorgt dafür, dass die GTP-GTR sofort ausgeführt wird, sobald die Invariante verletzt wird. Es sollte außerdem in Betracht gezogen werden, auch dann eine GTP-GTR nach Übersetzungsregel 7.8 für ein internes Ereignis zu generieren, falls keine Transition mit *inv broken*-Trigger (ohne Wächter) für einen bestimmten Quellzustand (mit relevantem Invarianten-Graphmuster) modelliert wurde. Das so spezifizierte interne Ereignis kann genutzt werden, um zur Laufzeit eine unerwartete Verletzung einer Invarianten zu erkennen. Die in diesem Fall angewendete GTR kann derartige Fehler melden. Außerdem sollte der Quellzustand verlassen und der Wert des entsprechenden Zustandsattributs auf *Exceptional* gesetzt werden.

Beispiel 7.5 (Elemente für Transition mit *inv broken*-Trigger)
Als Ausgangsmodell der folgenden Elemente bzgl. einer Transition mit *inv broken*-Trigger dienen die Alligator Eggs-Diagramme im Anhang A und der darin abgebildete Zustandsautomat *StateMachine_Alligator* (S. 272). Betrachtet wird die Transition *„EatingPhase/DyingPhase“* ($= *$). Sie wird durch einen *inv broken*-Trigger ausgelöst[9], weshalb $egtr_{*}^{main}$ das Invarianten-Graphmuster des Quellzustands nicht enthält. Es ist dafür im Graphmuster der negativen Anwendungsbedingung N_{*}^{brk} enthalten. Das GTP gtp_{*}^{rmv} stellt sicher, dass nach einem entsprechendem Zustandübergang keine Kante des Invarianten-Graphmusters des Quellzustands mehr existiert.



$$gtp_{*}^{rmv} = (pgtr_{*,:invalid}^{rmv}(owner \to owner))^{\geq 0} \\ ; (pgtr_{*,:applying}^{rmv}(owner \to owner))^{\geq 0} \\ ; (pgtr_{*,:eatSelection}^{rmv}(owner \to owner))^{\geq 0}$$

[9] Der Grund für die Modellierung eines *inv broken*-Triggers an dieser Stelle wird in Beispiel 6.4, S. 203, genauer erläutert.



△

Terminierungsknoten

Wenn der Zielknoten einer Transition der Terminierungsknoten ist, gelten besondere Regeln. In diesem Fall muss die Medienkomponenteninstanz zerstört werden. Folglich muss die entsprechende Komponentenhyperkante aus dem KSG entfernt werden. Die Entfernung von Komponentenhyperkanten erfolgt durch gtp^{term}_{tr}.

Übersetzungsregel 7.9 (GTP für Terminierungsknoten)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Falls Transition tr an einem Terminierungsknoten endet, d. h.

$$tr.target.oclIsKindOf(uml::PseudoState)\ and$$
$$tr.target.oclAsType(uml::PseudoState).kind{=}uml::PseudostateKind::terminate\ ,$$

ist das *GTP* $gtp^{term}_{tr} \in \mathcal{GTP}_{R^{main}_{tr},\mathcal{LV}}$ *zum Entfernen der Zustandsautomaten-Eigentümerhyperkante bzgl. der Transition* tr gegeben durch

$$gtp^{term}_{tr} = remove_{tr.source.smMed()}(ehd \rightarrow owner)\ .$$

Andernfalls gilt $gtp^{term}_{tr} = gtp^{\emptyset}$. △

Zusammengesetzte Zustände

In den bisherigen Übersetzungsregeln wurden zusammengesetzte Zustände noch nicht detailliert betrachtet. Zum einen ist es erforderlich, dass untergeordnete Zustände verlassen werden, falls ein übergeordneter Zustand verlassen wird. Die GTP-GTR für eine Transition, dessen Quellzustand untergeordnete Zustände besitzt, muss daher Aktionen durchführen, die zum Verlassen der untergeordneten Zustände notwendig sind.

Zum anderen ist es möglich, dass der Zielzustand einer Transition untergeordnete Zustände bzw. Regionen besitzt. Übersetzungsregel 7.5, S. 237, legt daher fest, dass die Zustandsattribute jeder Region, die dem Zielzustand direkt untergeordnet ist, auf *Init* gesetzt werden, d. h., die jeweiligen Initialzustände werden eingenommen. Nach der Anwendung von $fgtr^{main}_{tr}$, und bevor weitere GTRs aufgrund von *SendMessageAction*-Elementen angewendet (oder andere Ereignisse) verarbeitet werden, müssen diese Initialzustände aber wieder verlassen werden. Diese Vorgehensweise ist ebenfalls für Komponentenhyperkanten

wichtig, die im Rahmen eines Zustandsübergangs erzeugt werden, denn auch bei erzeugten Komponentenhyperkanten werden die Zustandsattribute für die Regionen auf oberster Ebene auf *Init* gesetzt (normalerweise im Rahmen der Konstruktion, vgl. Abschnitt 7.3.3, S. 237).

Durch die folgenden beiden Übersetzungsregeln können GTPs erstellt werden, welche die angesprochenen Aktionen durchführen.

Übersetzungsregel 7.10 (GTP zum Verlassen von Unterzuständen)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Das *GTP* $gtp_{tr}^{leav} \in \mathcal{GTP}_{R_{tr},\mathcal{LV}}$ *zum Verlassen der Unterzustände des Quellzustands der Transition* tr ist gegeben durch

$$gtp_{tr}^{leav} = \begin{cases} gtp_{tr.source,R_{tr}}^{leav} & \text{, falls } tr.source.oclIsKindOf(uml::State) \\ gtp^{\emptyset} & \text{, sonst} \end{cases} .$$

Gegeben seien ein Zustand $st \in \mathcal{ST}$ und die RHS $\overline{R} \in \mathcal{H}_{\mathcal{L}}$ einer zuvor anzuwendenden GTP-GTR. Das *GTP* $gtp_{st,\overline{R}}^{leav} \in \mathcal{GTP}_{\overline{R},\mathcal{LV}}$ *zum Verlassen aller Unterzustände des Zustands* st *bzgl. der RHS* $\overline{R}$ ist gegeben durch

$$\begin{aligned} gtp_{st,\overline{R}}^{leav} &= gtp_{rg_1,\overline{R}}^{leav} \; ; \; \ldots \; ; \; gtp_{rg_n,\overline{R}}^{leav} && \text{, wobei} \\ rg_1 \ldots rg_n &= st.region\text{->}asSequence() && . \end{aligned}$$

Gegeben seien eine Region $rg \in \mathcal{RG}$ und die RHS $\overline{R} \in \mathcal{H}_{\mathcal{L}}$ einer zuvor anzuwendenden GTP-GTR. Das *GTP* $gtp_{rg,\overline{R}}^{leav} \in \mathcal{GTP}_{\overline{R},\mathcal{LV}}$ *zum Verlassen der Region* rg *bzgl. der RHS* $\overline{R}$ ist gegeben durch

$$\begin{aligned} gtp_{rg,\overline{R}}^{leav} &= pgtr_{st_1}^{leav}(owner \to owner) \mid \ldots \mid pgtr_{st_n}^{leav}(owner \to owner) && \text{, wobei} \\ st_1 \ldots st_n &= rg.subvertex\text{->}select(oclIsKindOf(uml::State)) \\ &\quad .oclAsType(uml::State)\text{->}asSequence() && . \end{aligned}$$

Gegeben sei ein Zustand $st \in \mathcal{ST}$. Die *GTP-GTR* $pgtr_{st}^{leav} \in \mathcal{PG}_{\mathcal{LV}}$ *zum Verlassen des Zustands* st ist gegeben durch

$$\begin{aligned} pgtr_{st}^{leav} &= ((egtr_{st}^{leav}, acr_{st}^{leav}, \{con_{st}^{leav}\}, \emptyset), gtp_{st,R_{st}^{leav}}^{leav}) && \text{mit} \\ egtr_{st}^{leav} &= (L_{st}^{leav}, K_{st}^{leav}, R_{st}^{leav}) \\ &= (H_{st}^{stateself}, \nearrow (H_{st}^{stateself}), \nearrow (H_{st}^{stateself})) && \text{und} \\ con_{st}^{leav}(m, H_A) &= \langle\!\langle owner.getCurrentState(\{\{st.container\}\}) = \{\{st\}\} \rangle\!\rangle \end{aligned}$$

für alle attributierten Hypergraphen $H_A \in \mathcal{H}_{\mathcal{LV}}$ und Matches $H_{st}^{stateself} \xrightarrow{m} H$.

Die *Attributberechnungsregel* $acr_{st}^{leav} \in \mathcal{ACR}_{R_{st}^{leav},\mathcal{LV}}$ *zum Verlassen des Zustands* st muss folgende Schritte durchführen, wobei m den im Kontext von acr_{st}^{leav} verfügbaren Match bezeichne:

(1) Deaktivierung der Animatorinstanzen des Quellzustands, falls der Zustand ein Animationszustand ist, d. h. *st.oclIsKindOf(aml::AnimationState)*.

(2) Setzen des Zustandsattributs der Komponentenhyperkante $m_E(owner)$ für Region *st.container* auf den Wert *NotInRegion*.

(3) Setzen des Zeitpunktattributs der Komponentenhyperkante $m_E(owner)$ für Region $st.container$ auf den Zeitpunkt des Zustandsübergangs.

(4) Setzen von Attributen sowie Aufruf von Methoden der Komponentenhyperkante $m_E(owner)$ bzw. des zugeordneten Objekts gemäß folgender Sequenz von *SetAttributeAction*- und *CallMethodAction*-Elementen, falls $st.exit <> null$:

$$\begin{aligned}&st.exit.oclAsType(aml::ActionSequence).action->select(\\&\quad oclIsKindOf(aml::SetAttributeAction)\ or\\&\quad oclAsType(aml::CallMethodAction))\end{aligned} \quad . \quad \triangle$$

Das GTP gtp^{leav}_{tr} berücksichtigt alle Regionen und Unterzustände des Quellzustands. Dabei wird versucht, jeden Unterzustand st durch GTP-GTR $pgtr^{leav}_{st}$ zu verlassen. Die Anwendung der GTR ist jedoch nur möglich, falls sich die Zustandsautomaten-Eigentümerhyperkante in diesem Zustand befindet, was durch con^{leav}_{st} geprüft wird. Sobald ein Unterzustand auf diese Weise verlassen werden kann, d. h., die GTP-GTR anwendbar ist, müssen andere Zustände derselben Region nicht mehr kontrolliert werden, da immer nur ein Zustand pro Region aktiv sein kann. Durch die GTP-GTR $pgtr^{leav}_{st}$ werden neben der Rekursion alle notwendigen Schritte durchgeführt: das Entfernen der Mehrzweck-Hyperkanten im Invarianten-Graphmuster des jeweiligen Zustands und die Durchführung der Aktionen, die für die Attributberechnungsregel acr^{leav}_{st} gelistet sind.

Übersetzungsregel 7.11 (GTP zum Verlassen der Initialzustände)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Das *GTP* $gtp^{init}_{tr} \in \mathcal{GTP}_{R^{main}_{tr},\mathcal{LV}}$ *zum Verlassen der Initialzustände bzgl. Transition* tr ist gegeben durch

$$\begin{aligned}gtp^{init}_{tr} &= gtp^{init,trg}_{tr}\ ;\ gtp^{init}_{tr,ca_1}\ ;\ \ldots ; gtp^{init}_{tr,ca_n} && \text{, wobei}\\ ca_1 \ldots ca_n &= \mathcal{CA}^{crea}_{tr}\text{->}asSequence() && .\end{aligned}$$

Das *GTP* $gtp^{init,trg}_{tr} \in \mathcal{GTP}_{R^{main}_{tr},\mathcal{LV}}$ *zum Verlassen der Initialzustände des Zielzustands der Transition* tr ist gegeben durch

$$\begin{aligned}gtp^{init,trg}_{tr} = {}& pgtr_{tr_1}(owner \to owner)\\ & ;\ \ldots && \text{, wobei}\\ & ;\ pgtr_{tr_n}(owner \to owner)\end{aligned}$$

$$tr_1 \ldots tr_n = \begin{cases} tr.target.oclAsType(uml::State) \\ \qquad .region.initTr()\text{->}asSequence() \\ \qquad\qquad \text{, falls } tr.target.oclIsKindOf(uml::State) \\ \varepsilon \qquad\qquad \text{, sonst} \end{cases} .$$

Zusätzlich sei ein *CreateAction*-Element $ca \in \mathcal{CA}_{tr}^{crea}$ gegeben. Das *GTP* $gtp_{tr,ca}^{init} \in \mathcal{GTP}_{R_{tr}^{main},\mathcal{LV}}$ *zum Verlassen der Initialzustände einer durch CreateAction-Element ca erstellten Komponentenhyperkante bzgl. der Transition tr* ist gegeben durch

$$gtp_{tr,ca}^{init} = pgtr_{tr_1}(owner \to ca.getLabel()) \; ; \ldots \; ; pgtr_{tr_n}(owner \to ca.getLabel()) \quad \text{, wobei}$$

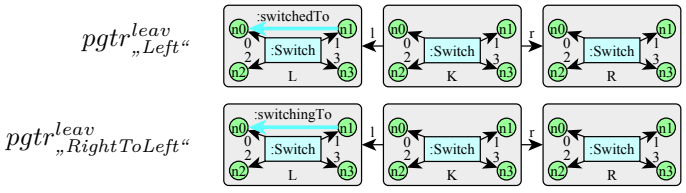
$$tr_1 \ldots tr_n = \begin{cases} ca.type.classifierBehavior.oclAsType(\\ \quad uml::StateMachine).region.initTr()->asSequence() \\ \qquad \text{, falls } ca.type.classifierBehavior <> null \text{ and} \\ \qquad\quad ca.type.classifierBehavior \\ \qquad\quad .oclIsKindOf(uml::StateMachine) \\ \varepsilon \qquad \text{, sonst} \end{cases} . \qquad \triangle$$

Dabei soll angenommen werden, dass die Zeichenkette, die durch *ca.getLabel()* zurückgegeben wird, der jeweils erzeugten Komponentenhyperkante im Kontext des GTPs entspricht, d. h. $ca.getLabel() \in E_{R_{tr}^{main}}$.

Neben der direkten Anwendung von gtp_{tr}^{init} werden Initialzustände auch durch interne Ereignisse verlassen, da an den zugehörigen Transitionen ein Abschlussereignis-Trigger angenommen wird (vgl. Übersetzungsregel 7.16, S. 254). Dies kann wichtig für das „freie Editieren“ sein, wenn neue Diagrammkomponenten platziert werden, welche die Initialzustände der obersten Regionen sofort im Anschluss verlassen sollen. Trotzdem wird auch die direkte Anwendung benötigt, damit Initialzustände verlassen werden, bevor die Anwendung von gtp_{tr}^{send} (eigentlich die Abarbeitung der nächsten internen Ereignisse) aufgrund der *SendMessageAction*-Elemente erfolgt (vgl. Übersetzungsregel 7.2, S. 233).

Beispiel 7.6 (GTP zum Verlassen der Unterzustände)
Als Ausgangsmodell des folgenden GTPs zum Verlassen der Unterzustände des Quellzustands einer Transition dienen die Avalanche-Diagramme im Anhang A und der darin abgebildete Zustandsautomat *StacheMachine_Switch* (S. 274). Betrachtet wird die Transition *„ToLeft/ToRight“*.

$$gtp_{„ToLeft/ToRight“}^{leav} = pgtr_{„Left“}^{leav}(owner \to owner) \mid pgtr_{„RightToLeft“}^{leav}(owner \to owner)$$



Aufgrund von $CONS^{main}_{„Left“}$ bzw. $CONS^{main}_{„RightToLeft“}$ ist die Anwendung von $pgtr^{leav}_{„Left“}$ und $pgtr^{leav}_{„RightToLeft“}$ jeweils nur erlaubt, wenn das Zustandsattribut für die Region *switchingLeft* den Wert *Left* bzw. *RightToLeft* enthält. Die jeweilige Attributberechnungsregel $acr^{leav}_{„Left“}$ bzw. $acr^{leav}_{„RightToLeft“}$ setzt das Zustandsattribut dann auf *NotInRegion* und auch das Zeitpunktattribut wird aktualisiert. Zuvor muss im Falle von Zustand *RightToLeft* die Animatorinstanz deaktiviert werden. Dies hat zur Folge, dass die Bewegung des Hebels an der Position stoppt, an der er sich zum Zeitpunkt des Zustandsübergangs befindet. Weitere GTPs werden nicht aufgerufen, da die beiden Zustände in Region *switchingLeft* keine weiteren Unterzustände haben, d. h. $gtp^{leav}_{„Left“,R^{leav}_{„Left“}} = gtp^{leav}_{„RightToLeft“,R^{leav}_{„RightToLeft“}} = gtp^{\emptyset}$.△

Beispiel 7.7 (GTP zum Verlassen der Initialzustände)
Als Ausgangsmodell des folgenden GTPs zum Verlassen der Initialzustände bzgl. einer Transition dienen die Alligator Eggs-Diagramme im Anhang A und der darin abgebildete Zustandsautomat *StateMachine_Alligator* (S. 272). Betrachtet wird die Transition *„EatingPhase/DyingPhase“* ($= *_1$).

$$\begin{aligned} gtp^{init}_{*_1} &= gtp^{init,trg}_{*_1} \\ &= pgtr_{„\bullet/RotateToBack“}(owner \to owner) \\ &\quad ; pgtr_{„\bullet/RelayoutPending“}(owner \to owner) \end{aligned}$$

In diesem Beispiel werden keine Komponentenhyperkanten erzeugt, weshalb lediglich $gtp^{init,trg}_{*_1}$ verarbeitet werden muss. Anders ist dies bei der linken Transition *„VictimAndHatchingEggSelected/VictimAndHatchingEggSelectedCopyCreated“* ($= *_2$), die eine *Egg*-Komponentenhyperkante erzeugt. Diese muss nach der Erzeugung ihre Initialzustände auf oberster Ebene verlassen.

$$\begin{aligned} gtp^{init}_{*_2} &= gtp^{init}_{*_2,„create\ new:Egg“} \\ &= pgtr_{„\bullet/Static“}(owner \to new) \\ &\quad ; pgtr_{„\bullet/Idle“}(owner \to new) \\ &\quad ; pgtr_{„\bullet/Observe“}(owner \to new) \end{aligned}$$ △

Nachricht senden

Abschließend wird beschrieben werden, wie *SendMessageAction*-Elemente verarbeitet werden. Prinzipiell stellt das Senden einer Nachrichten von einer Diagrammkomponente zur anderen ein internes Ereignis dar. Im präsentierten Ansatz wird der Austausch von Nachrichten allerdings nicht durch interne Ereignisse gemäß AAS/GT abgebildet. Stattdessen werden die entsprechenden GTP-GTRs, die durch Eintreten eines internen Ereignis angewendet werden würden, direkt angewendet. Dies geschieht am Ende der GTP-GTR $pgtr_{tr}$ durch die Anwendung von gtp^{send}_{tr} (vgl. Übersetzungsregel 7.2, S. 233). Bereits vor der Anwendung von gtp^{send}_{tr} gilt das durch $pgtr_{tr}$ verarbeitete Ereignis als abgearbeitet, d. h., der „Run-to-completion“-Schritt inkl. Zustandsübergang ist abgeschlossen (vgl. Abschnitt 5.3.2, S. 130).

An dieser Stelle sei angemerkt, dass die direkte Anwendung anderer GTRs nicht die einzige Möglichkeit zur Übersetzung von „Nachricht senden"-Aktionen darstellt. Bei dieser Vorgehensweise werden jedoch keine zusätzlichen Konstrukte benötigt, die im Rahmen von AAS/GT notwendig wären, und der Ablauf entspricht trotzdem der durch AML festgelegten Semantik. Die Semantik wird sogar konkretisiert, da durch die Anwendungsreihenfolge der GTRs bestimmt wird, in welcher Reihenfolge die Nachrichten gesendet werden, womit die Abarbeitungsreihenfolge (größtenteils) festgelegt ist.

Übersetzungsregel 7.12 (GTP zum Senden von Nachrichten)
Gegeben sei eine Transition $tr \in \mathcal{TR}$. Das *GTP* $gtp_{tr}^{send} \in \mathcal{GTP}_{R_{tr}^{main},\mathcal{LV}}$ *zum Senden der Nachrichten der Transition tr* ist gegeben durch

$$gtp_{tr}^{send} = gtp_{tr,sa_1}^{send} \ ; \ \dots \ ; \ gtp_{tr,sa_n}^{send} \quad \text{, wobei}$$

$$sa_1 \dots sa_n = tr.actSeq()\text{->}select(\ oclIsKindOf(aml::SendMessageAction))\ .oclAsType(aml::SendMessageAction) \quad .$$

Zusätzlich sei ein *SendMessageAction*-Element $sa \in \mathcal{SA}$ gegeben. Das *GTP* $gtp_{tr,sa}^{send} \in \mathcal{GTP}_{R_{tr}^{main},\mathcal{LV}}$ *zum Senden der Nachricht sa für Transition tr* ist gegeben durch

$$gtp_{tr,sa}^{send} = gtp_{ms_1,sa.receiver,R_{tr}^{main}}^{act} \ ; \ \dots \ ; \ gtp_{ms_n,sa.receiver,R_{tr}^{main}}^{act} \quad \text{, wobei}$$

$$ms_1 \dots ms_n = objectsOfType(aml::MessageSensor)\text{->}select(messageType = sa.type)\text{->}asSequence()\text{->}sortedBy(priority) \quad . \qquad \triangle$$

Dabei soll angenommen werden, dass die Zeichenkette *sa.receiver* einer entsprechenden Komponentenhyperkante im Kontext des GTPs entspricht, d. h. $sa.receiver \in E_{R_{tr}^{main}}$. Falls die Zeichenkette auf *null* gesetzt ist, entspricht sie der *Null-Hyperkante* $e^{\emptyset} \notin E_{R_{tr}^{main}}$.

Das GTP gtp_{tr}^{send} geht zunächst schrittweise alle *SendMessageAction*-Elemente durch. Für jedes Element wird ein weiteres GTP benötigt, das alle Nachrichtensensoren des AML/GT-Modells mit entsprechendem Typ durchläuft und für jeden Nachrichtensensor das *GTP zur flüchtigen Aktivierung des Sensors* nutzt.[10] Wie folgende Übersetzungsregel zeigt, ist die äußerste Schicht dieses GTPs unterschiedlich definiert, je nachdem ob die Nachricht an eine bestimmte Komponentenhyperkante oder an alle gesendet wird (Broadcast).

[10] Es werden auch dann alle passenden Nachrichtensensoren durchlaufen, wenn ein Empfänger festgelegt wird und dadurch ein Typ feststeht, der die Menge der infrage kommenden Nachrichtensensoren einschränkt. Die Prüfung auf Kompatibilität wird erst innerhalb der besagten GTP sichergestellt.

Übersetzungsregel 7.13 (GTP zur flüchtigen Aktivierung)
Gegeben seien ein Nachrichten- bzw. Benutzersensor $s \in \mathcal{MS} \cup \mathcal{US}$ und eine Komponentenhyperkante $rcv \in E_{\overline{R}}|_{\mathcal{CMP}}$ aus der RHS $\overline{R} \in \mathcal{H}_{\mathcal{L}}$ einer zuvor anzuwendenden GTP-GTR bzw. die Null-Hyperkante, d. h. $rcv = e^{\emptyset}$, wobei $e^{\emptyset} \notin E_{\overline{R}}$. Das *GTP* $gtp^{act}_{s,rcv,\overline{R}} \in \mathcal{GTP}_{\overline{R},\mathcal{LV}}$ *zur flüchtigen Aktivierung des Sensors* s *von Empfänger* rcv *bzgl. der RHS* $\overline{R}$ ist gegeben durch

$$gtp^{act}_{s,rcv,\overline{R}} = \begin{cases} \text{foreach } ircv : s.sensorOwner \text{ do} \\ \quad gtp^{act}_{s,ircv,\overline{R} \cup H_{ircv:s.sensorOwner},\mathcal{RG}^{top}} & \text{, falls } rcv = e^{\emptyset} \\ gtp^{act}_{s,rcv,\overline{R},\mathcal{RG}^{top}} & \text{, sonst} \end{cases} ,$$

wobei gelten muss $ircv \notin E_{\overline{R}} \ \wedge \ ircv \neq e^{\emptyset}$
und die Menge der *Regionen* $\mathcal{RG}^{top} \subseteq \mathcal{RG}$ *auf oberster Ebene* gegeben ist durch

$$\mathcal{RG}^{top} = \{rg \in \mathcal{RG} \mid rg.state = null\} .$$

Zusätzlich seien die Regionen $\mathcal{RG}' \subseteq \mathcal{RG}$ gegeben. Das *GTP* $gtp^{act}_{s,rcv,\overline{R},\mathcal{RG}'} \in \mathcal{GTP}_{\overline{R},\mathcal{LV}}$ *zur flüchtigen Aktivierung des Sensors* s *von Empfänger* rcv *bzgl. der RHS* $\overline{R}$ *für die Regionen* $\mathcal{RG}'$ ist gegeben durch

$$\begin{aligned} gtp^{act}_{s,rcv,\overline{R},\mathcal{RG}'} &= gtp^{act}_{s,rcv,\overline{R},rg_1} \ ; \ \ldots \ ; \ gtp^{act}_{s,rcv,\overline{R},rg_n} && \text{, wobei} \\ rg_1 \ldots rg_n &= \mathcal{RG}'\text{->}asSequence() && . \end{aligned}$$

Zusätzlich sei eine Region $rg \in \mathcal{RG}$ gegeben. Falls $rg.isValidFor(lab_{\overline{R}}(rcv))$, ist das *GTP* $gtp^{act}_{s,rcv,\overline{R},rg} \in \mathcal{GTP}_{\overline{R},\mathcal{LV}}$ *zur flüchtigen Aktivierung des Sensors* s *von Empfänger* rcv *bzgl. der RHS* $\overline{R}$ *für die Region* rg gegeben durch

$$\begin{aligned} gtp^{act}_{s,rcv,\overline{R},rg} &= ((gtp^{act}_{s,rcv,\overline{R},\mathcal{RG}'_1}{}^{\geq 1}) \mid \ldots \mid (gtp^{act}_{s,rcv,\overline{R},\mathcal{RG}'_n}{}^{\geq 1}) \\ &\qquad \mid pgtr_{tr_1}(b) \mid \ldots \mid pgtr_{tr_n}(b))^{\geq 0} && \text{, wobei} \\ \mathcal{RG}'_1 \ldots \mathcal{RG}'_n &= st_1.region \ldots st_n.region && \text{mit} \\ st_1 \ldots st_n &= rg.subvertex\text{->}select(oclIsKindOf(uml::State)) \\ &\qquad .oclAsType(uml::State)\text{->}asSequence() && , \\ tr_1 \ldots tr_n &= rg.transition\text{->}select(event = s) \\ &\qquad \text{->}asSequence()\text{->}sortedBy(guard = null) && \text{und} \end{aligned}$$

$$b = \begin{cases} (owner \rightarrow rcv) \cup (s.oclAsType(aml::MessageSensor) \\ \qquad .opponentRelation\text{->}at(1) \\ \qquad .opponentName \rightarrow owner) \\ \quad \text{, falls } s.oclIsKindOf(aml::MessageSensor) \\ \qquad and\ s.oclAsType(aml::MessageSensor) \\ \qquad .getOpponents()\text{->}size() = 1 \\ (owner \rightarrow rcv) \\ \quad \text{, sonst} \end{cases} .$$

Andernfalls gilt $gtp^{act}_{s,rcv,\overline{R},rg} = gtp^{\emptyset}$. △

Falls kein Empfänger bestimmt wird, besteht die äußerste Schicht von GTP $gtp^{act}_{s,rcv,\overline{R}}$ aus einer „foreach“-Schleife, die alle Sensoreigentümerinstanzen schrittweise durchgeht. In der nächsten Schicht werden die Regionen der Zustandsautomaten auf oberster Ebene durchlaufen. Davon ausgehend werden rekursiv Transitionen gesucht, die aufgrund eines passenden Sensortriggers geschaltet werden können. Es kommen hierbei nur noch Regionen in Frage, die kompatibel zum Typ der gegebenen Sensoreigentümerinstanz sind. Für die Prüfung wird die Operation *isValidFor* verwendet.

Wird eine entsprechende Transition gefunden, wird versucht die GTP-GTR der Transition anzuwenden. Dabei soll die Sensoreigentümerinstanz im Rahmen der GTP-GTR als *owner* verwendet werden. Im Falle eines Nachrichtensensors mit einem Verursacher wird zusätzlich der Sender der Nachricht für den zu suchenden Match vorgegeben. Der Sender trägt im Rahmen der GTP-GTR dann den Verursachernamen als Bezeichner. In der Übersetzungsregel wird die Zeichenkette in Attribut *opponentName* symbolisch für die entsprechende Komponentenhyperkante genutzt.

Die GTPs sind so aufgebaut, dass in jeder Region maximal eine Transition auf derselben Ebene oder mehrere Transitionen in maximal einem untergeordneten Zustand schaltet. Da die Rekursion in tiefer gelegene Regionen zuerst durchgeführt wird, erhalten die Transitionen darin eine höhere Priorität. Durch die angegebene Sortierung wird außerdem eine zusätzliche Priorisierung der Zustandsübergänge mit Wächter erreicht (vgl. Abschnitt 5.3.2, S. 131).

Beispiel 7.8 (GTP zum Senden von Nachrichten – inkl. „foreach“)
Als Ausgangsmodell des folgenden GTPs zum Senden der Nachrichten einer Transition dienen die Alligator Eggs-Diagramme im Anhang A und der darin abgebildete Zustandsautomat *StateMachine_Alligator* (S. 272). Betrachtet wird die Transition *„EatingPhase/DyingPhase“* $(= *)$.

$$\begin{aligned} gtp^{send}_{*} &= gtp^{send}_{*,send\ DoShrink\ to\ owner}\ ;\ gtp^{send}_{*,send\ DoHatch} \\ &= (pgtr_{\text{„}Static/Shrinking\text{“}}(owner \to owner))^{\geq 0} \\ &\quad ;\ \text{foreach } ircv : Egg \text{ do } (pgtr_{\text{„}PreShake/Shaking\text{“}}(owner \to ircv))^{\geq 0} \end{aligned}$$

Die betrachtete Transition enthält zwei *SendMessageAction*-Elemente. Für beide wird ein entsprechendes Unterprogramm generiert. Für das erste Unterprogramm werden zunächst passende Nachrichtensensoren für Nachricht *DoShrink* gesucht. Gefunden wird lediglich Nachrichtensensor *Shrink*. Der Sensoreigentümer ist *Member*, wobei aufgrund von Vererbung u. a. auch Alligatoren entsprechende Sensorinstanzen besitzen. Dieser Sensor ist nur in der Lage eine einzige Transition zu schalten. Dies ist die Transition *„Static/Shrinking“*, weshalb GTP-GTR $pgtr_{\text{„}Static/Shrinking\text{“}}$ (optional) angewendet wird. Die Nachricht verschickt ein Alligator dabei an sich selbst, wodurch der angegebene Teilmatch $(owner \to owner)$ resultiert. Analog verläuft die Suche bei Nachricht *DoHatch*. Diese Nachricht wird allerdings an alle Medienkomponenteninstanzen gesendet, weshalb *foreach* verwendet wird und als Teilmatch (für *owner*) jeweils die Komponentenhyperkante des Schleifendurchlaufs (*ircv*) vorgegeben wird. △

Beispiel 7.9 (GTP zum Senden von Nachrichten – mit Verursacher)
Als Ausgangsmodell des folgenden GTPs zum Senden der Nachrichten einer Transition dient das Diagramm für B/E-Netze im Anhang A, S. 269. Betrachtet man die Transition *„FiringToTransition/FiringToTransition“* $(= *)$ im Zustandsautomaten *StateMachine_Transition*, so ergibt sich folgendes GTP:

$$gtp_{*}^{send} = (pgtr_{\text{„}Regular/ToTransition\text{“}}(owner \to p, transitionTo \to owner))^{\geq 0}$$

Da der gefundene Nachrichtensensor *TokenToTransition* einen Verursacher modelliert, wird $(transitionTo \to owner)$ als zusätzlicher Teilmatch vorgegeben, d. h., die Komponentenhyperkante des Senders entspricht in der anzuwendenden GTP-GTR $pgtr_{\text{„}Regular/ToTransition\text{“}}$ der Komponentenhyperkante mit Bezeichner *transitionTo*. △

7.3.4 Graphtransformationsregeln für Ereignisse

In diesem Abschnitt werden die Übersetzungsregeln für die noch verbliebenen AAS/GT-Komponenten vorgestellt: interne und externe Ereignistypen inkl. Zuordnung zu den Ereignis-GTRs, Prioritätszuordnung und TCRs. Zur Beschreibung werden die Übersetzungsregeln und Hilfsdefinitionen des vorherigen Abschnitts benötigt.

Externe Ereignisse

Die flüchtige Aktivierung einer Benutzersensorinstanz „von außen“, z. B. durch Klicken auf eine bestimmte Diagrammkomponente, entspricht in AAS/GT einem externen Ereignis mit der Anwendung der zugehörigen Ereignis-GTR.

Übersetzungsregel 7.14 (Externe Ereignistypen)
Die Menge der externen Ereignistypen $\hat{\mathcal{Q}}$ ist gegeben durch die Menge der Benutzersensoren: $\hat{\mathcal{Q}} = \mathcal{US}$. △

Wie die flüchtige Aktivierung der Benutzersensoren und damit das externe Ereignis genau herbeigeführt werden kann, wird durch den Typ und weitere Details des Benutzersensors angegeben. Dies findet keine Entsprechung in AAS/GT und muss daher individuell übersetzt werden. In der Beispiel-Implementierung für DiaMeta werden Benutzersensoren mit den Typen *mouse_click*, *command*, *keyboard_down*, *keyboard_up* und *keyboard_typed* berücksichtigt, d. h., in der Sprachspezifikation können derartige Ereignisse unterschiedlich festgelegt werden. So entstehen bspw. Knöpfe (*mouse_click* oder *command*), die zur flüchtigen Aktivierung einer Benutzersensorinstanz führen, oder der Editor reagiert auf Tastendruck (*keyboard_down* etc.). Um aber eine konkrete Benutzersensorinstanz flüchtig aktivieren zu können, muss zuvor eine passende Diagrammkomponente (die Sensoreigentümerinstanz) gewählt werden.

Zusätzlich existiert ein Spezialfall. Wenn in den Details des Benutzersensors die Zeichenkette „Broadcast“ hinterlegt wird, muss keine Diagrammkomponente zuvor gewählt werden. In diesem Fall werden nacheinander alle Instanzen eines bestimmten Benutzersensors flüchtig aktiviert (vgl. Abschnitt 6.6, S. 209).

Die flüchtige Aktivierung einer Benutzersensorinstanz entspricht in einem AAS/GT der direkten Anwendung einer Ereignis-GTR. Die gewählte Diagrammkomponente wird dabei als Teilmatch zur Anwendung der Ereignis-GTR vorgegeben (für die Komponentenhyperkante *rcv* der LHS). Ein GTP, das passende Zustandsübergänge ermöglicht, wurde bereits in Übersetzungsregel 7.13, S. 250, vorgestellt. Das GTP-GTR in folgender Übersetzungsregeln kapselt dieses GTP und muss im Spezialfall (Broadcast) lediglich die Null-Hyperkante $e^{\emptyset}$ verwenden.

Übersetzungsregel 7.15 (GTP-GTR für Benutzersensor)
Gegeben sei ein Benutzersensor $us \in \mathcal{US}$. Die *GTP-GTR* $pgtr_{us} \in \mathcal{PG}_{\mathcal{LV}}$ *zur flüchtigen Aktivierung des Benutzersensors* us ist gegeben durch

$$pgtr_{us} = (fgtr_{us}, gtp_{us}) = ((egtr^{\emptyset}_{L_{us}}, acr^{\emptyset}, \emptyset, \emptyset), gtp_{us}) \quad \text{, wobei}$$

$$L_{us} = \begin{cases} H^{\emptyset} & \text{, falls } us.details = \text{"}Broadcast\text{"} \\ H_{rcv:us.sensorOwner} & \text{, sonst} \end{cases} \quad \text{und}$$

$$gtp_{us} = \begin{cases} gtp^{act}_{us,e^{\emptyset},L_{us}} & \text{, falls } us.details = \text{"}Broadcast\text{"} \\ gtp^{act}_{us,rcv,L_{us}} & \text{, sonst} \end{cases} . \qquad \triangle$$

Beispiel 7.10 (GTP-GTR für Benutzersensor)
Als Ausgangsmodell des folgenden GTPs zur flüchtigen Aktivierung eines Benutzersensors dient das Diagramm für B/E-Netze im Anhang A, S. 269,. Betrachtet wird der Benutzersensor *„FireAll“* (Broadcast).

$$pgtr_{\text{„}FireAll\text{“}} = ((egtr^{\emptyset}_{H^{\emptyset}}, acr^{\emptyset}, \emptyset, \emptyset), gtp_{\text{„}FireAll\text{“}})$$
$$gtp_{\text{„}FireAll\text{“}} = gtp^{act}_{\text{„}FireAll\text{“},e^{\emptyset},H^{\emptyset}} =$$
$$\text{foreach } ircv : Transition \text{ do}$$
$$(pgtr_{\text{„}Enabled/FiringToTransition\text{“}}(owner \rightarrow ircv))^{\geq 0}$$

Aufgrund des Broadcasts wird die Nachricht an alle Medienkomponenteninstanzen geschickt, die eine entsprechende Sensorinstanz besitzen können, d. h. alle mit Markierung *Transition*. Es gibt nur eine geeignete Transition, die aufgrund der Sensorinstanz schalten kann. Die GTP-GTR $pgtr_{\text{„}Enabled/FiringToTransition\text{“}}$ der Transition wird (optional) bei jedem Iterationsschritt angewendet, wobei die jeweilige Komponentenhyperkante des Schleifendurchlaufs (*ircv*) als Teilmatch (für *owner*) vorgegeben wird. $\triangle$

Interne Ereignisse

Einige der modellierten Transitionen entsprechen internen Ereignistypen. Dies ist der Fall, wenn der Zeitpunkt des Zustandsübergangs durch eine TCR berechnet werden kann, d. h. Transitionen mit folgenden Trigger-Arten:

- Abschlussereignis-Trigger (kein Trigger modelliert)
- *inv broken*-Trigger

- Zeitereignis-Trigger
- Bedingungssensor-Trigger (inkl. Kollisionssensor-Trigger)

Nachrichtensensoren führen nicht zur Erzeugung von internen Ereignissen, sondern werden von anderen GTRs aufgerufen (vgl. Abschnitt 7.3.3, S. 248ff).

Übersetzungsregel 7.16 (Interne Ereignistypen)
Die Menge der internen Ereignistypen $\tilde{\mathcal{Q}} \subseteq \mathcal{TR}$ ist gegeben durch die Menge der folgenden Transitionen:

$$\begin{aligned}\tilde{\mathcal{Q}} = \{tr \in \mathcal{TR} \mid\ & tr.trigger = null\ or \\ & tr.trigger.extension\_TriggerEx.isInvBrokenTrigger()\ or \\ & tr.trigger.event.oclIsKindOf(uml::TimeEvent)\ or \\ & tr.trigger.event.oclIsKindOf(aml::ConstraintSensor)\}\ .\end{aligned}$$ △

Ereignis-GTRs und Ereigniszuordnung

Die Menge der Ereignis-GTRs eines AAS/GTs ergibt sich aus den GTP-GTRs für Transitionen (Übersetzungsregel 7.2, S. 233), beschränkt auf die Transitionen für interne Ereignistypen (Übersetzungsregel 7.16), und den GTP-GTRs zur flüchten Aktivierung von Benutzersensoren (Übersetzungsregel 7.15, S. 253). Diese werden angewendet, sobald ein entsprechendes Ereignis eintritt.

Übersetzungsregel 7.17 (Ereignis-GTRs und Ereigniszuordnung)
Die Menge der Ereignis-GTRs ist gegeben durch

$$\mathcal{G} = \bigcup_{tr \in \tilde{\mathcal{Q}}} pgtr_{tr} \cup \bigcup_{us \in \hat{\mathcal{Q}}} pgtr_{us} = \bigcup_{qt \in \mathcal{Q}} pgtr_{qt}$$

Die Ereigniszuordnung ist gegeben durch $evrule(qt) = pgtr_{qt}$ für alle $qt \in \mathcal{Q}$.△

Prioritätszuordnung

Um die in Abschnitt 5.3.2, S. 131, definierten Prioritätsregeln für AML durch die internen Ereignisse des AAS/GTs abzubilden, kann für die Prioritätszuordnung folgende Kodierung verwendet werden.

Übersetzungsregel 7.18 (Prioritätszuordnung)
Die Prioritätszuordnung ist gegeben durch

$$prio(tr) = gp(tr) + dp(tr) \cdot 2 + bp(tr) \cdot 2 \cdot MAXD + sp(tr) \cdot 2 \cdot MAXD \cdot 4$$

für alle internen Ereignistyp $tr \in \tilde{\mathcal{Q}}$. Die Hilfsfunktionen $gp(tr)$, $dp(tr)$, $bp(tr)$, $sp(tr) : \tilde{\mathcal{Q}} \to \mathbb{N}_0$ sind dabei gegeben durch

$$gp(tr) = \begin{cases} 1 & \text{, falls } tr.guard <> null \\ 0 & \text{, sonst} \end{cases} ,$$

$$dp(tr) = tr.source.getDepth() ,$$

$$bp(tr) = \begin{cases} 0 & \text{, falls } tr.extension\_TriggerEx.isInvBrokenTrigger() \\ 1 & \text{, falls } tr.trigger = null \\ 2 & \text{, falls } tr.trigger.event.oclIsKindOf(uml::TimeEvent) \\ 3 & \text{, falls } tr.trigger.event.oclIsKindOf(aml::ConstraintSensor) \end{cases} \text{ und}$$

$$sp(tr) = \begin{cases} tr.trigger.event.oclAsType(aml::Sensor).priority \\ \qquad \text{, falls } tr.trigger.event.oclIsKindOf(aml::Sensor) \\ 0 \qquad \text{, sonst} \end{cases} . \ \triangle$$

Die darin verwendete Konstante $MAXD \in \mathbb{N}^+$ muss im Rahmen der Übersetzung auf einen Wert gesetzt werden, der höher als die maximale Verschachtelungstiefe der Zustände ist.

Die Kodierung berücksichtigt dabei, ob für eine Transition ein Wächter spezifiziert wurde (*gp*), welche Verschachtelungstiefe die Transition bzw. deren Quellzustand hat (*dp*), welche Ereignisart als Trigger verwendet wird (*bp*) und welcher Prioritätswert für den Sensor eines möglichen Sensortriggers modelliert wurde (*sp*).

Zeitberechnungsregeln

Für jeden internen Ereignistyp muss in AAS/GT eine TCR festgelegt werden, die den Zeitpunkt eines jeweiligen Ereignisses berechnet. Daher muss für jede Transition, die sich in der Menge der internen Ereignistypen befindet, eine TCR erzeugt werden muss. Dabei entscheidet vor allem die Art des Triggers, wie die TCR gebildet wird.

Hilfsdefinitionen

Folgende Funktion ermöglicht es, die in Zeitereignissen enthaltenen Ausdrücke zu verwenden:

- *gtime* ordnet jedem Zeitereignis $te \in \mathcal{TE}$ einen Ausdruck zu, der zur Laufzeit (daher die Verwendung innerhalb von $\langle\!\langle \ldots \rangle\!\rangle$-Blöcken) zu einem Zeitpunkt $t \in \mathbb{T}^\omega$ ausgewertet werden kann. $\triangle$

Übersetzungsregel 7.19 (Zeitberechnungszuordnung)
Gegeben sei eine Transition $tr \in \tilde{\mathcal{Q}}$. Die *Zeitberechnungsregel* $tcr_{tr} \in \mathcal{TC}_{\{pgtr_{tr}\}}$ *der Transition* tr ist für alle attributierten Hypergraphen $H_A \in \mathcal{H}_{\mathcal{LV}}$ und Matches $L_{tr} \overset{m}{\hookrightarrow} H$ je nach Fall wie folgt gegeben:

(1) Falls $tr.trigger = null\ and\ not(tr.source.oclisKindOf(aml::State))$, gilt

$$tcr_{tr}(m, H_A) = \langle\langle at \rangle\rangle \ .$$

(2) Falls $tr.extension\_TriggerEx.isInvBrokenTrigger()$, gilt

$$tcr_{tr}(m, H_A) = \langle\langle at \rangle\rangle \ .$$

(3) Falls $tr.trigger = null\ and\ tr.source.oclisKindOf(aml::State)$, gilt

$$tcr_{tr}(m, H_A) = max(\{\langle\langle owner.getStateChangeTime(\{\{tr.source\}\})\rangle\rangle, \langle\langle owner.getAnimationEndTime(\{\{tr.source\}\})\rangle\rangle\}) \ .$$

(4) Falls $tr.trigger.event.oclIsKindOf(uml::TimeEvent)\ and\ not(tr.trigger.event.oclAsType(uml::TimeEvent).isRelative)$, gilt

$$tcr_{tr}(m, H_A) = \langle\langle \{\{gtime(tr.trigger.event.oclAsType(uml::TimeEvent))\}\}\rangle\rangle \ .$$

(5) Falls $tr.trigger.event.oclIsKindOf(uml::TimeEvent)\ and\ tr.trigger.event.oclAsType(uml::TimeEvent).isRelative$, gilt

$$tcr_{tr}(m, H_A) = \langle\langle owner.getStateEnteredTime(\{\{tr.source\}\}) + \{\{gtime(tr.trigger.event.oclAsType(uml::TimeEvent))\}\}\rangle\rangle \ .$$

(6) Falls $tr.trigger.event.oclIsKindOf(uml::ConstraintSensor)$, gilt

$$tcr_{tr}(m, H_A) = \langle\langle at \rangle\rangle \ .$$

Die Zeitberechnungszuordnung ist gegeben durch $evtcr(tr) = tcr_{tr}$ für alle $tr \in \tilde{\mathcal{Q}}$. △

In den Fällen (1) und (2) soll das interne Ereignis zum Zeitpunkt der Berechnung (Animationszeit at) ausgelöst werden, d. h. „sofort“. Dabei ist (1) für eine Transition mit Abschlussereignis-Trigger und Pseudozustand als Quellknoten[11] und (2) für eine Transition mit *inv broken*-Trigger vorgesehen.

Fall (3) behandelt ebenfalls den Abschlussereignis-Trigger, allerdings ist ein Quellzustand gegeben. Eine Grundbedingung für eine anwendbare GTP-GTR bei einem derartigen Trigger ist, dass im Falle eines zusammengesetzten Zustands keine nicht-finalen Zustände innerhalb des Quellzustands aktiv sind (vgl. Übersetzungsregel 7.6, S. 239). Wird diese Bedingung erfüllt (ansonsten sind die Ergebnisse der TCR irrelevant), soll das interne Ereignis im Falle eines Animationszustands unmittelbar nach dem Endzeitpunkt der Animationsanweisungen ausgelöst werden. Ist dieser bereits vergangen oder sind keine Animationsanweisungen verfügbar, dann gilt der Zeitpunkt als möglicher Auslösezeitpunkt, an dem der Quellzustand eingenommen wurde oder sich zum letzten Mal ein

[11] Unter den gegebenen Einschränkungen muss dies der Initialzustand sein.

interner Zustand des Quellzustands verändert hat. Die verwendeten Methoden (im Zusammenhang mit *max*) stellen dabei die genannten Punkte sicher (vgl. Abschnitt 5.5, S. 178).

Die Fälle (4) und (5) werden bei Triggern für Zeitereignisse (mit relativem und absoluten Zeitpunkt) angewendet. Im relativen Fall (5) bezieht sich der ausgewertete Zeitpunkt auf den Zeitpunkt, an dem der Quellzustand eingenommen wurde. Es gilt anzumerken, dass die in beiden Fällen notwendige Auswertung nur zu dem Zeitpunkt durchgeführt werden darf, an dem der Quellzustand eingenommen wurde.

Fall (6) muss speziell betrachtet werden. Die dort angegebene Berechnungsvorschrift gilt nur für den Fall, dass im Kontext des Bedingungssensors keine animierten Attribute beobachtet werden. Gemäß dieser Vorschrift wird das interne Ereignis „sofort“ ausgelöst (vgl. (1) und (2)).

Schwierig wird die Situation, wenn ein Bedingungssensor animierte Werte überwachen muss. Darunter fallen Kollisionssensoren bspw. automatisch, da sie Bewegungen (animierte Positions-Attribute) und damit verbundene Kollisionen überwachen müssen. Die TCR muss in diesem Fall den Zeitpunkt berechnen, an dem die formulierte Bedingung mit animierten Attributen zum ersten Mal zutrifft. Da Attribute theoretisch beliebig animiert werden können, ist die Implementierung einer allgemeinen Lösung schwierig. Aus diesem Grund muss die generierte Standard-TCR (6) in entsprechenden Fällen *manuell angepasst* werden. In der Beispiel-Implementierung ist es möglich, eine solche Anpassung im generierten Klassencode einzubetten. Auf mögliche Näherungsverfahren oder regelmäßige Prüfung der Sensorenbedingungen („Polling“) soll an dieser Stelle nicht eingegangen werden.

Beispiel 7.11 (Angepasste TCR für Kollisionsberechnung in Avalanche)
Für die Sprache Avalanche entspricht die in Beispiel 7.3, S. 236, beschriebene Transition einem internen Ereignistyp, da ein Bedingungssensor als Trigger verwendet wird. Da dieser Kollisionssensor eine animierte Situation überwacht, muss eine Berechnung des Kollisionszeitpunkts zwischen Murmel und Hebel implementiert werden. Die Berechnung ist vergleichsweise einfach, da durch Graphmuster und zusätzliche Pfad-Anwendungsbedingungen bereits sichergestellt wird, dass sich die Murmel auf der richtigen Bahn der Weiche befindet. In Listing C.1/33 wird die im generierten Klassencode manuell eingefügte Berechnung gezeigt. △

7.4 Einbettung in das DiaMeta-System

Um die beschriebene Vorgehensweise zu validieren, wurde die Übersetzung von AML/GT-Modellen in eine AAS/GT-konforme Sprachspezifikation inkl. Codegenerierung implementiert. Hierfür wurde das DiaMeta-System erweitert. Mit dem erweiterten DiaMeta-System konnten danach zahlreiche animierte Editoren generiert werden, u. a. für Alligator Eggs, Avalanche und B/E-Netze. Die folgenden Abschnitte beschreiben, wie der Spezifikations- und Generierungsprozess angepasst wurde (Abschnitt 7.4.1). Anschließend folgt die Vorstellung des

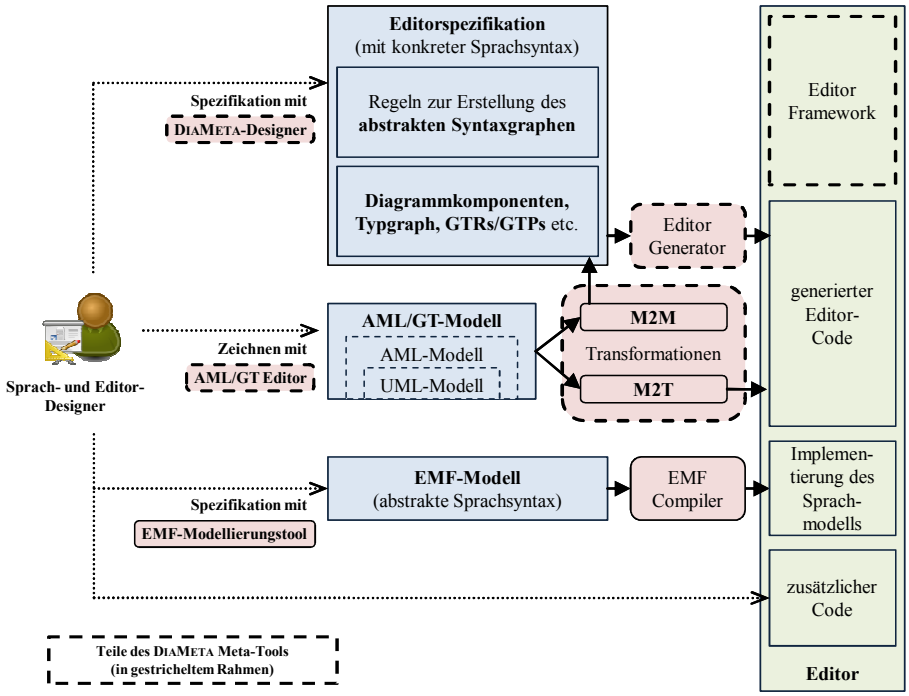


Abbildung 7.4: Spezifikation und Generierung eines animierten DiaMeta-Editors

AML/GT-Editors, ein in diesem Zusammenhang entstandenes Hilfsmittel und Editor zur Erstellung von AML/GT-Modellen (Abschnitt 7.4.2).

7.4.1 Angepasster Spezifikations- und Generierungsprozess

Abb. 7.4 illustriert den angepassten Spezifikations- und Generierungsprozess (vgl. Abb. 3.21, S. 95). Neben der unveränderten Modellierung der abstrakten Sprachsyntax mittels EMF muss die konkrete Sprachsyntax (inkl. Animation und Dynamik) in Form eines AML/GT-Diagramms modelliert werden. Hierfür steht ein vollständiger AML/GT-Editor zur Verfügung.

Das AML/GT-Modell kann durch unterschiedliche Komponenten transformiert werden. Zum einen kann daraus direkt Java-Code für den gewünschten Editor generiert werden: eine Modell-zu-Text-Transformation (M2T-Transformation). Dieser Teil wurde mittels Acceleo [Acc08] implementiert. Acceleo sieht sich selbst als eine „pragmatische“ Umsetzung der OMG-Spezifikation für die MOF Model to Text Transformation Language [MOF08].

Der größere Teil der Transformation, eine Modell-zu-Modell-Transformation (M2M-Transformation), wurde mittels Eclipse QVTo implementiert. Dabei stellt Eclipse QVTo eine Umsetzung der OMG-Spezifikation für QVTo [QVT11] dar.

Ziel der Transformation ist die XML-orientierte DiaMeta-Sprach- und Editorspezifikation. Diese enthält u. a. auch die Teile, die im Rahmen eines AAS/GTs spezifiziert werden müssen.

Nach der Transformation muss die entstandene DiaMeta-Sprach- und Editorspezifikation vervollständigt werden. Der wichtigste Teil, der darin zunächst fehlt, ist die Verknüpfung von konkreter und abstrakter Syntax. Diese wird durch Reduktionsregeln spezifiziert (vgl. Abschnitt 3.3.1, S. 90ff). Ansonsten können DiaMeta-spezifische Merkmale hinzugefügt werden, die nicht durch AML/GT abgedeckt werden und eher der Editorspezifikation und nicht der Sprachspezifikation zuzuordnen sind (vgl. Abschnitt 3.3.2, S. 94).

Aus der fertigen Spezifikation kann mit dem DiaMeta-Generator auf herkömmliche Weise Editor-Code erzeugt werden. Zur Fertigstellung des Editors sind danach ggf. noch Anpassungen und Erweiterungen notwendig, z. B. bestimmte TCRs für Bedingungssensoren (vgl. Abschnitt 7.3.4, S. 257) oder ein Layoutalgorithmus (vgl. Abschnitt 3.3.1, S. 92).

7.4.2 AML/GT-Editor

Der zur Spezifikation verwendbare AML/GT-Editor wurde selbst mittels DiaMeta generiert. Unter Verwendung des AML-Metamodells und des AML/GT-Profils[12] kann der AML/GT-Editor aus den erzeugten AML/GT-Diagrammen XMI-Dateien[13] exportieren. Eine solche Datei kann im Anschluss für die Modelltransformationen geladen und verarbeitet werden. Zudem kann der Editor die erstellten Diagramme bzw. Modelle jederzeit gegen das AML-Metamodell und das AML/GT-Profil validieren.

Der AML/GT-Editor wurde außerdem zu einem Tool ausgebaut, mit dem die Animationen in animierten DiaMeta-Editoren gesteuert und überwacht werden können. Bereits bei der Entwicklung von Editoren für die Sprachen aus Abschnitt 2.3, S. 30, konnten diese Möglichkeiten sinnvoll genutzt werden. Ein fehlerhafter Aufbau der Sprachdynamik, v. a. im Zusammenhang mit den GTs, kann damit schnell erkannt und verbessert werden. Gleichzeitig gewährt er jederzeit einen Überblick über die Vorgänge in einem animierten Diagramm und die durchgeführten GTs. Verglichen mit einer Sprachspezifikation auf der Grundlage von GTR-Listen (vgl. Abschnitt 6.1, S. 190), bietet der präsentierte Ansatz diesbezüglich enorme Vorteile.

[12] Das AML-Metamodell und das AML/GT-Profil wurden gemäß OMG-Standard mittels UML spezifiziert, dann aber in die notwendigen Ecore-Modelle transformiert.

[13] XML Metadata Interchange (XMI) ist ein Austauschformat der OMG für Daten, die auf der MOF aufbauen, d. h. in erster Linie für Modelle. Das Format wird inzwischen von vielen Entwicklungswerkzeugen unterstützt. Verwendet wird XMI in der Version 2.0 [XMI05].

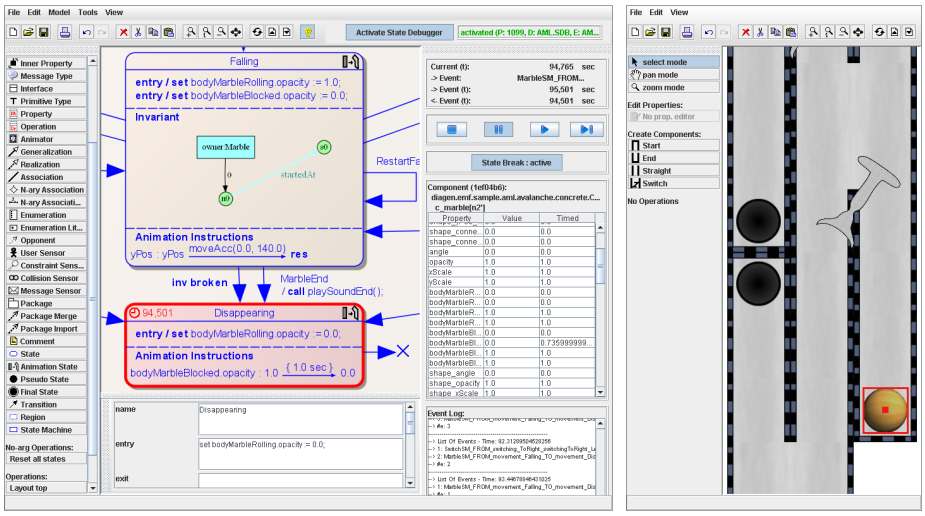


Abbildung 7.5: AML/GT-Debugger und überwachter Editor

Im Hinblick auf die Steuerung und Überwachung von animierten Editoren ist es mit dem AML/GT-Editor möglich,

- den Ablauf der Animationszeit zu kontrollieren, d. h., zu pausieren, zu überspringen, zu beschleunigen oder zu verlangsamen,
- die aktuellen Zustandsdaten direkt im AML/GT-Diagramm zu visualisieren[14],
- öffentliche und interne Attribute von Diagrammkomponenten zu manipulieren und
- den Animationsablauf zu unterbrechen, sobald ein bestimmter Zustand einer Diagrammkomponente erreicht wird.

Aufgrund der genannten Merkmale kann der AML/GT-Editor mit typischen Debuggern für (textuelle) Programmiersprachen verglichen werden, weshalb er in diesem Modus auch AML/GT-Debugger genannt wird. Auch in Arbeiten wie [BSE11, EJ01], in denen Abläufe in UML-Zustandsdiagrammen visualisiert werden sollen, wird in einem solchen Zusammenhang der Begriff „Debugging“ verwendet.

Ein Screenshot des AML/GT-Editors/Debuggers wird in Abb. 7.5 (links) abgebildet. Er zeigt einen Teil des AML/GT-Diagramms für Avalanche. Der daraus generierte Editor wird ebenfalls dargestellt (rechts). AML/GT-Debugger und Avalanche-Editor sind verbunden, d. h., mit dem AML/GT-Debugger kann

[14] Der aktuelle Zustand einer selektierten Diagrammkomponente wird eingefärbt (inkl. Zeitpunkt an dem der Zustand aktiv wurde), aktuelle Attributwerte werden zusammen mit deren zeitabhängigen Werten angezeigt, die Liste der geplanten internen Ereignisse wird geloggt etc.

die Animationszeit gesteuert werden. Außerdem werden darin „Livedaten“ der aktuell gewählten Avalanche-Komponente gezeigt. Im Screenshot ist dies eine Murmel, weshalb der AML/GT-Debugger ihren aktuellen Zustand *Disappering* und weitere Daten anzeigt. Der Animationsablauf wurde bei Erreichen dieses Zustands automatisch gestoppt, da auf diesem Zustand in Analogie zu gewöhnlichen Debuggern eine Art „Haltepunkt“ gesetzt wurde.

7.5 Verwandte Arbeiten

In diesem Abschnitt werden abschließend verwandte Arbeiten genannt, die auf ähnlichen Ideen wie AML/GT beruhen – auch in Bezug auf die Modellierung von Sprachen – oder sich in ähnlicher Weise mit Codegenerierung und Modelltransformation beschäftigen. In diesem Zusammenhang wird zusätzlich auf Arbeiten über die Semantik von UML-Zustandsautomaten eingegangen.

Codegenerierung aus Klassenmodellen

Die Generierung von Code aus UML-Klassendiagrammen wurde inzwischen sehr häufig untersucht und realisiert. Entsprechende M2T-Transformationen gehören inzwischen zur Standardfunktionalität vieler kommerzieller und frei erhältlicher UML-Werkzeuge, wie bspw. ArgoUML [Arg11] (Open Source), StarUML [Sta05] (Open Source), Enterprise Architect [EA112], Magic Draw [Mag12], Together [Tog08], UModel [UMo10], Visual Paradigm [Vis12], u. v. m. Zahlreiche Lösungen gibt es auch im Umfeld von Eclipse [Ecl12], z. B. die Codegenerierung für EMF [Ecl12], die im Grunde jedoch auf Ecore-Modellen basiert. Weitere Tools wie Fujaba oder Rhapsody, die weiter unten noch einmal erwähnt werden, unterstützen ebenfalls die Generierung von Klassencode. Java ist dabei eine häufig verfügbare Zielsprache. Bei ausreichender Anpassungsfähigkeit können derartige Tools auch zur Transformation des statischen Teils von AML- und AML/GT-Modellen verwendet werden.

Eine kleine Gegenüberstellung der Generierungsmöglichkeiten einiger der genannten Tools inkl. Fallstudien, die Vorstellung des eigenen Generators UJECTOR und weitere Verweise bietet [Usm09]. Ein interessanter Artikel ist auch [Ges08], da darin die oftmals schwierig umsetzbare Semantik von Assoziationen in Bezug auf Codegenerierung untersucht wird.

Codegenerierung aus Zustandsautomaten

Auch die Generierung von Code auf der Grundlage von Zustandsautomaten wird von vielen UML-Werkzeugen durch eingebaute Funktionalität unterstützt. Bereits erwähnt wurden ArgoUML, Enterprise Architect, UModel, Visual Paradigm, Rhapsody und Fujaba. Neben einem eigenen System (dCode) und zugehöriger Quellcodegenerierung werden weitere Tools in [AT01] aufgeführt. Ansonsten existieren unzählige weitere Arbeiten zu diesem Thema, z. B. [KM02, CJ05,

TVT07]. Herauszuheben sind dabei die Arbeiten von Harel, z. B. [HG96], die häufig als Grundlage für andere Arbeiten gesehen werden.

Bei der Generierung wird meist das Zustands-Entwurfsmuster (vgl. Abschnitt 7.3.2, S. 227) verwendet, allerdings gibt es auch andere Möglichkeiten wie bspw. Zustandsübergangstabellen. Einige existierende Muster werden auch in [NT03] referenziert.

An dieser Stelle sei noch einmal gesondert auf das Tool Fujaba [FNTZ00] hingewiesen. Die Fähigkeiten von Fujaba als GTS wurden bereits in Abschnitt 3.2.6, S. 86, hervorgehoben. Die darin eingesetzten Story-Diagramme (spezielle Aktivitätsdiagramme) lassen sich bedingt mit AML/GT-Zustandsdiagrammen vergleichen, da auch darin Graphmuster (bzw. Story-Patterns) eingesetzt werden und Aktivitätsdiagramme generell mit Zustandsdiagrammen vergleichbar sind. Während in AML/GT allerdings Invarianten modelliert werden, werden die GT-Schritte in Story-Diagrammen „direkt“ modelliert, was Fujaba zu einem System zur programmierten Graphersetzung macht.

Semantik von Zustandsautomaten

Durch die Übersetzung von Zustandsautomaten in das formale System AAS/GT, wird die Semantik der verwendeten AML/GT-Zustandsautomaten beschrieben. Dies ist ebenfalls bei vielen Tools der Fall, bei denen Code aus Zustandsautomaten generiert werden kann (siehe oben).

Viele Arbeiten beschäftigen sich aber auch direkt mit der genauen Semantik von Zustandsautomaten.[15] Dies ist aufgrund zahlreicher Spezifikationslücken und Fehler v. a. in der UML wichtig. Bereits in Abschnitt 5.3, S. 127, wurden Arbeiten erwähnt, die sich mit semantischen Problemen der UML befassen. Darin werden auch Projekte genannt, die sich mit Erweiterungen bzw. Teilmengen von UML für eine präzisere Semantik auseinandersetzen. Häufig referenziert werden auch die von Harel beschriebenen Semantiken von Statemate [HN96] und Rhapsody [HK04], wobei der zuletzt genannte Artikel v. a. auf die Unterschiede eingeht.

Nach [GBEE11, Erm06] gibt es zwei wichtige Ansätze zur Definition der formalen Semantik einer Sprache für Zustandsautomaten: die Definition einer Interpretersemantik (engl. interpreter semantics) und die Definition einer Compilersemantik (engl. compiler semantics). Zur Beschreibung dienen abstrakte Zustandsmaschinen [BCR00], Petri-Netze bzw. High-level-Netze [BDM02, BP01], Core State Machines [FS07], Transitionssysteme [LMM99] und einige mehr. Eine Übersicht kann in [Cra06] gefunden werden.

Im Falle der Interpretersemantik wird die operationelle Semantik der UML-Zustandsautomaten beschrieben, indem ein Interpreter für die Automaten angegeben wird. Da GTSe eine fundierte mathematische Grundlage bieten, werden solche Beschreibungen auch auf Basis von GTRs formuliert. Beispiele für solche Arbeiten sind [Var02, BELT04, Erm06, GBEE11].

[15] Die Webseite des „UML2 Semantics Project“ (`www.cs.queensu.ca/~stl/internal/uml2/`) bietet u. a. für UML-Zustandsautomaten eine sehr lange Liste an Arbeiten.

Auf der anderen Seite kann für Zustandsautomaten auch eine Compilersemantik definiert werden. Dabei wird die Semantik der Automaten beschrieben, indem eine Übersetzung der Automaten in eine semantische Domäne angegeben wird (vgl. [Erm06]). Falls GTSe diese semantische Domäne repräsentieren, kann man hierbei auch von GTS-Compilersemantik sprechen. In [MSP96, GPP98, Kus01, KGKK02, EHKZ05, KZDX09] wird jeweils eine solche GTS-Compilersemantik formuliert. In [KGKK02] wird zusätzlich erläutert, wie GTRs mit Operationen eines UML-Klassendiagramms und Zustandsübergängen eines UML-Zustandsdiagramms verknüpft werden können.

Visuelle Modellierung von Invarianten und Bedingungen

Bei der Modellierung mit UML wird häufig die (textuelle) OCL [CODA+11] verwendet, um dem Modell zusätzliche Informationen hinzuzufügen. Sie gilt inzwischen sogar als Bestandteil der UML. So kann die OCL z. B. zur Modellierung von Invarianten (meist in UML-Klassendiagrammen) oder für Bedingungen (als Wächter in UML-Zustandsdiagrammen) eingesetzt werden.

Im Rahmen von AML/GT werden Graphmuster für die oben genannten Zwecke eingesetzt. Daher liegt ein Vergleich von OCL und AML/GT-Konstrukten nahe. Es wurde aber noch nicht intensiv untersucht, ob und wie die Techniken austauschbar sind oder sich sogar ergänzen können. Einen Schritt in diese Richtung präsentiert [CCGL10]. In der Arbeit werden in OCL formulierte Vor- und Nachbedingungen genutzt, um einfache GTRs mit LHS und RHS nachzubilden. In [WTEK08] wird hingegen gezeigt, wie (bestimmte) OCL-Zusicherungen in äquivalente Graphbedingungen umgewandelt werden können. Schließlich wird in [Baa06] darauf eingegangen, wie OCL und GTs kombiniert werden können.

Bottoni et al. präsentieren einen Ansatz, um OCL zu visualisieren. Dabei ist die Sprache Visual OCL entstanden. Mit dieser Sprache können OCL-Ausdrücke visuell (im Sinne von „nicht textuell“) in Diagrammen notiert werden [BKPPT00, BKPPT01]. Einen ähnlichen Zweck wie OCL bzw. Visual OCL, allerdings nicht in Zusammenhang mit UML-Diagrammen, verfolgen auch Constraint Diagrams [Ken97].

Modellierung zur Sprachspezifikation

Modellierung wird zur Spezifikation von Sprachen bereits häufig eingesetzt. Dabei wird fast ausschließlich die abstrakte Sprachsyntax in Form von Metamodellen spezifiziert (vgl. Abschnitt 3.4, S. 95). Mittels AML/GT kann hingegen die konkrete Syntax einer (animierten) Sprache spezifiziert werden.

Nichtsdestotrotz sind die Modelle für konkrete und abstrakte Syntax strukturell oftmals sehr ähnlich (vgl. Abschnitt 7.1, S. 216). Daher existieren Arbeiten wie bspw. [FB05], die Möglichkeiten beschreiben, die konkrete Sprachsyntax direkt auf Basis der abstrakten Sprachsyntax – repräsentiert durch Metamodelle – zu spezifizieren. Andere Arbeiten stellen dar, wie Sprachen mit gegebener abstrakter Syntax, die ebenfalls durch ein Metamodell repräsentiert wird, mit einer Ausführungssemantik versehen werden können. Beim sogenannten Dynamic Meta

Modeling (DMM) [EHHS00, BSE11] wird das Metamodell bspw. zunächst erweitert, um darin Ausführungszustände unterbringen zu können. Danach werden GTRs festgelegt, deren LHS und RHS aus Kollaborationsdiagrammen bestehen, was einen Vergleich mit AML/GT zulässt. Ansonsten stellt DMM einen weiteren Ansatz dar, bei dem die abstrakte Sprachsyntax um Simulationsaspekte erweitert wird (vgl. Abschnitt 4.4, S. 118).

Kapitel 8

Schlußbemerkungen

Dieses Kapitel fasst die Ergebnisse dieser Arbeit und ihren wissenschaftlichen Beitrag zusammen. Abgeschlossen wird mit einem Ausblick, welche potentiellen Möglichkeiten der Lösungsansatz zusätzlich bietet und wie die Arbeit zukünftig fortgeführt werden kann.

8.1 Zusammenfassung und Ergebnisse

Mit dem auf AAS basierenden Formalismus AAS/GT wird ein Animationsansatz vorgestellt, der zur (Teil-)Spezifikation der konkreten Syntax einer interaktiven animierten Sprache verwendet werden kann. Im Formalismus werden dabei insbesondere Graphen und GTRs eingesetzt. Auch die Generierung von Editoren ist auf dieser Grundlage möglich. Bei Verwendung des beschriebenen Animationsansatzes werden außerdem Probleme anderer Ansätze in einem ähnlichem Umfeld umgangen (vgl. Abschnitt 4.4, S. 118).

Um die Sprachspezifikation auch auf höherer Ebene zu ermöglichen, wird zudem die Modellierungssprache AML präsentiert. Neben dem zustandsbasierten Verhalten einzelner Sprachkomponenten kann darin auch die Darstellung der konkreten Syntax inkl. präziser grafischer Animationsabläufe modelliert werden. Es ist ebenfalls möglich, den Detailgrad von AML-Modellen unterschiedlich zu wählen, wodurch AML sowohl zur Planung als auch zur Feinspezifikation eingesetzt werden kann. Mittels AML/GT wird außerdem eine Spracherweiterung vorgestellt, die es ermöglicht, hypergraph-spezifische Konstrukte in AML einzubetten. Schließlich wird beschrieben, wie ein AML/GT-Modell in ein System übersetzt werden kann, das auf der Grundlage von AAS/GT arbeitet.

Die Ergebnisse zeigen, dass AML und AML/GT zur Spezifikation interaktiver animierter Sprachen auch praktisch eingesetzt werden können. Die Modellierungssprachen eignen sich dabei zur Planung von animierten Sprachen, zur schnellen Erstellung von Editor-Prototypen oder zur Realisierung vollständiger Editoren. Der verwendete, modellgetriebene Ansatz wurde durch Generierung zahlreicher animierter Editoren (vgl. Abschnitt 2.3, S. 30) getestet und validiert.

Hierfür wurde eine Lösung inkl. AML-Editor, AML/GT-Editor, Modelltransformationen und Codegenerierung auf der Grundlage des Meta-Tools DiaMeta vollständig implementiert. Besonderheiten und Einschränkungen wurden dabei in Abschnitt 7.3.1, S. 224, beschrieben. Auch der folgende Abschnitt greift diesbezüglich einige Ideen auf.

8.2 Ausblick

Mittels AML/GT kann eine konkrete Sprachsyntax fast vollständig spezifiziert werden. Es gibt allerdings Aspekte, die nicht durch AML/GT abgedeckt werden. So ist bspw. die Spezifikation von Layout-Beschränkungen für eine visuelle Sprache nicht direkt möglich. Für die Implementierung des Alligator Eggs-Editors musste der Algorithmus zur Neuanordung der Sprachelemente bei einem Reduktionsvorgang daher von Hand programmiert werden. Die Spezifikationen von Layout-Beschränkungen könnte jedoch genutzt werden, um auch (Hilfs-)Code für automatisches Diagrammlayout zu generieren. Einige Meta-Tools wie GenGED [Bar98] bieten bspw. die Möglichkeit, Layout-Beschränkungen für die konkrete Syntax einer visuellen Sprache zu spezifizieren. Auch die in [Mai12] vorgestellten Layout-Muster und deren Kombinationsmöglichkeiten könnten Anwendung in AML/GT finden. Eine Layout-Engine, welche derartige Layout-Muster interpretiert und für Diagramme angewendet werden kann, wurde bereits in DiaMeta integriert.

Die Nutzbarkeit alternativer AML-Basisframeworks (vgl. Abschnitt 5.5, S. 175) ermöglicht, animierte Sprachen auf unterschiedlichen Grundgerüsten zu entwickeln. Das vorgestellte Basisframework ist auf die Entwicklung diagrammorientierter Sprachen ausgelegt. Es könnte daher untersucht werden, ob dreidimensionale animierte Sprachen mit einem entsprechenden Basisframework ebenfalls umgesetzt werden können. Auch eine Physik-Engine könnte im Basisframework integriert werden und entsprechende Animatoren bereitstellen. Unterstützt die Physik-Engine *a priori* Kollisionserkennung (auch dynamische Kollisionserkennung [Eri04]), könnten bspw. TCRs für Kollisionssensoren automatisch erzeugt werden. Derzeit müssen Formeln zur Berechnung von Kollisionszeitpunkten (oder für ähnlich gelagerte Probleme bei Bedingungssensoren) manuell erstellt werden (vgl. Abschnitt 7.3.4, S. 257).

Es ist außerdem absehbar, dass durch die Umsetzung weiterer animierter Sprachen mittels AML wiederkehrende Animationsmuster in den Modellen erkannt werden können, da in vielen Sprachen ähnliche Elemente oder Abläufe verwendet werden. Diese Muster könnten – analog zu allgemeinen Entwurfsmustern [GHJJ96] – als wiederverwendbare Schablonen herausgearbeitet werden. Bereits bei der Umsetzung der vorgestellten Editoren und der Analyse existierender Sprachen konnten bestimmte Muster erkannt werden. So kann bspw. für die Animation von Schaltvorgängen in B/E-Netzen (siehe Abschnitt 2.3.1, S. 32) oder Sprachen wie Machinations (siehe Abschnitt 2.2, S. 25) ein Muster erstellt werden. Die Animationen in Alligator Eggs (siehe Abschnitt 2.3.2, S. 36) müssen in mehrere Phasen unterteilt werden, wobei in der letzte Phase die Berechnung

eines neuen Layouts angefordert und die Neupositionierung durchgeführt wird. Diese Vorgehensweise kann in anderen Sprachen, z. B. für AVL-Bäume (siehe Abschnitt 2.2, S. 24), ebenfalls genutzt werden. Ein letztes Beispiel, das sich nicht auf die Modellierung in AML beschränkt, bezieht sich auf die zuvor erwähnte Notwendigkeit von a priori Kollisionserkennung. Ist keine entsprechende Physik-Engine verfügbar, kann ein spezielles Muster verwendet werden, bei dem die Position von Sprachelementen mit komplexen Bewegungsabläufen in jeweils kurzen Intervallen aktualisiert wird. Dies kann durch Schleifen mit Triggern für Zeitereignisse an den jeweiligen Zuständen geschehen. In diesem Fall können für Kollisionssensoren dann Verfahren angewendet werden, die Kollisionen *a posteriori* erkennen (auch statische Kollisionserkennung genannt). Dadurch können zwar Effekte wie das sogenannte *Tunneln* auftreten, allerdings arbeiten derartige Algorithmen meist effizienter (vgl. [Eri04]).

AML nutzt in erster Linie Konzepte anderer Modellierungssprachen (vgl. Abschnitt 5.6, S. 182). Daher kann davon ausgegangen werden, dass Analysen dieser Konzepte im Rahmen von anderen Sprachen zum Teil auch für AML gelten, z. B. in Bezug auf Benutzerfreundlichkeit. Trotzdem sollte AML selbst hinsichtlich solcher Aspekte noch genauer evaluiert werden. Es sollte insbesondere untersucht werden, ob die in AML/GT verwendeten graph-spezifischen Konzepte auch von Entwicklern ohne Erfahrung mit Graphen intuitiv verständlich sind. Es ist davon auszugehen, dass AML und AML/GT auf Grundlage der Evaluationsergebnisse verbessert und optimiert werden können.

Anhang A

AML/GT-Diagramme

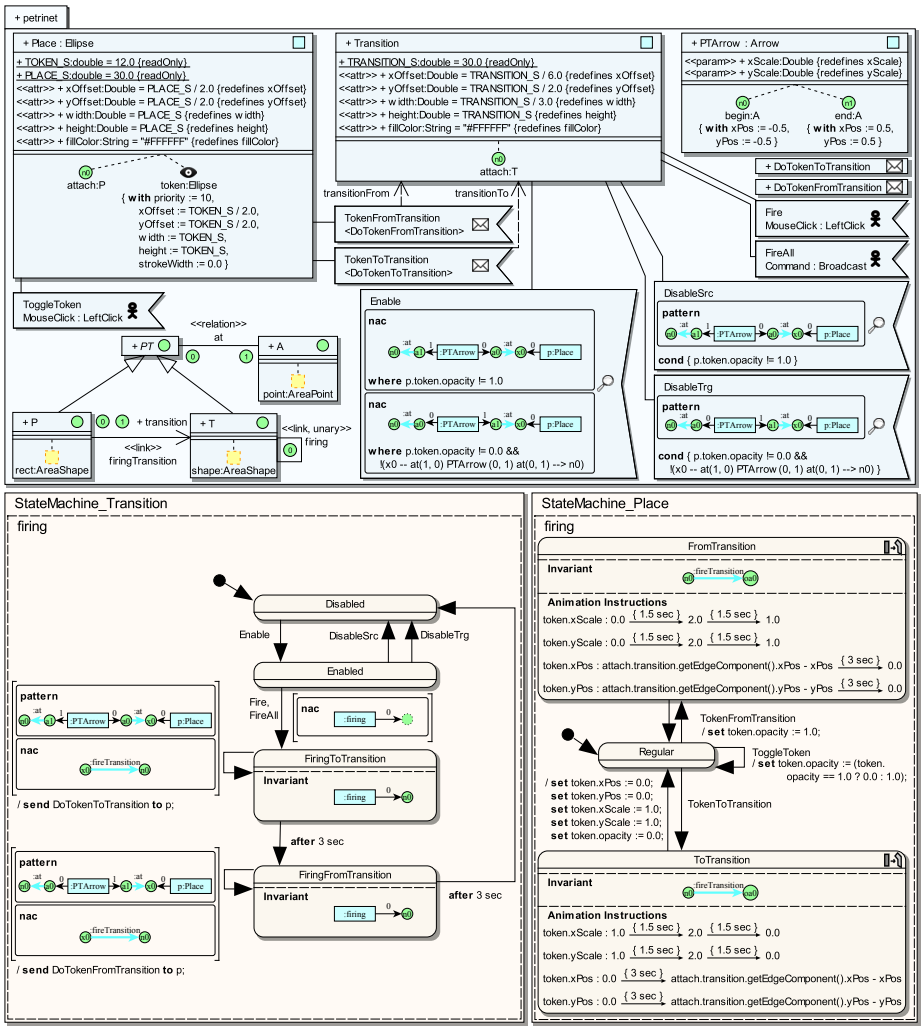


Abbildung A.1: AML/GT-Model für B/E-Netz-Editor

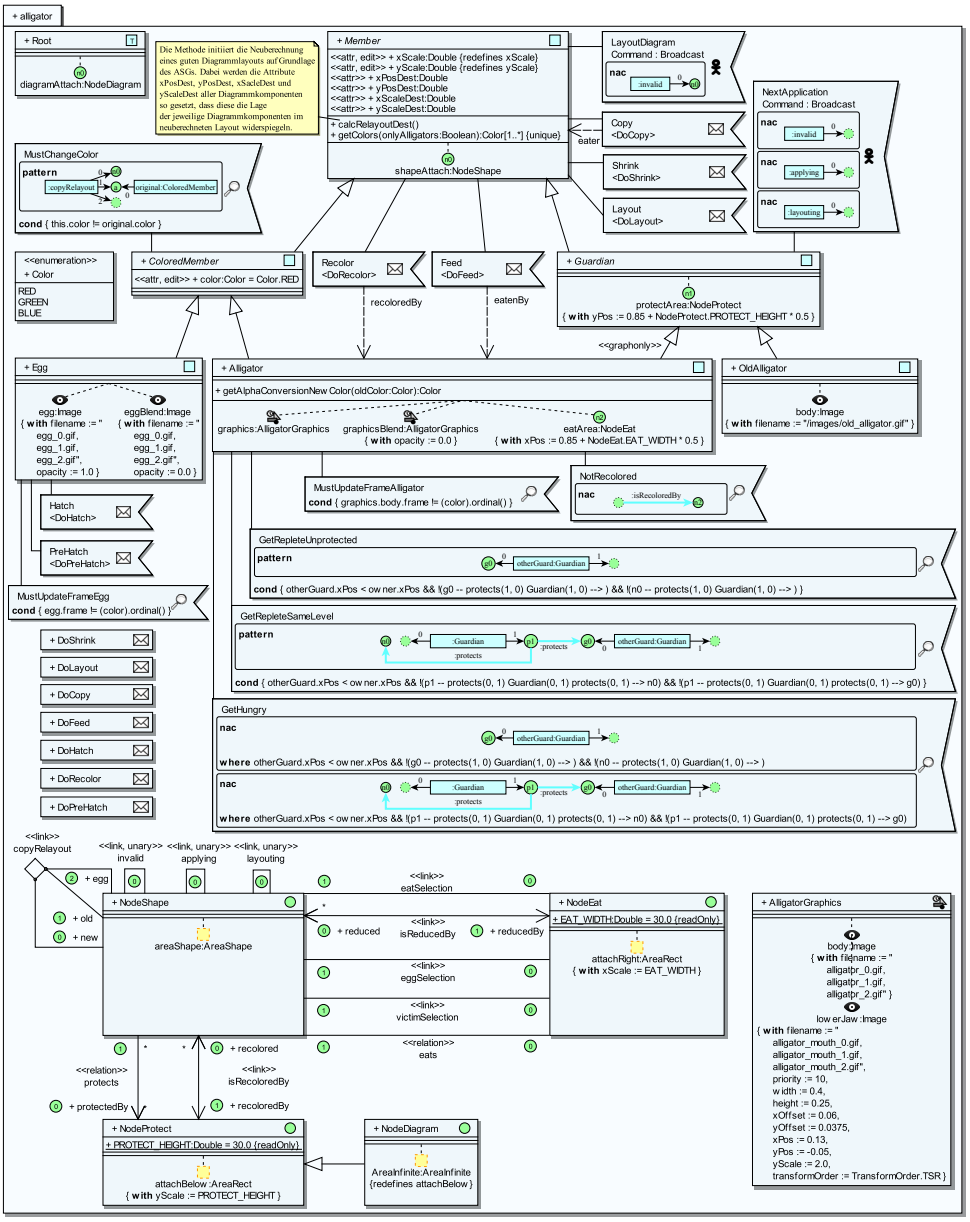


Abbildung A.2: AML/GT-Model für Alligator Eggs-Editor (Struktur)

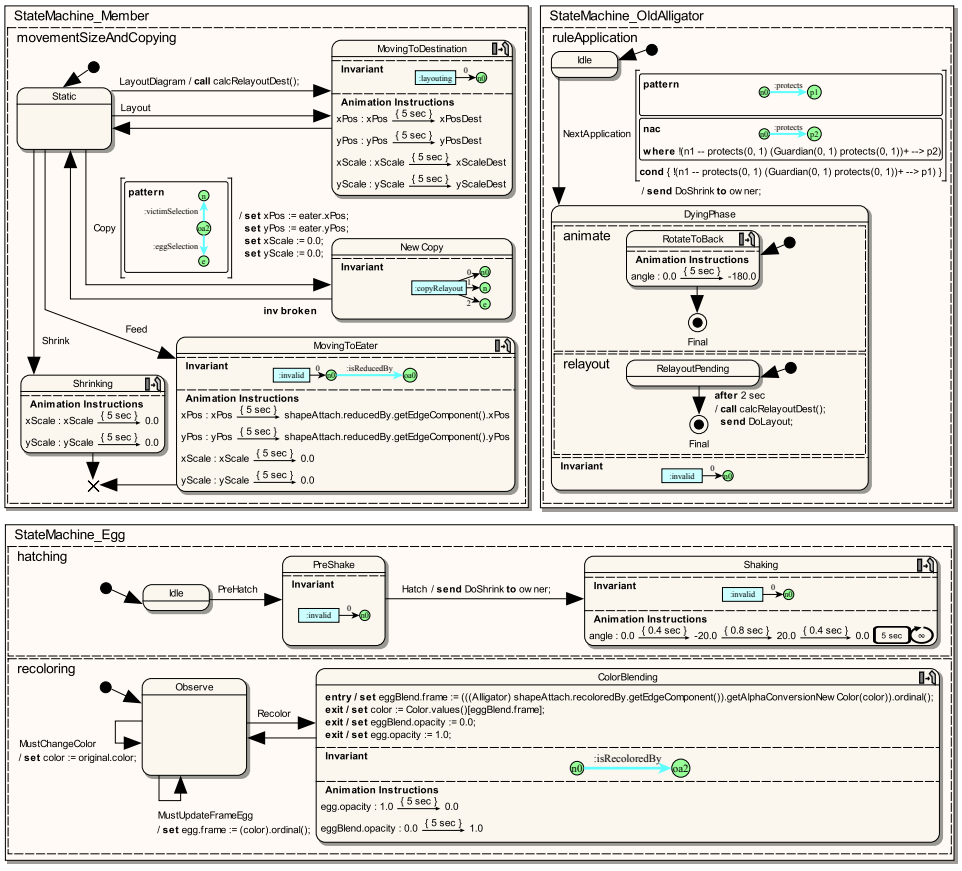


Abbildung A.3: AML/GT-Model für Alligator Eggs-Editor (Verhalten I)

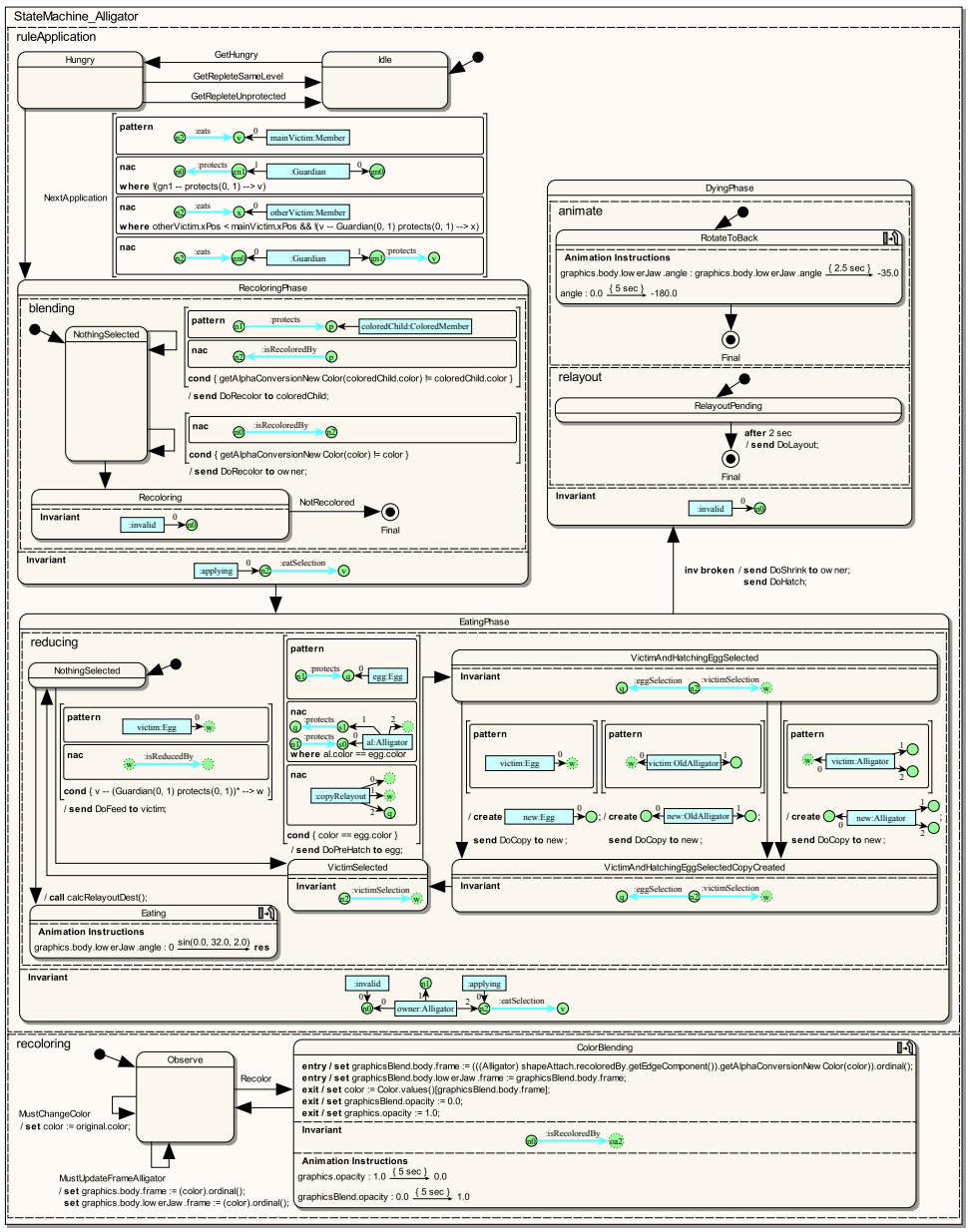


Abbildung A.4: AML/GT-Model für Alligator Eggs-Editor (Verhalten II)

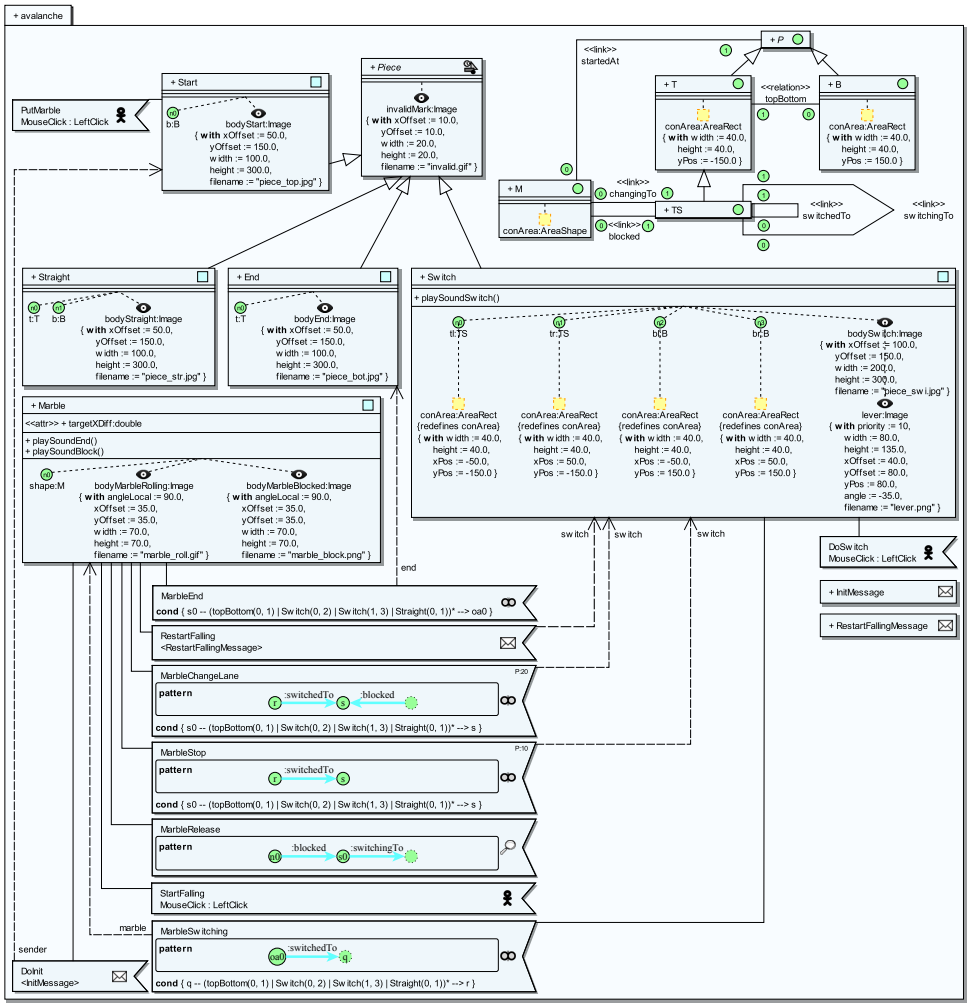


Abbildung A.5: AML/GT-Model für Avalanche-Editor (Struktur)

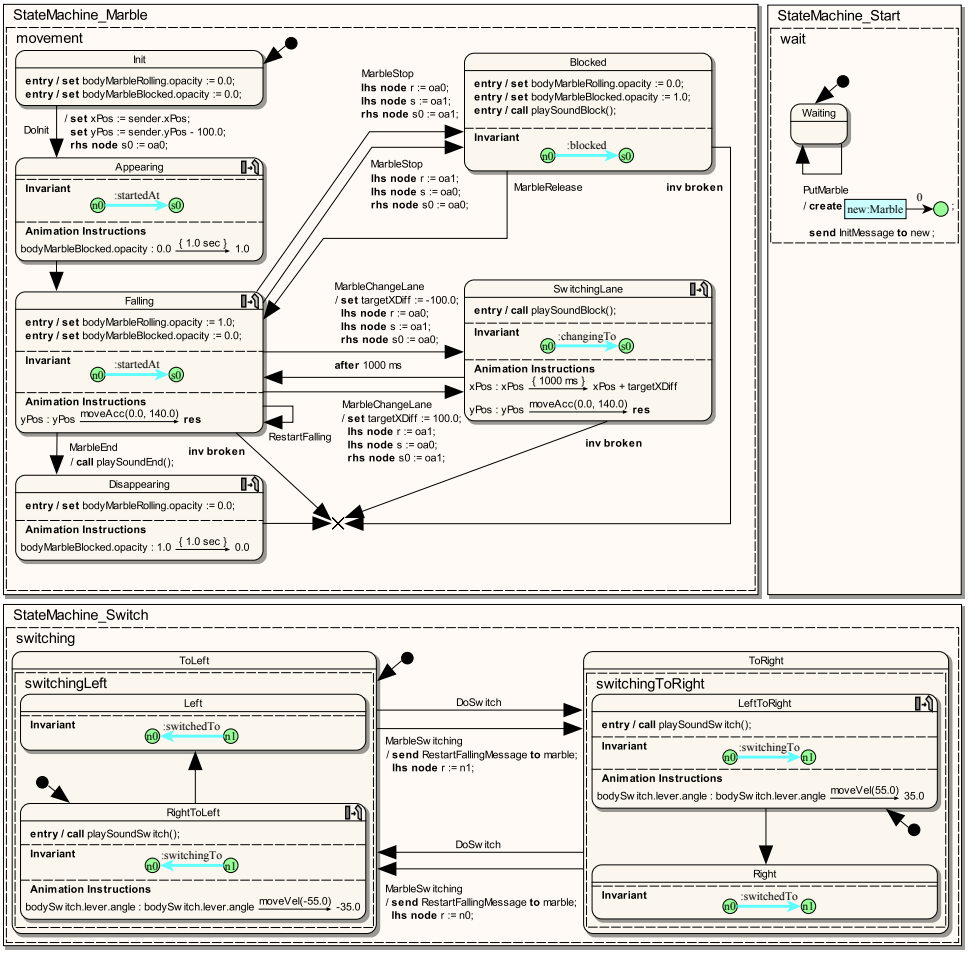


Abbildung A.6: AML/GT-Model für Avalanche-Editor (Verhalten)

Anhang B

Zusicherungen und Operationen (AML und AML/GT)

B.1 Operationen und Zusicherungen des AML-Metamodells

Listing B.1 und Listing B.2 zeigen die Operationen und Zusicherungen des AML-Metamodells mittels OCL. Einige Zusicherungen oder Operationen sind allerdings schlecht in der OCL darstellbar. Diese sind entsprechend markiert und wurden im Rahmen der Beispiel-Implementierung durch Java-Code umgesetzt.

```
1  -- liefert alle Animatoren der Medienkomponente;
2  -- beachtet außerdem Basismedienkomponenten und Neudefinition
3  -- (hier vereinfacht: keine Berücksichtigung der Zugriffsmodifizierer)
4  context aml::MediaComponent::getAllAnimators() : Set(aml::Animator)
5  body : let allAnimatorsNoRedef : Set(aml::Animator) =
6           ownedAnimator->union(
7             allParents()
8             ->select(oclIsKindOf(aml::MediaComponent))
9             .oclAsType(aml::MediaComponent).ownedAnimator
10          )->asSet() in
11       allAnimatorsNoRedef -
12          allAnimatorsNoRedef.redefinedAnimator->asSet()
13
14 -- liefert alle inneren Merkmale der Medienkomponente;
15 -- beachtet dabei Basismedienkomponenten und Neudefinition
16 -- (hier vereinfacht: keine Berücksichtigung der Zugriffsmodifizierer)
17 context aml::MediaComponent::getAllInnerProperties()
18         : Set(aml::InnerProperty)
19 body : let allInnerPropertiesNoRedef : Set(aml::InnerProperty) =
20          ownedInnerProperty->union(
21            allParents()
22            ->select(oclIsKindOf(aml::MediaComponent))
23            .oclAsType(aml::MediaComponent).ownedInnerProperty
24          )->asSet() in
```

```
25          allInnerPropertiesNoRedef -
26            allInnerPropertiesNoRedef.redefinedInnerProperty->asSet()
27
28  -- liefert alle inneren Merkmale der Medienkomponente, wobei auch innere
29  -- Merkmale abgefragt werden, die sich tiefer in der Hierarchie befinden;
30  -- beachtet außerdem Basismedienkomponenten und Neudefinition
31  context aml::MediaComponent::getAllInnerPropertiesIncludingSub()
32          : Set(aml::InnerProperty)
33  body : getAllInnerProperties()->union(
34            getAllInnerProperties().mediaComponentType
35            .getAllInnerPropertiesIncludingSub())
36
37  -- liefert die Liste der Verursacher des Sensors
38  context aml::OpponentSensor::getOpponents()
39          : Sequence(aml::MediaComponent)
40  body : opponentRelation.opponent
41
42  -- prüft, ob Neudefinition möglich ist, wobei typische Konsistenzprüfungen
43  -- (z. B. der zugeordnete Typ) nicht durchgeführt werden
44  context aml::InnerProperty::isConsistentWith(
45          redefinee : Redefinable Element) : Boolean
46  pre : redefinee.isRedefinitionContextValid(self)
47  body : redefinee.oclIsKindOf(aml::InnerProperty)
48
49  -- liefert das (direkt) übergeordnete innere Merkmal
50  -- im Kontext der aktuellen Medienkomponente
51  context aml::InnerProperty::getParent() : aml::InnerProperty
52  body : let ownerMediaComp : aml::MediaComponent =
53               owner.oclAsType(aml::MediaComponent) in
54         if (ownerMediaComp.owner.oclIsKindOf(aml::InnerProperty))
55         then ownerMediaComp.owner.oclAsType(aml::InnerProperty)
56         else null
57         endif
58
59  -- liefert die (direkt) untergeordneten inneren Merkmale
60  -- im Kontext der aktuellen Medienkomponente
61  context aml::InnerProperty::getChildren() : OrderedSet(aml::InnerProperty)
62  body : if (ownedType = null) then OrderedSet{}
63         else ownedType.ownedInnerProperty
64         endif
65
66  -- prüft, ob Neudefinition möglich ist, wobei vor allem geprüft wird,
67  -- ob die Parameterliste kompatibel ist (conformsTo)
68  context aml::Animator::isConsistentWith(
69          redefinee : Redefinable Element) : Boolean
70  pre : redefinee.isRedefinitionContextValid(self)
71  body : redefinee.oclIsKindOf(aml::Animator) and
72           let an: aml::Animator = redefinee.oclAsType(aml::Animator) in
73           self.ownedParameter->size() = an.ownedParameter->size() and
74           Set{1..an.ownedParameter->size()}
75           ->forAll(i | an.ownedParameter->at(i).type
76             .conformsTo(self.ownedParameter->at(i).type))
77
78  -- prüft, ob zwei Feature-Selektoren dasselbe Feature gewählt haben
79  context aml::FeatureSelector::hasSameFeature(
80          otherFeatureSelector : FeatureSelector) : Boolean
81  body : -- komplexe Abfrage --
```

Listing B.1: Operationen des AML-Metamodells

```
1  -- Eine Medienkomponente darf nur von Medienkomponenten abgeleitet werden.
2  context aml::MediaComponent
3  inv : general->forAll(oclIsKindOf(aml::MediaComponent))
4
5  -- Statische innere Merkmale sind nicht erlaubt.
6  context aml::InnerProperty
7  inv : isStatic = false
8
9  -- Ein abstraktes inneres Merkmal ist nur
10 -- in einer abstrakten Medienkomponente erlaubt.
11 context aml::InnerProperty
12 inv : (not(owner.oclAsType(aml::MediaComponent).isAbstract)) implies
13         (not(mediaComponentType.isAbstract))
14
15 -- Der zugewiesene Typ muss der möglicherweise
16 -- enthaltenen Medienkomponente entsprechen.
17 context aml::InnerProperty
18 inv : ownedType <> null implies
19         ownedType = mediaComponentType
20
21 -- Ein Sensor darf nur als Trigger innerhalb von Zustandsautomaten
22 -- verwendet werden, die auch zur Medienkomponente des Sensors gehören.
23 context aml::Sensor
24 inv : getModel().allOwnedElements()
25       ->select(oclIsKindOf(uml::Transition))
26       .oclAsType(uml::Transition).trigger
27       ->select(event = self)
28       .owner.oclAsType(uml::Transition)
29       .containingStateMachine().getContext().oclAsType(uml::Classifier)
30       ->forAll(cont | cont.allParents()->including(cont)
31       ->includes(self.sensorOwner))
32
33 -- Animationszustände können nur für Medienkomponenten modelliert werden.
34 context aml::AnimationState
35 inv : containingStateMachine()._context.oclIsKindOf(aml::MediaComponent)
36
37 -- Zwei Animationsanweisungen innerhalb desselben Animationszustands dürfen
38 -- nicht dasselbe Attribut animieren.
39 context aml::AnimationState
40 inv : animationInstruction->forAll(
41         i1 | animationInstruction->forAll(
42           i2 | i1 = i2 or
43           not(i1.hasSameFeature(i2))))
44
45 -- Zwei Animationsanweisungen, die in möglichen Parallelzuständen verwendet
46 -- werden, dürfen nicht dasselbe Attribut animieren.
47 context aml::AnimationInstruction
48 inv : -- komplexe Invariante --
49
50 -- Der verwendete Animator muss für die Medienkomponente definiert sein.
51 context aml::AnimationInstructionStep
52 inv : (animator <> null) implies
53         owner.owner.oclAsType(aml::AnimationState)
54         .containingStateMachine()
55         .getContext().oclAsType(aml::MediaComponent)
56         .getAllAnimators()
57         ->includes(self.animator)
58
59 -- Die Anzahl der Argumente muss zu den Animatorparametern passen.
60 context aml::AnimationInstructionStep
61 inv : if (animator = null)
62       then arguments->size() = 0
63       else arguments->size() = animator.ownedParameter
64         ->select(direction = uml::ParameterDirectionKind::_in)
65         ->size()
66       endif
67
```

```
68 -- Die Angabe zur Selektion eines inneren Merkmals muss konsistent sein.
69 context aml::FeatureSelector
70 inv : innerPropertyPath->size() = innerPropertyPathSelector->size()
71
72 -- Es wird kein Feature selektiert.
73 context aml::SendMessageAction
74 inv : feature->size() = 0 and
75         innerPropertyPath->size() = 0 and
76         innerPropertyPathSelector->size() = 0
77
78 -- Die Anzahl der Argumente muss zu Operationsparametern passen.
79 context aml::CallMethodAction
80 inv : arguments->size() = method.ownedParameter
81         ->select(direction = uml::ParameterDirectionKind::_in)->size()
82
83 -- Das ausgewählte Attribut darf nicht schreibgeschützt sein.
84 context aml::SetAttributeAction
85 inv : not(attribute.isReadOnly)
86
87 -- Das ausgewählte Attribut darf nicht schreibgeschützt sein.
88 context aml::CreateAction
89 inv : attribute <> null implies not(attribute.isReadOnly)
90
91 -- Es kann keine Instanz einer abstrakten Klasse erzeugt werden.
92 context aml::CreateAction
93 inv : type.isAbstract = false
```

Listing B.2: Zusicherungen des AML-Metamodells

B.2 Operationen und Zusicherungen des AML/GT-Profils

Die Operationen und Zusicherungen des AML/GT-Profils werden in Listing B.3 und Listing B.4 gezeigt. Die Listings enthalten einige Konstrukte, die derzeit von nur wenigen UML-Tools unterstützt werden. Sie sind daher kaum bekannt, aber zum Verständnis sehr wichtig. Zum einen wird das Präfix *base_* sehr häufig verwendet, um von einem angewendeten Stereotypen auf die dadurch erweiterte Metaklasse zuzugreifen. Beispielsweise kann *base_MediaComponent* im Kontext des Stereotyps *MediaComponentEx* benutzt werden, um auf die *MediaComponent*-Metaklasse zuzugreifen. Auch für den umgekehrten Weg gibt es ein Konstrukt. Hierfür muss das Präfix *extension_* verwendet werden, also *extension_MediaComponentEx* im genannten Beispiel. Diese Präfixe werden auch in der Spezifikation der UML erläutert [UML11].

```
1  -- prüft, ob die Medienkomponente einen Knoten spezifziert
2  context amlgtp::MediaComponentEx::isNodeComponent() : Boolean
3  body : type = amlgtp::MediaComponentType::node
4
5  -- prüft, ob die Medienkomponente eine Komponentenhyperkante spezifiziert
6  context amlgtp::MediaComponentEx::isEdgeComponent() : Boolean
7  body : type = amlgtp::MediaComponentType::edge or
8           type = amlgtp::MediaComponentType::edgeTopLevel
9
10 -- liefert die Anzahl der Knoten einer Medienkomponente,
11 -- die eine Komponentenhyperkante spezifiziert
```

```
12 context amlgtp::MediaComponentEx::numberOfNodes() : Integer
13 body : let numbers : Bag(uml::Integer) =
14          base_MediaComponent.getAllInnerPropertiesIncludingSub()->select(
15            prop : aml::InnerProperty | prop.extension_NodeEx <> null
16          ).extension_NodeEx.nodeNumber in
17        numbers->size()
18
19 -- prüft, ob das Attribut als Attribut einer Komponentenhyperkante
20 -- spezifiziert wird
21 context amlgtp::PropertyEx::isAttribute() : Boolean
22 body : type = amlgtp::PropertyType::attribute or
23          type = amlgtp::PropertyType::attributeEditable
24
25 -- prüft, ob die Assoziation als Relationshyperkante spezifiziert wird
26 context amlgtp::AssociationEx::isRelation() : Boolean
27 body : type = amlgtp::AssociationType::relation
28
29 -- prüft, ob die Assoziation als Mehrzweck-Hyperkante spezifiziert wird
30 context amlgtp::AssociationEx::isLink() : Boolean
31 body : type = amlgtp::AssociationType::link or
32          type = amlgtp::AssociationType::linkUnary
33
34 -- prüft, ob der Trigger das "inv broken"-Ereignis verwendet
35 context amlgtp::TriggerEx::isInvBrokenTrigger() : Boolean
36 body : base_Trigger.event <> null and
37          base_Trigger.event.name = 'inv broken' and
38          base_Trigger.event.oclIsKindOf(uml::ExecutionEvent)
```

Listing B.3: Operationen des AML/GT-Profils

```
1  -- Medienkomponenten für Komponentenhyperkanten
2  -- benötigen konsistente Knoten.
3  context amlgtp::MediaComponentEx
4  inv : let numbers : Bag(uml::Integer) =
5          base_MediaComponent.getAllInnerPropertiesIncludingSub()->select(
6            prop : aml::InnerProperty | prop.extension_NodeEx <> null
7          ).extension_NodeEx.nodeNumber in
8        isEdgeComponent() implies
9          (numberOfNodes() > 0 and
10           Bag{0..numberOfNodes() - 1} = numbers)
11
12 -- Für innere Merkmale muss eine Multiplizität von 1 spezifiziert werden.
13 context amlgtp::InnerPropertyEx
14 inv : base_InnerProperty.lower = 1 and base_InnerProperty.upper = 1
15
16 -- Eine Knotennummer kann einem inneren Merkmal nur zugewiesen werden,
17 -- wenn als Typ eine Knoten-Medienkomponente spezifiziert wird.
18 context amlgtp::Node
19 inv : nodeNumber >= 0 implies
20          base_InnerProperty.mediaComponentType <> null and
21          base_InnerProperty.mediaComponentType
22          .extension_MediaComponentEx.isNodeComponent()
23
24 -- Attribute für Hyperkanten oder editierbare Attribute müssen bestimmte
25 -- Kriterien erfüllen (sie dürfen bspw. nicht statisch oder
26 -- schreibgeschützt sein).
27 context amlgtp::PropertyEx
28 inv : type <> amlgtp::PropertyType::regular implies
29          base_Property.isReadOnly = false and
30          base_Property.isStatic = false and (
31          base_Property.owner.oclIsKindOf(aml::MediaComponent) or (
32            base_Property.owner.oclIsKindOf(uml::Association) and
33            base_Property.owner.oclAsType(uml::Association).isBinary() and
34            base_Property.owner.oclAsType(uml::Association).memberEnd
35           ->forAll(type.oclIsKindOf(aml::MediaComponent))))
```

```
36
37 -- Die Enden von "nicht-gewöhnlichen" Assoziationen
38 -- dürfen nicht als "geordnet" oder "eindeutig" modelliert werden.
39 context amlgtp::AssociationEx
40 inv : type <> amlgtp::AssociationType::regular implies
41         base_Association.memberEnd->forAll(not(isUnique or isOrdered))
42
43 -- Die Enden von Assoziationen für Mehrzweck- oder Relationshyperkanten
44 -- müssen mit Knoten-Medienkomponenten verbunden werden.
45 -- Zusätzlich müssen Assoziationen für Relationshyperkanten binär sein.
46 context amlgtp::AssociationEx
47 inv : (isRelation() or isLink()) implies
48         (isLink() or base_Association.isBinary()) and
49         base_Association.memberEnd->forAll(
50           type.oclIsKindOf(aml::MediaComponent) and
51           type.oclAsType(aml::MediaComponent)
52           .extension_MediaComponentEx.isNodeComponent())
53
54 -- Assoziationen für unäre Mehrzweck-Verbindungshyperkanten
55 -- müssen binär sein, wobei beide Enden mit derselben
56 -- Medienkomponente verbunden sein müssen.
57 context amlgtp::AssociationEx
58 inv : type = amlgtp::AssociationType::linkUnary implies
59         base_Association.isBinary() and
60         base_Association.memberEnd->at(1).type =
61           base_Association.memberEnd->at(2).type
62
63 -- Nur Hyperkanten-Medienkomponenten dürfen Sensoren besitzen.
64 context amlgtp::SensorEx
65 inv : base_Sensor.sensorOwner
66         .extension_MediaComponentEx.isEdgeComponent()
67
68 -- Verursacher müssen Hyperkanten-Medienkomponenten sein.
69 context amlgtp::OpponentRelationshipEx
70 inv : base_OpponentRelationship.opponent
71         .extension_MediaComponentEx.isEdgeComponent()
72
73 -- Nachrichtensensoren dürfen maximal einen Verursacher spezifizieren.
74 context amlgtp::MessageSensorEx
75 inv : base_MessageSensor.opponentRelation->size() <= 1
76
77 -- Zustandsautomaten sind nur für Hyperkanten-Medienkomponenten zulässig.
78 context amlgtp::StateMachineEx
79 inv : base_StateMachine.getContext() <> null and base_StateMachine
80         .getContext().oclIsKindOf(aml::MediaComponent) implies
81         base_StateMachine.getContext().oclAsType(aml::MediaComponent)
82         .extension_MediaComponentEx.isNodeComponent() = false
83
84 -- Erstellte Medienkomponenten müssen Hyperkanten-Medienkomponenten sein.
85 -- Bei der Erstellung von Hyperkanten-Medienkomponenten muss
86 -- außerdem die korrekte Knotenanzahl angegeben werden.
87 context amlgtp::CreateActionEx
88 inv : base_CreateAction.type.oclIsKindOf(aml::MediaComponent) implies
89         base_CreateAction.type.oclAsType(aml::MediaComponent)
90         .extension_MediaComponentEx.isEdgeComponent() and
91         base_CreateAction.type.oclAsType(aml::MediaComponent)
92         .extension_MediaComponentEx.getNumberOfNodes()
93           = base_CreateAction.constructorParams->size() - 1
94
95 -- Der "inv broken"-Trigger kann nur verwendet werden,
96 -- falls für den Zustand ein Invarianten-Graphmuster spezifiziert ist.
97 context amlgtp::TriggerEx
98 inv : let sourceVertex : uml::Vertex =
99         base_Trigger.owner.oclAsType(uml::Transition).source in
100      isInvBrokenTrigger() implies (
101        sourceVertex.oclIsKindOf(uml::State) and
102        sourceVertex.oclAsType(uml::State).stateInvariant <> null)
103
```

```
104 -- Transition mit "inv broken"-Trigger darf keine Schleife sein.
105 context amlgtp::TriggerEx
106 inv : isInvBrokenTrigger() implies
107         base_Trigger.owner.oclAsType(uml::Transition).source <>
108         base_Trigger.owner.oclAsType(uml::Transition).target
```

Listing B.4: Zusicherungen des AML/GT-Profils

B.3 Weitere verwendete Operationen

Im Rahmen der Übersetzungsregeln in Kapitel 7 werden zusätzlich folgende Operationen verwendet:

```
1  -- liefert den Zustandsautomateneigentümer für einen Knoten
2  context uml::Vertex::smMed() : aml::MediaComponent
3  body : self.containingStateMachine()._context.oclAsType(aml::MediaComponent)
4
5  -- liefert die für die Transition relevanten Aktionssequenzelemente
6  -- (bezieht "entry/exit"-Effekte der Zustände und
7  -- Effekte der Transition in richtiger Reihenfolge ein)
8  context uml::Transition::actSeq() : Sequence(aml::ActionSequenceElement)
9  body : if (source.oclIsKindOf(uml::State) and
10            source.oclAsType(uml::State).exit <> null) then
11           source.oclAsType(uml::State)
12           .exit.oclAsType(aml::ActionSequence).action->asSequence()
13         else Sequence {} endif
14         ->union(
15           effect.oclAsType(aml::ActionSequence).action->asSequence())
16         ->union(
17           if (target.oclIsKindOf(uml::State) and
18               target.oclAsType(uml::State).entry <> null) then
19             target.oclAsType(uml::State)
20             .entry.oclAsType(aml::ActionSequence).action->asSequence()
21           else Sequence {} endif)
22
23 -- liefert die Transition einer Region, welche den Initialzustand verlässt
24 context uml::Region::initTr() : uml::Transition
25 body : transition->any(source.oclIsKindOf(uml::Pseudostate) and
26           source.oclAsType(uml::Pseudostate).kind =
27             uml::PseudostateKind::initial)
28
29 -- liefert die Verschachtelungstiefe eines Knotenpunkts
30 context uml::Vertex::getDepth() : Integer
31 body : if (container = null or container.state = null)
32         then 0
33         else self.container.state.getDepth() + 1
34         endif
35
36 -- prüft, ob eine Komponentenhyperkante erzeugt werden muss
37 context aml::CreateAction::isForEdge() : Boolean
38 body : type.oclIsKindOf(aml::MediaComponent) and
39         type.oclAsType(aml::MediaComponent)
40         .extension_MediaComponentEx.isEdgeComponent()
41
42 -- liefert den Bezeichner der zu erstellenden Komponentenhyperkante
43 context aml::CreateAction::getLabel() : String
44 body : self.constructorArgs->at(1)
45
46 -- prüft, ob der Zustandsautomat der übergebenen Medienkomponente
47 -- die Region enthalten kann
48 context uml::Region::isValidFor(mediaComp : aml::MediaComponent) : Boolean
```

```
49 body : let mediaCompOfRegion : aml::MediaComponent = containingStateMachine()
50           .getContext().oclAsType(aml::MediaComponent) in
51         mediaCompOfRegion = mediaComp or
52         mediaCompOfRegion.allParents()->includes(mediaComp) or
53         mediaComp.allParents()->includes(mediaCompOfRegion)
54
55 -- liefert alle Regionen des Zustandsautomaten einer Medienkomponente,
56 -- wobei Zustandsautomaten von Basismedienkomponenten einbezogen werden
57 context aml::MediaComponent::allRegions() : Set(uml::Region)
58 body : objectsOfType(uml::Region)->select(
59           r : uml::Region |
60           allParents().oclAsType(uml::BehavioredClassifier)->including(self)
61           .classifierBehavior->includes(r.containingStateMachine()))
```

Listing B.5: Zusätzlich verwendete Operationen

Anhang C

Generierter Quellcode (Beispiele)

Nachfolgend werden zwei Code-Beispiele gezeigt. Der darin enthaltene Code kann auf Grundlage von AML/GT-Modellen und unter Verwendung des präsentierten Ansatzes generiert werden. Er enthält Abschnitte, die mit dem Kommentar „Start of user code“ beginnen und mit „End of user code“ enden. Dies sind Schlüsselwörter, die auf den verwendeten Code-Generator Acceleo [Acc08] zurückzuführen sind. Innerhalb solcher Abschnitte kann nach der Codegenerierung Code zur Implementierung von Operationen, Animatoren, eigenem Zeichencode (nicht gezeigt) oder speziellen TCRs manuell eingebettet werden. Nach einer erneuten Generierung – ggf. mit leichten Anpassungen – bleibt der Code innerhalb dieser Abschnitte erhalten.

Das Beispiel in Listing C.1 zeigt ausschnittweise den generierten Code für Medienkomponente *Marble* aus Abb. A.5, S. 273. Im Abschnitt „Zugriffsmethoden“ werden Methoden erzeugt, um alle relevanten Daten der Medienkomponente abzufragen. Dazu zählen auch Methoden für den Zugriff auf Attribute der Komponentenhyperkante, für den Zugriff auf verknüpfte Medienkomponenteninstanzen (inkl. untergeordneter Medienkomponenten) und die Abfrage zeitabhängiger Attributwerte (vgl. Definition 5.7, S. 148). Unterschiedliche Präfixe weisen auf die jeweilige Art der Zugriffsmethode hin. Das Präfix „S“ weist z. B. auf die Abfrage des statischen Werts eines Attributs hin, während „A“ auf die Abfrage des zeitabhängigen Werts zur Animationszeit hinweist. Da die Medienkomponente *Marble* eigentlich keinen Animator enthält, soll außerdem angenommen werden, dass innerhalb der Medienkomponente ein Animator namens *ComplexAnim* mit einem Parameter namens *param* vom Typ *double* existiert.

In Listing C.2 wird der generierte Klassencode für Zustand *Falling* aus Abb. A.6, S. 274, ausschnittweise dargestellt.

```
1
2  public abstract class Marble
3      extends avalanche.Piece
4      implements avalanche.IMarble {
5
6      // private Daten
7      /*...*/
8
9      // Zugriffsmethoden
10     /*...*/
11     public Double get_S_bodyMarbleRolling_yScale() { /*...*/ }
12     public double get_A_bodyMarbleRolling_yScale() { /*...*/ }
13     public void set_S_bodyMarbleRolling_yScale(Double newVal) { /*...*/ }
14     /*...*/
15
16     // Erzeugung von Animatorinstanzen
17     public diameta.IAnimator createAnimatorComplexAnim(
18             final Object startVal,
19             final Object targetVal,
20             final double startTime,
21             final Double duration,
22             final double param)
23         ) {
24         // Start of user code ANIMATOR_avalanche.Marble.ComplexAnim_param
25
26         return new diameta.IAnimator { /*...*/ };
27
28         // End of user code
29     }
30     /*...*/
31
32     // Zeitberechnungsregeln (TCRs)
33     public Double getCollisionTimeMarbleStop(
34             diameta.Context ctx,
35             avalanche.IMarble owner,
36             avalanche.ISwitch switch) {
37         // Start of user code COLLISION_TIME_avalanche.MarbleStop
38
39         return owner.getStateEnteredTime(
40                   owner.getCurrentState(
41                     avalanche.StateMachine_Marble.Movement.class)) +
42                Math.sqrt(((switch.get_P_yPos().getVal() -
43                   owner.get_P_yPos().getVal() - 25.0) * 2.0) / 140.0);
44
45         // End of user code
46     }
47     /*...*/
48
49     // Operationen
50     public void playSoundBlock() {
51         // Start of user code BODY_avalanche.Marble.PlaySoundBlock_void
52         /* ... */
53         // End of user code
54     }
55     /*...*/
56
57     // Medienkomponente zeichnen und Form berechnen
58     public void draw(java.awt.Graphics2D g2d) {
59         /* ... */ // vollständig generiert
60     }
61     public java.awt.Shape getShape() {
62         /* ... */ // vollständig generiert
63     }
64 }
```

Listing C.1: Generierter Quellcode für Medienkomponente

```
1
2  import java.util.*;
3
4  // vollständig generiert
5  public class Falling
6      extends diameta.AnimationState {
7
8      // Rückgabe des Eigentümers
9      public avalanche.IMarble getOwner() {
10         return (avalanche.IMarble) super.getOwner();
11     }
12
13     // Abfrage untergeordneter und übergeordneter Regionen
14     public Class<? extends diameta.Region> getRegionType() {
15         return avalanche.StateMachine_Marble.Movement.class;
16     }
17     public Set<Class<? extends diameta.Region>> getSubRegionTypes() {
18         return new HashSet<Class<? extends diameta.Region>>();
19     }
20
21     // Anwendung von Animationsanweisungen auf Attribute des Eigentümers
22     // (inkl. Erzeugung und Aktivierung von Animatorinstanzen)
23     protected void applyAnimationInstructions() {
24         /* ... */
25     }
26 }
```

Listing C.2: Generierter Quellcode für Animationszustand

Abkürzungsverzeichnis

AAS	Abstraktes Animationssystem (*abstract animation system*)
AAS/GT	AAS mittels Graphen und GTs
AML	Animation Modeling Language
AML/GT	AML for Graph Transformations
API	Schnittstelle zur Anwendungsprogrammierung (*application program interface*)
ASG	Abstrakter Syntaxgraph
B/E-Netz	Bedingungs-/Ereignis-Netz
CHR	Entfernung einer Komponentenhyperkante (*component hyperedge removal*)
CIM	Computation Independent Model
CSP	Constraint-Satisfaction-Problem
DEVS	Discrete Event System Specification
DPO	Double-Pushout
DSL	Domänenspezifische Sprache (*domain-specific language*)
DTD	Document Type Definition (`http://www.w3.org/TR/xml/`)
EBNF	Erweiterte Backus-Naur-Form
EMF	Eclipse Modeling Framework (`http://www.eclipse.org/modeling/emf/`)
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GMP	Graphical Modeling Project (`http://www.eclipse.org/modeling/gmp/`)
GPL	General Purpose Language
GT	Graphtransformation
GTP	Graphtransformationsprogramm
GTR	Graphtransformationsregel
GTS	Graphtransformationssystem
GUI	Grafische Benutzeroberfläche (*graphical user interface*)

IDE	Integrierte Entwicklungsumgebung (*integrated development environment*)
KSG	Konkreter Syntaxgraph
LHS	Linke Seite (*left-hand side*)
M2M	Modell-zu-Modell
M2T	Modell-zu-Text
MDA	Model Driven Architecture (`http://www.omg.org/mda/`)
MML	Multimedia Modeling Language
MOF	Meta Object Facility (`http://www.omg.org/spec/MOF/`)
OCL	Object Constraint Language (`http://www.omg.org/spec/OCL/`)
OMG	Object Management Group (`http://www.omg.org`)
OOAD	Objektorientierte Analyse und Design
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
QVTO	Operational QVT Language
RHS	Rechte Seite (*right-hand side*)
SPO	Single-Pushout
SSIML	Scene Structure and Integration Modelling Language
TCR	Zeitberechnungsregel (*time calculation rule*)
UI	Benutzerschnittstelle (*user interface*)
UML	Unified Modeling Language (`http://www.omg.org/spec/UML/`)
VI	Virtuelles Instrument
VP	Visuelle Programmierung
VRML	Virtual Reality Modeling Language
XAML	Extensible Application Markup Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Symbolverzeichnis

Allgemein

$\mathbb{B} = \{wahr, falsch\}$	Menge der Booleschen Werte	52
$t \in \mathbb{T}$	absoluter Zeitpunkt	100
$\omega \in \mathbb{T}^\omega$	unerreichbarer Zeitpunkt	100
$\mathbf{P}(X)$	Potenzmenge der Menge X	52
$\|X\|$	Kardinalität der Menge X	52
ε	leeres Wort	52
$\|w\|$	Länge des Worts w	52
$[w]$	Menge aller in Wort w vorkommenden Symbole	52
X^*	Menge aller Wörter über X	52
X^n	Menge aller Wörter über X der Länge n	52
$f\|_X$	Einschränkung der Funktion f auf X	52
f^*	symbolweise Erweiterung der Funktion f	52
$\lfloor x \rfloor$	Abrundungsfunktion	133
$max/min\ M$	größtes/kleinstes Element der Menge M	133
$B \sqsubset A$	Typ B ist abgeleitet von Typ A	54
id_X	identische Abbildung auf die Menge X	229

Hypergraphen und Hypergraphmorphismen

$lbl \in \mathcal{L}$	Markierung aus dem Markierungsalphabet	52
$\mathcal{LV} = (\mathcal{L}, \mathcal{V})$	Markierungsalphabet mit Atributen	52
$arity(lbl)$	Stelligkeit der Markierung lbl	52
$\mathcal{CMP} \subseteq \mathcal{L}$	Markierungen für Komponentenhyperkanten	55
$\mathcal{LNK} \subseteq \mathcal{L}$	Markierungen für Verbindungshyperkanten	55
$\mathcal{REL} \subseteq \mathcal{LNK}$	Markierungen für Relationshyperkanten	55
$\mathcal{GEN} \subseteq \mathcal{LNK}$	Markierungen für Mehrzweck-Hyperkanten	55
$H \in \mathcal{H}_\mathcal{L}$	Hypergraph (auch L, K, R, D)	52
$H^\emptyset$	leerer Hypergraph	60
$H_{e:lbl}$	elementarer Hypergraph (Kante e, Markierung lbl)	60
$H_A \in \mathcal{H}_\mathcal{LV}$	attributierter Hypergraph, wobei $H_A = (H, A)$	54
$v \in V_H$	Knoten des Hypergraphen H	52

$e \in E_H$	(Hyper-)Kante des Hypergraphen H	52
$e^{\emptyset} \notin E_H$	Null-Hyperkante	249
$E_H\|_{\mathcal{X}} \subseteq E_H$	Menge der Kanten mit Markierung $lbl \in \mathcal{X}$	59
$vis_H(e)$	besuchte Knoten der Kante e	52
$\overline{vis}_H(v)$	Menge der Kanten, die Knoten v besuchen	59
$lab_H(e)$	Markierung der Kante e	52
$H_1 \xrightarrow{m} H_2$	Hypergraphmorphismus, wobei $m = (m_V, m_E)$	61
$H_1 \overset{p}{\rightharpoonup} H_2$	partieller Hypergraphmorphismus	62
$H_1 \overset{m}{\rightarrowtail} H_2$	Hypergraphmonomorphismus	63
$H' \overset{m}{\hookrightarrow} H$	Hypergraphinklusion	63

Graphtransformationen

$egtr \in \mathcal{EG}_{\mathcal{L}}$	einfache Graphtransformationsregel	65
$egtr_L^{\emptyset} \in \mathcal{EG}_{\mathcal{L}}$	einfache Graphtransformationsregel ohne Änderung	65
$acr \in \mathcal{ACR}_{R,\mathcal{LV}}$	Attributberechnungsregel	70
$acr^{\emptyset} \in \mathcal{ACR}_{R,\mathcal{LV}}$	Attributberechnungsregel ohne Änderung	70
$con \in \mathcal{CON}_{L,\mathcal{LV}}$	funktionale Anwendungsbedingung	70
$nac \in \mathcal{NAC}_{L,\mathcal{LV}}$	negative Anwendungsbedingung	71
$(m, H_A) \models x$	Erfüllung der Anwendungsbedingung x	70
$fgtr \in \mathcal{FG}_{\mathcal{LV}}$	erweiterte Graphtransformationsregel	71
$remove_{del} \in \mathcal{DG}_{\mathcal{L}}$	GTR zum Entfernen einer Kante (Markierung del)	74
$pgtr \in \mathcal{PG}_{\mathcal{LV}}$	GTR mit Graphtransformationsprogramm	78
$(m, H_A) \models gtr$	Anwendbarkeit von GTR gtr (Match m)	65
$(p, H_A) \models\!\!\approx gtr$	Anwendbarkeit von GTR gtr (Teilmatch p)	65
$H_A \overset{gtr,m,\ldots}{\Longrightarrow} H_{A'}^{\dagger}$	Anwendung von GTR gtr (Match m)	65
$H_A \overset{gtr,p}{\rightsquigarrow} H_{A'}^{\dagger}$	Anwendung von GTR gtr (Teilmatch p)	65
$H_{A'}^{\dagger} \overset{acr,m^{\dagger}}{\Longrightarrow} H_{A^{\dagger}}^{\dagger}$	Anwendung von Attributberechnungsregel acr	70
$gtp \in \mathcal{GTP}_{\overline{R},\mathcal{LV}}$	Graphtransformationsprogramm	80
$gtp^{\emptyset} \in \mathcal{GTP}_{\overline{R},\mathcal{LV}}$	leeres GTP	80
$\overline{gtp}^{\emptyset} \in \mathcal{GTP}_{\overline{R},\mathcal{LV}}$	niemals anwendbares GTP	80
$(\overline{m}^{\dagger}, \overline{H}^{\dagger}, H_A) \models gtp$	Anwendbarkeit von GTP gtp	81
$H_A \overset{gtp,\overline{m}^{\dagger},\overline{H}^{\dagger}}{\Longrightarrow} H_{A^{\dagger}}^{\dagger}$	Anwendung von GTP gtp	81
$gtr(b)$	GTP: Anwendung von GTR gtr (Matchvorgabe b)	80
$gtp_1 \,;\, gtp_2$	GTP: GTP-Sequenz	80
$gtp_1 \mid gtp_2$	GTP: GTP-Alternative	80
$(gtp')^{\geq c}$	GTP: Anzahl c der mindestens anwendbaren GTRs	80
foreach … do …	GTP: „foreach“-Schleife	80

Abstraktes Animationssystem (AAS)

$z \in Z$	Zustand	101
$q \in Q$	Ereignis	101
$\tilde{q} \in \tilde{Q}$	internes Ereignis	101
$\hat{q} \in \hat{Q}$	externes Ereignis	101

AAS mittels Graphen und GTs (AAS/GT)

Animation Modeling Language (AML)

$\tilde{d} \in \mathbb{T}^{\phi}$	Animatorkonfiguration: angegebene Dauer	145
$arg \in {\mathcal{V}_{\mathcal{T}}}^{argn}$	Animatorkonfiguration: Argumentwerte (Anzahl $argn$)	145
$\mathcal{CFG}_{sig} \subseteq \mathcal{CFG}_{\mathcal{T}}$	zu Signatur sig kompatible Animatorkonfigurationen	145
$an \in \mathcal{AN}_{\mathcal{T}}$	Animator	145
$aow \in \mathcal{M}$	Animator: Eigentümer	145
af	Animator: Animatorfunktion	145
ad	Animator: Animationsdauerfunktion	145
$ian \in \mathcal{IAN}_{\mathcal{T}}$	Animatorinstanz	146
$\overline{ian} \notin \mathcal{IAN}_{\mathcal{T}}$	inaktive Animatorinstanz	146
$app(obj, attr)$	aktive Animatorinstanz von obj und $attr$	146
$\check{t}(ian)$	tatsächlicher Endzeitpunkt	147
$\check{x}(ian)$	tatsächlicher Zielwert	147
$saval(v, attr)$	statischer Attributwert	133
$taval(obj, attr, t)$	zeitabhängiger Attributwert	148
$cs \in \mathcal{CS}_{\mathcal{T}}$	Bedingungssensor	161
$csow \in \mathcal{M}$	Bedingungssensor: Eigentümer	161
$csop \in \mathcal{M}^{csopn}$	Bedingungssensor: Verursacher (Anzahl $csopn$)	161
csc	Bedingungssensor: Prädikat	161

Generierung

$ao \in \mathcal{AO}$	UML-Assoziation	222
$mc \in \mathcal{MC}$	UML-Medienkomponente	222
$cst \in \mathcal{CST}$	UML-Zusicherung	228
$te \in \mathcal{TE}$	UML-Zeitereignis	228
$us \in \mathcal{US}$	AML-Benutzersensor	228
$ms \in \mathcal{MS}$	AML-Nachrichtensensor	228
$sa \in \mathcal{SA}$	AML-*SendMessageAction*-Element	228
$ca \in \mathcal{CA}$	AML-*CreateAction*-Element	228
$rg \in \mathcal{RG}$	UML-Region	228
$vx \in \mathcal{VX}$	UML-Knotenpunkt	228
$st \in \mathcal{ST}$	UML-Zustand	228
$tr \in \mathcal{TR}$	UML-Transition	228
$strc \in \mathcal{STRC}$	Bedingungs-Code (Zeichenkette)	229
$\langle\langle \ldots \rangle\rangle$	Java-Code zur Umsetzung einer Funktion	230
$\{\{\ldots\}\}$	Generierter Teil innerhalb von Java-Code	230
$gpattern(cst)$	Graphmuster in cst	229
$gpatternR(cst, tr)$	$gpattern$ nach Übereinstimmungs-Anweisungen (rhs)	229
$gpatternL(cst, tr)$	$gpattern$ nach Übereinstimmungs-Anweisungen (lhs)	229
$gcons(cst)$	Bedingungs-Code in cst (positiv)	229
$gnacs(cst)$	Bedingungs-Codes inkl. Graphmuster in cst (negativ)	229
$gcreate(ca)$	elementarer Hypergraph in ca	229
$\nearrow (H)$	Hypergraph H ohne Verbindungshyperkanten	229
$cscons(strc)$	Funktionale Anwendungsbedingungen in $strc$	230
$gtime(te)$	Ausdruck für Zeitpunkt $t \in \mathbb{T}^{\omega}$ (Laufzeitauswertung)	255

Verzeichnis der AML- und AML/GT-Symbole

Medienkomponenten

Sensoren

Animatormodifikation

Abbildungsverzeichnis

Beispielverzeichnis

Definitionsverzeichnis

Verzeichnis der Übersetzungsregeln

Verzeichnis der Listings und Algorithmen

Tabellenverzeichnis

Literaturverzeichnis

[Acc08] Obeo: *Acceleo User Guide, 2.6.* Webseite. `http://www.acceleo.org/`. Version: 2008

[AKRS06] Amelunxen, Carsten ; Königs, Alexander ; Rötschke, Tobias ; Schürr, Andy: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, Arend (Hrsg.) ; Warmer, Jos (Hrsg.): *Proceedings of the Second European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'06)* Bd. 4066, Springer-Verlag, 2006 (Lecture Notes in Computer Science), S. 361–375

[Arg11] *ArgoUML, Version 0.34.* Webseite. `http://argouml.tigris.org/`. Version: Dezember 2011

[AT01] Ali, Jauhar ; Tanaka, Jiro: Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams. In: *ACIS International Journal of Computer & Information Science* 2 (2001), März, Nr. 1, S. 24–36

[AVL62] Adel'son-Vel'skij, G. M. ; Landis, E. M.: An Algorithm for the Organization of Information. In: *Soviet Mathematics Doklady* (1962), Nr. 3, S. 1259–1263

[BA94] Burnett, Margaret M. ; Ambler, Allen L.: Interactive Visual Data Abstraction in a Declarative Visual Programming Language. In: *Journal of Visual Languages & Computing* 5 (1994), Nr. 1, S. 29–60

[Baa06] Baar, Thomas: OCL and Graph-Transformations – A Symbiotic Alliance to Alleviate the Frame Problem. In: Bruel, Jean-Michel (Hrsg.): *MoDELS Satellite Events* Bd. 3844, Springer-Verlag, 2006 (Lecture Notes in Computer Science), S. 20–31

[Bar84] Barendregt, Hendrik P.: *Studies in Logic and the Foundations of Mathematics.* Bd. 103: *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984

[Bar98] Bardohl, Roswitha: GENGED – A Generic Graphical Editor for Visual Languages Based on Algebraic Graph Grammars. In: *P*roceedings of the 1998 IEEE Symposium on Visual Languages (VL'98)[IEE98], S. 48–55

[BB94] Burnett, Margaret ; Baker, Marla: A Classification System for Visual Programming Languages. In: *Journal of Visual Languages & Computing* 5 (1994), Nr. 3, S. 287–300

[BCR00] Börger, Egon ; Cavarra, Alessandra ; Riccobene, Elvinia: Modeling the Dynamics of UML State Machines. In: Gurevich, Yuri (Hrsg.) ; Kutter, Philipp W. (Hrsg.) ; Odersky, Martin (Hrsg.) ; Thiele, Lothar (Hrsg.): *Proceedings of the International Workshop on Abstract State Machines: Theory and Applications (ASM'00)* Bd. 1912, Springer-Verlag, 2000 (Lecture Notes in Computer Science), S. 223–241

[BCS96] Byrne, Michael ; Catrambone, Richard ; Stasko, John T.: Do Algorithm Animations Aid Learning? / GIT-GVU-96-18, GVU Center, Georgia Institute of Technology. Atlanta, GA, USA, August 1996. – Forschungsbericht

[BDM02] Bernardi, Simona ; Donatelli, Susanna ; Merseguer, José: From UML Sequence Diagrams and Statecharts to Analysable Petri Net models. In: *Proceedings of the 3rd International Workshop on Software and Performance (WOSP'02)*, 2002, S. 35–45

[BDUW08] Buchmann, Thomas ; Dotor, Alexander ; Uhrig, Sabrina ; Westfechtel, Bernhard: Model-Driven Software Development with Graph Transformations: A Comparative Case Study. In: [SNZ08], S. 345–360

[Bee94] Beeck, Michael von d.: A Comparison of Statecharts Variants. In: [LRV94], S. 128–148

[BEHE08] Biermann, Enrico ; Ermel, Claudia ; Hurrelmann, Jonas ; Ehrig, Karsten: Flexible visualization of automatic simulation based on structured graph transformation. In: *VL/HCC*, 2008, S. 21–28

[BEL+03] Bardohl, Roswitha ; Ehrig, Hartmut ; Lara, Juan D. ; Runge, Olga ; Taentzer, Gabriele ; Weinhold, Ingo: Node Type Inheritance Concept for Typed Graph Transformation / TU Berlin. 2003 (19). – Forschungsbericht

[BELT04] Bardohl, Roswitha ; Ehrig, Hartmut ; Lara, Juan de ; Taentzer, Gabriele: Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In: Wermelinger, Michel (Hrsg.) ; Margaria,

Tiziana (Hrsg.): *Proceedings of 7th International Conference on the Fundamental Approaches to Software Engineering (FASE'04)* Bd. 2984, Springer-Verlag, 2004 (Lecture Notes in Computer Science), S. 214–228

[BG03] Burmester, Sven ; Giese, Holger: The Fujaba Real-Time Statechart Plugin. In: Giese, Holger (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proceedings of the 1st International Fujaba Days 2003*, 2003

[BH08] Bruck, James ; Hussey, Kenn: Customizing UML: Which Technique is Right for You? / IBM. 2008. – Forschungsbericht

[BK00] Berthold, Ingrid Michael R. F. Michael R. Fischer ; Koch, Manuel: Attributed Graph Transformation with Partial Attribution. In: Ehrig, Hartmut (Hrsg.) ; Taentzer, Gabriele (Hrsg.): *Proceedings of the Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*, 2000 (Technical Report), S. 171–178

[BKPPT00] Bottoni, Paolo ; Koch, Manuel ; Parisi-Presicce, Francesco ; Taentzer, Gabriele: Consistency Checking and Visualization of OCL Constraints. In: [EKS00], S. 294–308

[BKPPT01] Bottoni, Paolo ; Koch, Manuel ; Parisi-Presicce, Francesco ; Taentzer, Gabriele: A Visualization of OCL Using Collaborations. In: [GK01], S. 257–271

[BLL85] Billstein, Rick ; Libeskind, Shlomo ; Lott, Johnny W.: *LOGO: MIT LOGO for the Apple*. Redwood City, CA, USA : Benjamin-Cummings Publishing Co., Inc., 1985

[BNBK06] Balasubramanian, Daniel ; Narayanan, Anantha ; Buskirk, Christopher P. ; Karsai, Gabor: The Graph Rewriting and Transformation Language: GReAT. In: [ZV06]

[Bod06] Bode, Helmut: *MATLAB-SIMULINK: Analyse und Simulation dynamischer Systeme*. 2. Auflage. Vieweg+Teubner Verlag, 2006

[BP01] Baresi, Luciano ; Pezzè, Mauro: On Formalizing UML with High-Level Petri Nets. In: Agha, Gul (Hrsg.) ; Cindio, Fiorella de (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Proceedings of Concurrent Object-Oriented Programming and Petri Nets* Bd. 2001, Springer-Verlag, 2001 (Lecture Notes in Computer Science), S. 276–304

[Bri10] Brieler, Florian: *A Generic Approach to the Recognition and Analysis of Sketched Diagrams Using Context Information*, Universität der Bundeswehr München, Fakultät für Informatik, Diss., 2010

[BRJ06] Booch, Grady ; Rumbaugh, James ; Jacobson, Ivar: *Das UML-Benutzerhandbuch.* Addison-Wesley, 2006

[Bro88] Brown, Marc H.: Perspectives on Algorithm Animation. In: O'Hare, J. J. (Hrsg.): *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'88).* New York, NY, USA : ACM, 1988, S. 33–38

[BSE11] Bandener, Nils ; Soltenborn, Christian ; Engels, Gregor: Extending DMM Behavior Specifications for Visual Execution and Debugging. In: Malloy, Brian A. (Hrsg.) ; Staab, Steffen (Hrsg.) ; Brand, Mark van d. (Hrsg.): *Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10)* Bd. 6563, Springer-Verlag, 2011 (Lecture Notes in Computer Science), S. 357–376

[BW02] Brucker, Achim D. ; Wolff, Burkhart: A Proposal for a Formal OCL Semantics in Isabelle/HOL. In: Carreño, Victor (Hrsg.) ; Muñoz, César A. (Hrsg.) ; Tahar, Sofiène (Hrsg.): *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'02)* Bd. 2410, Springer-Verlag, 2002 (Lecture Notes in Computer Science), S. 99–114

[CBC96] Carlson, Paul ; Burnett, Margaret M. ; Cadiz, Jonathan J.: A Seamless Integration of Algorithm Animation into a Visual Programming Language. In: Catarci, Tiziana (Hrsg.) ; Costabile, Maria F. (Hrsg.) ; Levialdi, Stefano (Hrsg.) ; Santucci, Giuseppe (Hrsg.): *Proceedings of the 1996 International Workshop on Advanced Visual Interfaces (AVI'96)*, ACM, 1996, S. 194–202

[CCGL10] Cabot, Jordi ; Clarisó, Robert ; Guerra, Esther ; Lara, Juan de: Synthesis of OCL Pre-conditions for Graph Transformation Rules. In: Tratt, Laurence (Hrsg.) ; Gogolla, Martin (Hrsg.): *Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations (ICMT'10)* Bd. 6142, 2010 (Lecture Notes in Computer Science), S. 45–60

[CDP03] Cooper, Stephen ; Dann, Wanda ; Pausch, Randy: Teaching Objects-first in Introductory Computer Science. In: Grissom, Scott (Hrsg.) ; Knox, Deborah (Hrsg.) ; Joyce, Daniel T. (Hrsg.) ; Dann, Wanda (Hrsg.): *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'03)*, ACM, 2003, S. 191–195

[CE94] Carstensen, Martin ; Ebert, Jürgen: Ansatz und Architektur von KOGGE / Universität Koblenz-Landau, Institut für Softwaretechnik. 1994 (2/94). – Forschungsbericht

[CE00] Czarnecki, Krzysztof ; Eisenecker, Ulrich W.: *Generative Programming – Methods, Tools and Applications.* Addison-Wesley, 2000

[CEKR02] Corradini, Andrea (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Proceedings of the 1st International Conference on Graph Transformation (ICGT'02).* Bd. *2505.* Springer-Verlag, 2002 (Lecture Notes in Computer Science)

[CFH09] Cox, Philip (Hrsg.) ; Fish, Andrew (Hrsg.) ; Howse, John (Hrsg.): *Workshop on Visual Languages and Logic, Satellite event of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'09).* Bd. *510.* 2009 (CEUR Workshop Proceedings)

[Cha87] Chang, Shi-Kuo: Visual Languages: A Tutorial and Survey. In: Gorny, Peter (Hrsg.) ; Tauber, Michael (Hrsg.): *Visualization in Programming* Bd. 282. Springer-Verlag, 1987, S. 1–23

[Chu36] Church, Alonzo: An Unsolvable Problem of Elementary Number Theory. In: *American Journal of Mathematics* 58 (1936), S. 345–363

[CHZ95] Citrin, W. ; Hall, R. ; Zorn, B.: Programming with visual expressions. In: *Proceedings of the 1990 IEEE Symposium on Visual Languages (VL'95)*, IEEE Computer Society, 1995, S. 294–301

[CJ05] Chauvel, Franck ; Jézéquel, Jean-Marc: Code Generation from UML Models with Semantic Variation Points. In: Briand, Lionel C. (Hrsg.) ; Williams, Clay (Hrsg.): *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)* Bd. 3713, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 54–68

[CJKW07] Cook, Steve ; Jones, Gareth ; Kent, Stuart ; Wills, Alan C.: *Domain Specific Development with Visual Studio DSL Tools.* Addison-Wesley, 2007

[CK01] Cengarle, María V. ; Knapp, Alexander: A Formal Semantics for OCL 1.4. In: [GK01], S. 118–133

[CK09] Cramer, Bastian ; Kastens, Uwe: Animation automatically generated from simulation specifications. In: DeLine, Robert (Hrsg.) ; Minas, Mark (Hrsg.) ; Erwig, Martin (Hrsg.): *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'09)*, IEEE Computer Society, 2009, S. 157–164

[CLOT97] Costagliola, Gennaro ; Lucia, Andrea D. ; Orefice, Sergio ; Tortora, Genoveffa: A Framework of Syntactic Models for the Implementation of Visual Languages. In: *P*roceedings of the 1997 IEEE Symposium on Visual Languages (VL'97)[IEE97], S. 58–65

[CM03] Chok, Sitt S. ; Marriott, Kim: Automatic Generation of Intelligent Diagram Editors. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 10 (2003), Nr. 3, S. 244–276

[CMR+97] Corradini, Andrea ; Montanari, Ugo ; Rossi, Francesca ; Ehrig, Hartmut ; Heckel, Reiko ; Löwe, Michael: Algebraic Approaches to Graph Transformation – Part I: Basic Concepts and Double Pushout Approach. In: [Roz97], Kapitel 3, S. 163–246

[CMR02] Crespo, Yania ; Marqués, José Manuel ; Rodríguez, Juan José: On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies. In: Black, Andrew P. (Hrsg.) ; Ernst, Erik (Hrsg.) ; Grogono, Peter (Hrsg.) ; Sakkinen, Markku (Hrsg.): *Proceedings of the Inheritance Workshop (ECOOP'02)*, 2002, S. 30–37

[CODA+11] Chimiak-Opoka, Joanna D. ; Demuth, Birgit ; Awenius, Andreas ; Chiorean, Dan ; Gabel, Sebastien ; Hamann, Lars ; Willink, Edward D.: OCL Tools Report based on the IDE4OCL Feature Model. In: Cabot, Jordi (Hrsg.) ; Clarisó, Robert (Hrsg.) ; Gogolla, Martin (Hrsg.) ; Wolff, Burkhart (Hrsg.): *Proceedings of the 11th International Workshop on OCL and Textual Modelling (OCL'11)* Bd. 44, 2011 (Electronic Communications of the EASST)

[Con97] Conway, Matthew J.: *Alice: Easy–to–Learn 3D Scripting for Novices*, University of Virginia, Faculty of the School of Engineering and Applied Science, Diss., Dezember 1997

[Cra06] Crane, Michelle L.: On the Syntax and Semantics of State Machines / Queen's University. 2006. – Forschungsbericht

[Cra10] Cramer, Bastian: *Generierung von Animation und Simulation für graphische Strukturediteren*, Universität Paderborn, Fakultät für Elektrotechnik, Informatik und Mathematik, Diss., September 2010

[DIN66] *DIN 66001 – Informationsverarbeitung, Sinnbilder für Datenfluss- und Programmablaufpläne.* DIN-Norm, September 1966

[DKH97] Drewes, Frank ; Kreowski, Hans-Jörg ; Habel, Annegret: Hyperedge Replacement, Graph Grammars. In: [Roz97], Kapitel 2, S. 95–162

[DMH11] Diprose, James P. ; MacDonald, Bruce A. ; Hosking, John G.: Ruru: A spatial and interactive visual programming language for novice robot programming. In: Costagliola, Gennaro (Hrsg.) ; Ko, Andrew J. (Hrsg.) ; Cypher, Allen (Hrsg.) ; Nichols, Jeffrey (Hrsg.) ; Scaffidi, Christopher (Hrsg.) ; Kelleher, Caitlin (Hrsg.) ; Myers, Brad A. (Hrsg.): *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*, IEEE Computer Society, 2011, S. 25–32

[Dmi04] Dmitriev, Sergey: Language Oriented Programming: The Next Programming Paradigm. In: *JetBrains 'onBoard' electronic monthly magazine* (2004), November. `www.onboard.jetbrains.com/articles/04/10/lop/`

[Dor11] Dormans, Joris: Simulating Mechanics to Study Emergence in Games. In: *Workshop on Artificial Intelligence in the Game Design Process (IDP'11), Papers from the 2011 AIIDE Workshop* Bd. WS-11-19, AAAI, 2011 (AAAI Workshops)

[DW80] Dörfler, W. ; Waller, D. A.: A category-theoretical approach to hypergraphs. In: *Archiv der Mathematik* 34 (1980), S. 185–192

[EA112] SparxSystems: *Enterprise Architect, Version 10*. Webseite. `http://www.sparxsystems.com/products/ea/`. Version: August 2012

[Ecl12] Eclipse Foundation: *Eclipse*. Webseite. `http://www.eclipse.org/`. Version: November 2012

[ECP95] Esteban, Olivier ; Chatty, Stéphane ; Palanque, Philippe A.: Whizz'Ed: a visual environment for building highly interactive software. In: Nordby, Knut (Hrsg.) ; Helmersen, Per H. (Hrsg.) ; Gilmore, David J. (Hrsg.) ; Arnesen, Svein A. (Hrsg.): *Proceedings of the International Conference on Human-Computer Interaction (INTERACT'95)*, Chapman & Hall, 1995, S. 121–126

[EE05] Ermel, Claudia ; Ehrig, Karsten: View Transformation in Visual Environments applied to Algebraic High-Level Nets. In: *Electronic Notes in Theoretical Computer Science* 127 (2005), Nr. 2, S. 61–86

[EEHT05] Ehrig, Karsten ; Ermel, Claudia ; Hänsgen, Stefan ; Taentzer, Gabriele: Towards Graph Transformation Based Generation of Visual Editors Using Eclipse. In: *Electronic Notes in Theoretical Computer Science* 127 (2005), Nr. 4, S. 127–143

[EEKR00] Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98)*. Bd. *1764*. Springer-Verlag, 2000 (Lecture Notes in Computer Science)

[EEPPR04] Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Parisi-Presicce, Francesco (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Proceedings of the 2nd International Conference on Graph Transformation (ICGT'04)*. Bd. *3256*. Springer-Verlag, 2004 (Lecture Notes in Computer Science)

[EG07] Ehrig, Karsten (Hrsg.) ; Giese, Holger (Hrsg.): *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'07)*. Bd. *6*. 2007 (Electronic Communications of the EASST)

[EH86] In: Ehrig, Hartmut ; Habel, Annegret: *Graph Grammars with Application Conditions*. Springer-Verlag, 1986, S. 87–100

[EHHS00] Engels, Gregor ; Hausmann, Jan H. ; Heckel, Reiko ; Sauer, Stefan: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: [EKS00], S. 323–337

[EHK+97] Ehrig, Hartmut ; Heckel, Reiko ; Korff, Martin ; Löwe, Michael ; Ribeiro, Leila ; Wagner, Annika ; Corradini, Andrea: Algebraic Approaches to Graph Transformation – Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In: [Roz97], Kapitel 4, S. 247–312

[EHKZ05] Ermel, Claudia ; Hölscher, Karsten ; Kuske, Sabine ; Ziemann, Paul: Animated Simulation of Integrated UML Behavioral Models Based on Graph Transformation. In: Erwig, Martin (Hrsg.) ; Schürr, Andy (Hrsg.): *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, IEEE Computer Society, 2005, S. 125–133

[EJ01] Esser, Robert ; Janneck, Jörn W.: Moses – a tool suite for visual modeling of discrete-event systems. In: *Proceedings of the 2001 IEEE Symposium on Human Centric Computing Languages and Environments (HCC'03)*, IEEE Computer Society, 2001

[EK99] Evans, Andy ; Kent, Stuart: Core Meta-Modelling Semantics of UML: The pUML Approach. In: France, Robert B. (Hrsg.) ; Rumpe, Bernhard (Hrsg.): *UML 1999 – The Unified Modeling Language, Beyond the Standard, 2nd International Conference* Bd. 1723, Springer-Verlag, 1999 (Lecture Notes in Computer Science), S. 140–155

[EK00] Engstrom, Eric ; Krueger, Jonathan: Building and Rapidly Evolving Domain-Specific Tools with DOME. In: *Proceedings of the 1997 IEEE Symposium on Visual Languages (CACSD'00)*, IEEE Computer Society, 2000, S. 83–88

[EKS00] Evans, Andy (Hrsg.) ; Kent, Stuart (Hrsg.) ; Selic, Bran (Hrsg.): *UML 2000 – The Unified Modeling Language, Advancing the Standard, 3rd International Conference*. Bd. *1939*. Springer-Verlag, 2000 (Lecture Notes in Computer Science)

[EMF12] Eclipse Foundation: *EMF – Eclipse Modeling Framework*. Webseite. `http://www.eclipse.org/modeling/emf/`. Version: Dezember 2012

[EPS73] Ehrig, Hartmut ; Pfender, Michael ; Schneider, Hans J.: Graph-Grammars: An Algebraic Approach. In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT'73)*, IEEE Computer Society, 1973, S. 167–180

[EPT04] Ehrig, Hartmut ; Prange, Ulrike ; Taentzer, Gabriele: Fundamental Theory for Typed Attributed Graph Transformation. In: [EEPPR04], S. 161–177

[Eri04] Ericson, Christer: *Real-Time Collision Detection*. Morgan Kaufmann, 2004

[Erm06] Ermel, Claudia: *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*, Technische Universität Berlin, Fak. IV, Diss., 2006

[ERT99] Ermel, C. ; Rudolf, M. ; Taentzer, G.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools. In: Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999, Kapitel The AGG approach: language and environment, S. 551–603

[Exp12] Microsoft: *Expression Blend 4*. Webseite. `http://www.microsoft.com/expression/products/Blend_Overview.aspx`. Version: 2012

[FB05] Fondement, Frédéric ; Baar, Thomas: Making Metamodels Aware of Concrete Syntax. In: Hartman, Alan (Hrsg.) ; Kreische, David (Hrsg.): *ECMDA-FA* Bd. 3748, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 190–204

[Fla04] Flake, Stephan: Towards the Completion of the Formal Semantics of OCL 2.0. In: Estivill-Castro, Vladimir (Hrsg.): *Proceedings of the 27th Conference on Australasian Computer Science (ACSC 2004)* Bd. 26, Australian Computer Society, 2004 (CRPIT), S. 73–82

[Fla12] Adobe Systems: *Adobe Flash Professional CS6*. Webseite. `http://www.adobe.com/de/products/flash`. Version: 2012

[FMRS07] Fuss, Christian ; Mosler, Christof ; Ranger, Ulrike ; Schultchen, Erhard: The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES. In: [EG07]

[FNTZ00] Fischer, Thorsten ; Niere, Jörg ; Torunski, Lars ; Zündorf, Albert: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: [EEKR00], S. 296–309

[FS07] Fecher, Harald ; Schönborn, Jens: UML 2.0 State Machines: Complete Formal Semantics Via core state machine. In: Brim, Lubos (Hrsg.) ; Haverkort, Boudewijn R. (Hrsg.) ; Leucker, Martin (Hrsg.) ; Pol, Jaco van d. (Hrsg.): *Proceedings of the 11th International Workshop on Formal Methods: Applications and Technology, 11th International Workshop (FMICS'06) and the 5th International Workshop (PDMC'06)* Bd. 4346, Springer-Verlag, 2007 (Lecture Notes in Computer Science), S. 244–260

[FSKR05] Fecher, Harald ; Schönborn, Jens ; Kyas, Marcel ; Roever, Willem P.: 29 New Unclarities in the Semantics of UML 2.0 State Machines. In: Lau, Kung-Kiu (Hrsg.) ; Banach, Richard (Hrsg.): *Proceedings of the 7th International Conference on Formal Engineering Method (ICFEM'05* Bd. 3785, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 52–65

[FUM11] Object Management Group: *Semantics of a Foundational Subset for Executable UML Models (fUML), 1.0*. `http://www.omg.org/spec/FUML/1.0/`. Version: Februar 2011

[GBEE11] Golas, Ulrike ; Biermann, Enrico ; Ehrig, Hartmut ; Ermel, Claudia: A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation. In: Echahed, Rachid (Hrsg.) ; Habel, Annegret (Hrsg.) ; Mosbah, Mohamed (Hrsg.): *Proceedings of the 3rd International Workshop on Graph Computation Models (GCM'2010)* Bd. 39, 2011 (Electronic Communications of the EASST), S. 111–126

[GBG+06] Geiss, Rubino ; Batz, Gernot V. ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, Andrea (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Montanari, Ugo (Hrsg.) ; Ribeiro, Leila (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Proceedings of the 3rd International Conference on Graph Transformation (ICGT'06)* Bd. 4178, Springer-Verlag, 2006 (Lecture Notes in Computer Science), S. 383–397

[Ges08] Gessenharter, Dominik: Mapping the UML2 Semantics of Associations to a Java Code Generation Model. In: Czarnecki, Krzysztof (Hrsg.) ; Ober, Ileana (Hrsg.) ; Bruel, Jean-Michel (Hrsg.) ; Uhl, Axel (Hrsg.) ; Völter, Markus (Hrsg.): *MoDELS* Bd. 5301, Springer-Verlag, 2008 (Lecture Notes in Computer Science), S. 813–827

[GHJJ96] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; John, Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 3. Auflage. Addison-Wesley, 1996

[GHV02] Gyapay, Szilvia ; Heckel, Reiko ; Varró, Dániel: Graph Transformation with Time: Causality and Logical Clocks. In: [CEKR02], S. 120–134

[Gil07] Gilbert, Nigel: *Agent-Based Models*. Sage Publications, 2007 (Quantitative Applications in the Social Sciences)

[GK01] Gogolla, Martin (Hrsg.) ; Kobryn, Cris (Hrsg.): *Proceedings of the 4th International Conference on The Unified Modeling Language: Modeling Languages, Concepts, and Tools*. Bd. *2185*. Springer-Verlag, 2001 (Lecture Notes in Computer Science)

[GL07] Guerra, Esther ; Lara, Juan de: Adding Recursion to Graph Transformation. In: [EG07]

[GM98] Geiger, Christian ; Müller, Wolfgang: Visuelle Spezifikation, Modellierung und Animation im Systementwurf. In: Lorenz, Peter (Hrsg.) ; Preim, Bernhard (Hrsg.): *Simulation und Visualisierung 1998 (SimVis'98)*, SCS Publishing House e.V., 1998, S. 206–220

[GME05] *GME 5 User's Manual – Version 5.0*. Webseite. `http://w3.isis.vanderbilt.edu/Projects/gme/`. Version: 2005

[GMR98] Geiger, Christian ; Müller, Wolfgang ; Rosenbach, Waldemar: SAM – An Animated 3D Programming Language. In: *P*roceedings of the 1998 IEEE Symposium on Visual Languages (VL'98)[IEE98], S. 228–235

[GPP98] Gogolla, Martin ; Parisi-Presicce, Francesco: State Diagrams in UML: A Formal Semantics using Graph Transformations. In: Rumpe, Bernhard (Hrsg.) ; Broy, Manfred (Hrsg.) ; Coleman, Derek (Hrsg.) ; Maibaum, Tom S. (Hrsg.): *Proceedings of the ICSE'98 Workshop on Precise Semantics for Modeling Techniques (PSMT'98)*, Technische Universität München, 1998, S. 55–72

[Gra98] Grant, Calum A. M.: Visual Language Editing Using a Grammar-Based Visual Structure Editor. In: *Journal of Visual Languages & Computing* 9 (1998), S. 351–374

[GS03] Greenfield, Jack ; Short, Keith: Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In: Crocker, Ron (Hrsg.) ; Jr., Guy L. S. (Hrsg.): *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*, ACM, 2003, S. 16–27

[Har87] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), S. 231–274

[Hec06] Heckel, Reiko: Graph Transformation in a Nutshell. In: *Electronic Notes in Theoretical Computer Science* 148 (2006), Nr. 1, S. 187–198

[Hen07] Henning, Peter A.: *Taschenbuch Multimedia*. Hanser Fachbuchverlag, 2007

[Het02] Hetland, Magnus L.: *Practical Python*. Apress, 2002 (Books for professionals by professionals)

[HG96] Harel, David ; Gery, Eran: Executable Object Modeling with Statecharts. In: Rombach, H. D. (Hrsg.) ; Maibaum, T. S. E. (Hrsg.) ; Zelkowitz, Marvin V. (Hrsg.): *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*, IEEE Computer Society, 1996, S. 246–257

[HHT96] Habel, Annegret ; Heckel, Reiko ; Taentzer, Gabriele: Graph Grammars with Negative Application Conditions. In: *Fundamenta Informativae* 26 (1996), Juni, S. 287–313

[HK04] Harel, David ; Kugler, Hillel: The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In: Ehrig, Hartmut (Hrsg.) ; Damm, Werner (Hrsg.) ; Desel, Jörg (Hrsg.) ; Grosse-Rhode, Martin (Hrsg.) ; Reif, Wolfgang (Hrsg.) ; Schnieder, Eckehard (Hrsg.) ; Westkämper, Engelbert (Hrsg.): *Integration of Software Specification Techniques for Applications in Engineering* Bd. 3147, Springer-Verlag, 2004 (Lecture Notes in Computer Science), S. 325–354

[HKT02] Heckel, Reiko ; Küster, Jochen M. ; Taentzer, Gabriele: Confluence of Typed Attributed Graph Transformation Systems. In: [CEKR02], S. 161–176

[HMTW95] Heckel, Reiko ; Müller, Jürgen ; Taentzer, Gabriele ; Wagner, Annika: Attributed Graph Transformations with Controlled Application of Rules. In: Valiente, G. (Hrsg.) ; Rossello Llompart, F. (Hrsg.) ; Technical Report B – 19, Universitat de les Illes Balears (Veranst.): *Proceedings of the Colloquium on Graph*

Transformation and its Application in Computer Science Technical Report B – 19, Universitat de les Illes Balears, Universitat de les Illes Balears, 1995, S. 41–53

[HMU00] Hopcroft, John E. ; Motwani, Rajeev ; Ullman, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation.* 2nd edition. Addison-Wesley, 2000

[HN96] Harel, David ; Naamad, Amnon: The STATEMATE semantics of statecharts. In: *ACM Transactions on Software Engineering and Methodology* 5 (1996), Oktober, Nr. 4, S. 293–333

[HP03] Harada, Yasunori ; Potter, Richard: Fuzzy Rewriting – Soft Program Semantics for Children. In: *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC'03)*, IEEE Computer Society, 2003, S. 39–46

[HS73] Herrlich, Horst ; Strecker, George E.: *Category Theory – An Introduction.* Boston : Allyn & Bacon, 1973

[Hun02] Hundhausen, Christopher D.: A Meta-Study of Algorithm Visualization Effectiveness. In: *Journal of Visual Languages & Computing* 13 (2002), Juni, Nr. 3, S. 259–290

[IEE97] *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL'97).* IEEE Computer Society, 1997

[IEE98] *Proceedings of the 1998 IEEE Symposium on Visual Languages (VL'98).* IEEE Computer Society, 1998

[ISO96] *ISO/EIC 14977 : 1996(E) – Extended Backus-Naur Form (EBNF).* 1996

[Jav12] *JavaFX.* Webseite. `http://docs.oracle.com/javafx/`. Version: Februar 2012

[JH01] Jamal, Rahman ; Hagestedt, Andre: *LabVIEW: Das Grundlagenbuch.* Addison-Wesley, 2001 (Scientific Computing)

[Jun00] Jung, Matthias: *Ein Generator zur Entwicklung visueller Sprachen*, Universität Paderborn, Fachbereich Mathematik-Informatik, Diss., 2000

[Kah96] Kahn, Kenneth M.: ToonTalk – An Animated Programming Environment for Children. In: *Journal of Visual Languages & Computing* 7 (1996), April, S. 197–217

[Kah98] Kahn, Kenneth M.: Helping Children to Learn Hard Things: Computer Programming with Familiar Objects and Actions. In: Druin, Allison (Hrsg.): *The Design of Children's Technology.* Morgan Kaufmann, 1998

[Kah04] Kahn, Kenneth M.: ToonTalk – Steps Towards Ideal Computer-Based Learning Environments. In: Tokoro, Mario (Hrsg.) ; Steels, Luc (Hrsg.): *A Learning Zone of One's Own: Sharing Representations and Flow in Collaborative Learning Environments*. IOS Press, 2004, S. 253–270

[Kai09] Kaiser, Richard: *C++mit Microsoft Visual C++2008: Einführung in Standard-C++, C++/CLI und die objektorientierte Windows .NET Programmierung*. Springer-Verlag, 2009

[Kay05] Kay, Alan: Squeak Etoys, Children & Learning. In: *Viewpoints Research Institute, VPRI Research Note RN-2005-001* (2005). `http://www.vpri.org/pdf/rn2005001_learning.pdf`

[KC09] Kerkouche, Elhillali ; Chaoui., Allaoua: A Graphical Tool Support to Process and Simulate ECATNets Models Based on Meta-Modelling and Graph Grammars. In: *INFOCOMP Journal of Computer Science* 8 (2009), Nr. 4, S. 37–44

[KDV07] Kienzle, Jörg ; Denault, Alexandre ; Vangheluwe, Hans: Model-Based Design of Computer-Controlled Game Character Behavior. In: Engels, Gregor (Hrsg.) ; Opdyke, Bill (Hrsg.) ; Schmidt, Douglas C. (Hrsg.) ; Weil, Frank (Hrsg.): *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)* Bd. 4735, Springer-Verlag, 2007 (Lecture Notes in Computer Science), S. 650–665

[Ken97] Kent, Stuart: Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In: Mary E. S. Loomis, A. Michael B. Toby Bloom B. Toby Bloom (Hrsg.): *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, ACM, 1997, S. 327–341

[KGKK02] Kuske, Sabine ; Gogolla, Martin ; Kollmann, Ralf ; Kreowski, Hans-Jörg: An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In: Butler, Michael J. (Hrsg.) ; Petre, Luigia (Hrsg.) ; Sere, Kaisa (Hrsg.): *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'02)* Bd. 2335, Springer-Verlag, 2002 (Lecture Notes in Computer Science), S. 11–28

[KHF06] Klügl, Franziska ; Herrler, Rainer ; Fehler, Manuel: SeSAm: Implementation of Agent-based Simulation using Visual Programming. In: Nakashima, Hideyuki (Hrsg.) ; Wellman, Michael P. (Hrsg.) ; Weiss, Gerhard (Hrsg.) ; Stone, Peter (Hrsg.): *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, ACM, 2006, S. 1439–1440

[KK93] Keller, Peter R. ; Keller, Mary M.: *Visual cues: practical data visualization.* IEEE Computer Society, 1993

[KM02] Knapp, Alexander ; Merz, Stephan: Model checking and code generation for UML state machines and collaborations. In: Reif, W. (Hrsg.) ; Schellhorn, G. (Hrsg.): *In G. Schellhorn and W. Reif. 5 th Workshop on Tools for System Design and Verification (FM-TOOLS*, 2002 (Ulmer Informatik-Berichte 2000-07), S. 59–64

[KM07] Kindborg, Mikael ; McGee, Kevin: Visual Programming with Analogical Representations: Inspirations from a Semiotic Analysis of Comics. In: *Journal of Visual Languages & Computing* 18 (2007), April, Nr. 2, S. 99–125

[Knu73] Knuth, Donald E.: *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition).* Addison-Wesley, 1973

[KS90a] Kahn, Kenneth M. ; Saraswat, Vijay A.: Actors as a Special Case of Concurrent Constraint Programming. In: Archibald, Jerry L. (Hrsg.) ; Yakemovic, K. C. B. (Hrsg.): *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP'90).* New York, NY, USA : ACM, 1990, S. 57–66

[KS90b] Kahn, Kenneth M. ; Saraswat, Vijay A.: Complete Visualizations of Concurrent Programs and their Executions. In: *Proceedings of the 1990 IEEE Symposium on Visual Languages (VL'90)*, IEEE Computer Society, 1990, S. 7–15

[KS02] Kastens, Uwe ; Schmidt, Carsten: VL-Eli: A Generator for Visual Languages – System Demonstration. In: *Electronic Notes in Theoretical Computer Science* 65 (2002), Nr. 3, S. 139–143

[KST01] Kehoe, Colleen ; Stasko, John T. ; Taylor, Ashley: Rethinking the Evaluation of Algorithm Animations as Learning Aids: an Observational Study. In: *International Journal of Human-Computer Studies* 54 (2001), Februar, Nr. 2, S. 265–284

[Kus01] Kuske, Sabine: A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In: [GK01], S. 241–256

[KZDX09] Kong, Jun ; Zhang, Kang ; Dong, Jing ; Xu, Dianxiang: Specifying behavioral semantics of UML diagrams through graph transformations. In: *Journal of Systems and Software* 82 (2009), Nr. 2, S. 292–306

[Lab12] National Instruments: *LabVIEW System Design Software, Version 2012.* Webseite. `http://www.ni.com/labview/`. Version: Mai 2012

[Lak86] Lakin, Fred: Spatial Parsing for Visual Languages. In: Chang, S. K. (Hrsg.) ; Ichikawa, T. (Hrsg.) ; Ligomenides, P. A. (Hrsg.): *Visual Languages*. Plenum Press, 1986, S. 35–85

[LBE+07] Lara, Juan de ; Bardohl, Roswitha ; Ehrig, Hartmut ; Ehrig, Karsten ; Prange, Ulrike ; Taentzer, Gabriele: Attributed graph transformation with node type inheritance. In: *Theoretical Computer Science* 376 (2007), Nr. 3, S. 139–163

[LEO05] Lambers, L. ; Ehrig, H. ; Orejas, F.: Efficient detection of conflicts in graph-based model transformation. In: *Electronic Notes in Theoretical Computer Science* 152 (2005), S. 97–109

[LETE04] Lara, Juan de ; Ermel, Claudia ; Taentzer, Gabriele ; Ehrig, Karsten: Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets. In: *Electronic Notes in Theoretical Computer Science* 109 (2004), S. 17–29

[LKW93] Löwe, Michael ; Korff, Martin ; Wagner, Annika: An Algebraic Framework for the Transformation of Attributed Graphs. In: Sleep, M. R. (Hrsg.) ; Plasmeijer, M. J. (Hrsg.) ; Eekelen, M. C. J. D. (Hrsg.) ; Sleep, Ronan (Hrsg.) ; Plasmeijer, Marinus (Hrsg.) ; Eekelen, Marko van (Hrsg.): *Term Graph Rewriting: Theory and Practice*. John Wiley & Sons Ltd, 1993, Kapitel 14, S. 185–199

[LLMC05] Levendovszky, Tihamer ; Lengyel, László ; Mezei, Gergely ; Charaf, Hassan: A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. In: *Electronic Notes in Theoretical Computer Science* 127 (2005), Nr. 1, S. 65–75

[LMB+01] Ledeczi, Akos ; Maroti, Miklos ; Bakay, Arpad ; Karsai, Gabor ; Garrett, Jason ; Thomasson, Chuck ; Nordstrom, Greg ; Sprinkle, Jonathan ; Volgyesi, Peter: The Generic Modeling Environment. In: *Proceedings of the IEEE International Workshop on Intelligent Signal Processing (WISP'01)*, IEEE Computer Society, 2001

[LMM99] Latella, Diego ; Majzik, István ; Massink, Mieke: Towards a Formal Operational Semantics of UML Statechart Diagrams. In: Ciancarini, Paolo (Hrsg.) ; Fantechi, Alessandro (Hrsg.) ; Gorrieri, Roberto (Hrsg.): *Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)* Bd. 139, Kluwer Academic Publishers, 1999 (IFIP Conference Proceedings)

[LRV94] Langmaack, Hans (Hrsg.) ; Roever, Willem P. (Hrsg.) ; Vytopil, Jan (Hrsg.): *Proceedings of the 3rd International Symposium*

on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94). Bd. *863*. Springer-Verlag, 1994 (Lecture Notes in Computer Science)

[LSP08] Langelier, Guillaume ; Sahraoui, Houari A. ; Poulin, Pierre: Exploring the Evolution of Software Quality with Animated Visualization. In: Bottoni, Paolo (Hrsg.) ; Rosson, Mary B. (Hrsg.) ; Minas, Mark (Hrsg.): *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'08)*, IEEE Computer Society, 2008, S. 13–20

[LV02] Lara, Juan de ; Vangheluwe, Hans: AToM^3: A Tool for Multi-formalism and Meta-modelling. In: Kutsche, Ralf-Detlef (Hrsg.) ; Weber, Herbert (Hrsg.): *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE'02)*, Springer-Verlag, 2002, S. 174–188

[LVA04] Lara, Juan de ; Vangheluwe, Hans ; Alfonseca, Manuel: Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. In: *Software and System Modeling* 3 (2004), Nr. 3, S. 194–209

[LW94] Liskov, Barbara H. ; Wing, Jeannette M.: A Behavioral Notion of Subtyping. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), November, Nr. 6, S. 1811–1841

[Mag12] No Magic: *Magic Draw, Version 17.0.2*. Webseite. `http://www.nomagic.com/products/magicdraw.html`. Version: Juli 2012

[Mai12] Maier, Sonja: *A Pattern-based Approach for the Combination of Different Layout Algorithms in Diagram Editors*, Universität der Bundeswehr München, Fakultät für Informatik, Diss., 2012

[Mar82] Martin, George E.: *Transformation Geometry: An Introduction to Symmetry*. Springer-Verlag, 1982

[Maz10] Mazanek, Steffen: *Exploiting hypergraph grammars for the realization of syntax-based user assistance in diagram editors*, Universität der Bundeswehr München, Fakultät für Informatik, Diss., 2010

[MB02] Mellor, Stephen J. ; Balcer, Marc: *Executable Uml: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002

[MG97] Minas, Mark ; Gottschall, Johann: Specifying Animated Diagram Languages. In: *Proceedings of the International Workshop on Theory of Visual Languages (TVL'97)*, 1997

[Min98] Minas, Mark: Automatically Generating Environments for Dynamic Diagram Languages. In: *P*roceedings of the 1998 IEEE Symposium on Visual Languages (VL'98)[IEE98], S. 70–71

[Min00] Minas, Mark: Hypergraphs as a Uniform Diagram Representation Model. In: [EEKR00], S. 281–295

[Min01] Minas, Mark: *Spezifikation und Generierung graphischer Diagrammeditoren*. Aachen : Shaker, 2001. – zugl. Habilitationsschrift Universität Erlangen-Nürnberg, 2000

[Min02] Minas, Mark: Concepts and realization of a diagram editor generator based on hypergraph transformation. In: *Science of Computer Programming* 44 (2002), August, Nr. 2, S. 157–180

[Min04] Minas, Mark: Visual specification of visual editors with DiaGen. In: Pfaltz, John L. (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Böhlen, Boris (Hrsg.): *Proceedigs of the 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)* Bd. 3062, Springer-Verlag, 2004 (Lecture Notes in Computer Science), S. 473–478

[Min06a] Minas, Mark: Generating Meta-Model-Based Freehand Editors. In: [ZV06]

[Min06b] Minas, Mark: Syntax Analysis for Diagram Editors: A Constraint Satisfaction Problem. In: Celentano, Augusto (Hrsg.) ; Mussio, Piero (Hrsg.): *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI'2006)* Bd. 3748, 2006 (ACM), S. 167–170

[MK08] Morgado, Leonel ; Kahn, Ken: Towards a specification of the ToonTalk language. In: *Journal of Visual Languages & Computing* 19 (2008), Nr. 5, S. 574–597

[MM96] Marriott, Kim ; Meyer, Bernd: Towards a Hierarchy of Visual Languages. In: *Proceedings of the 1996 IEEE Symposium on Visual Languages (VL'96)*, IEEE Computer Society, 1996, S. 196–203

[MM03] Miller, Joaquin ; Mukerji, Jishnu: MDA Guide Version 1.0.1 / Object Management Group (OMG). Version: Juni 2003. `http://www.omg.org/mda/`. 2003. – Forschungsbericht

[MM10] Maier, Sonja ; Minas, Mark: Combination of Different Layout Approaches. In: Bottoni, Paolo (Hrsg.) ; Guerra, Esther (Hrsg.) ; Lara, Juan de (Hrsg.): *Proceedings of the 2nd International Workshop Visual Formalisms for Patterns (VFfP'2010)* Bd. 31, 2010 (Electronic Communications of the EASST)

[MMC09] Mészáros, Tamás ; Mezei, Gergely ; Charaf, Hassan: Engineering the Dynamic Behavior of Metamodeled Languages. In: *Simulation* 85 (2009), Nr. 11, S. 793–810

[MMM08] Maier, Sonja ; Mazanek, Steffen ; Minas, Mark: Layout Specification on the Concrete and Abstract Syntax Level of a Diagram Language. In: Fish, Andrew (Hrsg.) ; Störrle, Harald (Hrsg.): *Proceedings of the 2nd International Workshop on Layout of (Software) Engineering Diagrams (LED'2008)* Bd. 13, 2008 (Electronic Communications of the EASST)

[MOF08] Object Management Group: *MOF Model to Text Transformation Language, 1.0*. `http://www.omg.org/spec/MOFM2T/1.0/`. Version: Januar 2008

[MOF11] Object Management Group: *Meta Object Facility (MOF) Core Specification, 2.4.1*. `http://www.omg.org/spec/MOF/2.4.1/`. Version: August 2011

[Mor08] Moroney, Lawrence: Erstellen von Animationen mit XAML und Expression Blend. In: *MSDN Magazin* (2008), August

[MPB12] Mirlacher, Thomas ; Palanque, Philippe ; Bernhaupt, Regina: Engineering Animations in User Interfaces. In: *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'12)*. New York, NY, USA : ACM, 2012, S. 111–120

[MRR98] Magalhães, Léo Pini ; Raposo, Alberto B. ; Ricarte, Ivan Luiz M.: Animation modeling with petri nets. In: *Computers & Graphics* 22 (1998), Nr. 6, S. 735–743

[MS94] Mukherjea, Sougata ; Stasko, John T.: Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger. In: *ACM Transactions on Computer-Human Interaction* 1 (1994), September, Nr. 3, S. 215–244

[MSP96] Maggiolo-Schettini, Andrea ; Peron, Adriano: A graph rewriting framework for Statecharts semantics. In: Cuny, Janice (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph Grammars and Their Application to Computer Science* Bd. 1073. Springer-Verlag, 1996, S. 107–121

[Mye90] Myers, Brad A.: Taxonomies of Visual Programming and Program Visualization. In: *Journal of Visual Languages & Computing* 1 (1990), März, Nr. 1, S. 97–123

[NBD+05] Nierstrasz, Oscar ; Bergel, Alexandre ; Denker, Marcus ; Ducasse, Stéphane ; Gälli, Markus ; Wuyts, Roel: On the Revival of Dynamic Languages. In: Gschwind, Thomas (Hrsg.) ; Assmann, Uwe (Hrsg.) ; Nierstrasz, Oscar (Hrsg.): *Proceedings of the 4th International Workshop on Software Composition (SC'2005)* Bd. 3628, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 1–13

[NRA+02] Naps, Thomas ; Rössling, Guido ; Almstrum, Vicki ; Dann, Wanda ; Fleischer, Rudolf ; Hundhausen, Christopher ; Korhonen, Ari ; Malmi, Lauri ; McNally, Myles ; Rodger, Susan ; Velázquez-Iturbide, J. Ángel: Exploring the Role of Visualization and Engagement in Computer Science Education. In: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'02)*, ACM, 2002, S. 131–152

[NS73] Nassi, Isaac ; Shneiderman, Ben: Flowchart techniques for structured programming. In: *ACM SIGPLAN Notices* 8 (1973), August, Nr. 8, S. 12–26

[NT03] Niaz, Iftikhar A. ; Tanaka, Jiro: Code Generation From UML Statecharts. In: Hamza, M.H. (Hrsg.): *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications*, IASTED/ACTA Press, 2003, S. 315–321

[NTCO07] North, M. J. ; Tatara, E. ; Collier, N. T. ; Ozik, J.: Visual Agent-based Model Development with Repast Simphony. In: *Proceedings of the Agent 2007 Conference on Complex Interaction and Social Emergence*, 2007

[OCL12] Object Management Group: *Object Constraint Language, 2.3.1.* `http://www.omg.org/spec/OCL/2.3.1/`. Version: Januar 2012

[OO99] O'Loughlin, John ; O'Sullivan, Carol: Real-time Animation of Objects Modelled using Constructive Solid Geometry. In: Zára, Jirí (Hrsg.): *Proceedings of the 15th Spring Conference on Computer Graphics (SCCG'99)*, 1999, S. 67–73

[Pan02] Pane, John F.: *A Programming System for Children that is Designed for Usability*, Carnegie Mellon University, Computer Science Department, Diss., Mai 2002

[Pat94] Pattis, Richard E.: *Karel the Robot: A Gentle Introduction to the Art of Programming.* 2nd edition. New York, NY, USA : John Wiley & Sons, Inc., 1994

[PFG+06] Pantel, Marc ; Farail, Patrick ; Gaufillet, Pierre ; Canals, Agusti ; Le Camus, Christophe ; Sciamma, David ; Michel, Pierre ; Crégut, Xavier: The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEms Design. In: *Proceedings of the 3rd European Congress on Embedded Real Time Software (ERTS'06)*, 2006

[PK07] Peppler, Kylie A. ; Kafai, Yasmin B.: From SuperGoo to Scratch: Exploring creative digital media production ininformal learning. In: *Learning, Media and Technology Special Issue: Media Education*

Goes Digital 32(2) (2007), 149–166. http://site.educ.indiana.edu/Portals/206/pdfs/SupergooToScratch_LMT2007.pdf

[Ple09] Pleuss, Andreas: *Model-Driven Development of Interactive Multimedia Applications – Towards Better Integration of Software Engineering and Creative Design*, Ludwig-Maximilians-Universität München, Fakultät für Mathematik, Informatik und Statistik, Diss., 2009

[PR08] Pedro, Luis ; Risoldi, Matteo: Metamodeling with Eclipse / Université de Genève. Version: Mai 2008. http://smv.unige.ch/research-projects/mtv/files/MetaModelingWithEclipse.pdf. 2008. – Forschungsbericht

[PS04] Plump, Detlef ; Steinert, Sandra: Towards Graph Programs for Graph Algorithms. In: [EEPPR04], S. 128–143

[QVT11] Object Management Group: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 1.1*. http://www.omg.org/spec/QVT/1.1/. Version: Januar 2011

[RAK06] Rössling, Guido ; Ackermann, Tobias ; Kulessa, Simon: Visualisierung von Algorithmen und Datenstrukturen. In: Mühlhäuser, Max (Hrsg.) ; Rössling, Guido (Hrsg.) ; Steinmetz, Ralf (Hrsg.): *DeLFI 2006, 4. e-Learning Fachtagung Informatik* Bd. 87, Gesellschaft für Informatik e.V., 2006 (Lecture Notes in Informatics), S. 231–242

[RB88] Rydeheard, David E. ; Burstall, Rod M.: *Computational category theory*. Prentice Hall, 1988

[RDE+08] Rensink, Arend ; Dotor, Alexander ; Ermel, Claudia ; Jurack, Stefan ; Kniemeyer, Ole ; Lara, Juan de ; Maier, Sonja ; Staijen, Tom ; Zündorf, Albert: Ludo: A Case Study for Graph Transformation Tools. In: [SNZ08], S. 493–513

[RDV09] Rivera, José E. ; Durán, Francisco ; Vallecillo, Antonio: A graphical approach for modeling time-dependent behavior of DSLs. In: [CFH09], S. 51–55

[Rei85] Reisig, Wolfgang: *Petri nets: an introduction*. New York, NY, USA : Springer-Verlag, 1985

[Rep00] Repenning, Alexander: AgentSheets®: an Interactive Simulation Environment with End-User Programmable Agents. In: *Interaction 2000*, 2000

[RG08] Rensink, Arend (Hrsg.) ; Gorp, Pieter V. (Hrsg.): *Proceedings of the 4th International Workshop on Graph-Based Tools: The Contest*. 2008

[RI06] Repenning, Alexander ; Ioannidou, Andri: AgentCubes: Raising the Ceiling of End-User Development in Education through Incremental 3D. In: Grundy, John (Hrsg.) ; Howse, John (Hrsg.): *Proceedings of the 2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'06)*, IEEE Computer Society, 2006, S. 27–34

[Rie96] Rieber, Llyod P.: Animation as Feedback in a Computer-Based Simulation: Representation Matters. In: *Educational Technology Research and Development* 44 (1996), S. 5–22

[RMMH+09] Resnick, Mitchel ; Maloney, John ; Monroy-Hernández, Andrés ; Rusk, Natalie ; Eastmond, Evelyn ; Brennan, Karen ; Millner, Amon ; Rosenbaum, Eric ; Silver, Jay ; Silverman, Brian ; Kafai, Yasmin: Scratch: Programming for All. In: *Communications of the ACM* 52 (2009), November, Nr. 11, S. 60–67

[Rob03] Robbins, Jennifer N.: Chapter 17: Animated GIFs. In: *Learning Web Design: A Beginner's Guide to HTML, Graphics, and Beyond*, O'Reilly, 2003 (Learning Series)

[Rod02] Rodger, Susan H.: Introducing Computer Science Through Animation and Virtual Worlds. In: Gersting, Judith L. (Hrsg.) ; Walker, Henry M. (Hrsg.) ; Grissom, Scott (Hrsg.): *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*, ACM, 2002, S. 186–190

[Roz97] Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997

[RQZ07] Rupp, Chris ; Queins, Stefan ; Zengler, Barbara: *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. 3. Auflage. Hanser Fachbuchverlag, 2007

[RS95] Repenning, Alexander ; Sumner, Tamara: Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. In: *Computer* 28 (1995), Nr. 3, S. 17–25

[RV06] Ráth, István ; Varró, Dániel: Challenges for Advanced Domain-Specific Modeling Frameworks. In: *Proceedings of the 1st International Workshop on Domain Specific Program Development (DSPD'06)*, 2006

[RW99] Reggio, G. ; Wieringa, R. J.: Thirty one Problems in the Semantics of UML 1.3 Dynamics. In: *In Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'99) – Workshop on Rigorous Modeling and Analysis of the UML: Challenges and Limitations*, 1999

[Sak12] Sakellariou, Ilias: Agent based Modelling and Simulation using State Machines. In: Pina, Nuno (Hrsg.) ; Kacprzyk, Janusz (Hrsg.) ; Obaidat, Mohammad S. (Hrsg.): *Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies (SIMULTECH'12)*, SciTePress, 2012, S. 270–279

[SBL93] Stasko, John T. ; Badre, Albert ; Lewis, Clayton: Do Algorithm Animations Assist Learning?: an Empirical Study and Analysis. In: Ashlund, Stacey (Hrsg.) ; Mullet, Kevin (Hrsg.) ; Henderson, Austin (Hrsg.) ; Hollnagel, Erik (Hrsg.) ; White, Ted N. (Hrsg.): *Proceedings of the International Conference on Human-Computer Interaction (INTERACT'93)*, ACM, 1993, S. 61–66

[Sch96] Schürr, Andy: PROGRES for Beginners / Lehrstuhl für Informatik III, RWTH Aachen, Germany. 1996. – Forschungsbericht

[Sch98] Schiffer, Stefan: *Visuelle Programmierung. Grundlagen und Einsatzmöglichkeiten.* Addison-Wesley, 1998

[Sch06] Schmidt, Carsten: *Generierung von Struktureditoren für anspruchsvolle visuelle Sprachen*, Universität Paderborn, Diss., 2006

[SCT00] Smith, David C. ; Cypher, Allen ; Tesler, Larry: Programming by Example: Novice Programming Comes of Age. In: *Communications of the ACM* 43 (2000), März, Nr. 3, S. 75–81

[SE99] Sauer, Stefan ; Engels, Gregor: OMMMA: An Object-Oriented Approach for Modeling Multimedia Information Systems. In: Golubchik, Leana (Hrsg.) ; Tsotras, Vassilis J. (Hrsg.): *Proceedings of the 5th Workshop on Multimedia Information Systems (MIS'99)*, 1999, S. 64–71

[SE07] Sauer, Stefan ; Engels, Gregor: Easy Model-Driven Development of Multimedia User Interfaces with GuiBuilder. In: Stephanidis, Constantine (Hrsg.): *Proceedings of the 4th International Conference on Universal Access in Human-Computer Interaction (UAHCI'07)* Bd. 4554, Springer-Verlag, 2007 (Lecture Notes in Computer Science), S. 537–546

[SG99] Simons, Anthony J H. ; Graham, Ian: 30 Things that Go Wrong in Object Modeling with UML 1.3. In: *Behavioral Specifications of Businesses and Systems*, Kluwer Academic Publishers, 1999, S. 237–257

[Sha01] Shaw, Mary M.: The Coming-of-Age of Software Architecture Research. In: Müller, Hausi A. (Hrsg.) ; Harrold, Mary J. (Hrsg.) ; Schäfer, Wilhelm (Hrsg.): *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, IEEE Computer Society, 2001, S. 656–664

[Shu88] Shu, N.C.: *Visual Programming*. Van Nostrand Reinhold, 1988

[Sin95] Singh, Ghan Bir: Single versus multiple inheritance in object oriented programming. In: *SIGPLAN OOPS Messenger* 6 (1995), Januar, Nr. 1, S. 30–39

[SM09] Strobl, Torsten ; Minas, Mark: Implementing an Animated Lambda-Calculus. In: [CFH09], S. 96–109

[SM10] Strobl, Torsten ; Minas, Mark: Specifying and Generating Editing Environments for Interactive Animated Visual Models. In: Küster, Jochen (Hrsg.) ; Tuosto, Emilio (Hrsg.): *Proceedings of the 9th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'10)* Bd. 29, 2010 (Electronic Communications of the EASST)

[SM12] Strobl, Torsten ; Minas, Mark: Generating Graph Transformation Rules from AML/GT State Machine Diagrams for Building Animated Model Editors. In: Schürr, Andy (Hrsg.) ; Varró, Dániel (Hrsg.) ; Varró, Gergely (Hrsg.): *Proceedigs of the 4th International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE'11)* Bd. 7233, Springer-Verlag, 2012 (Lecture Notes in Computer Science), S. 65–80

[Smi87] Smith, Randall B.: Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic. In: Carroll, John M. (Hrsg.) ; Tanner, Peter P. (Hrsg.): *Proceedings of Joint Conference on Human Factors in Computing Systems and Graphics Interface (CHI+GI'87)*, ACM, 1987, S. 61–67

[SMPV10] Strobl, Torsten ; Minas, Mark ; Pleuss, Andreas ; Vitzthum, Arnd: From the Behavior Model of an Animated Visual Language to its Editing Environment Based on Graph Transformation. In: Lara, Juan de (Hrsg.) ; Varró, Dániel (Hrsg.): *Proceedings of the 4th International Workshop on Graph-Based Tools (GraBaTs'10)* Bd. 32, 2010 (Electronic Communications of the EASST)

[SNZ08] Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proceedigs of the 3rd International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE'07)*. Bd. *5088*. Springer-Verlag, 2008 (Lecture Notes in Computer Science)

[Sta90] Stasko, John T.: Tango: A Framework and System for Algorithm Animation. In: *IEEE Computer* 23 (1990), September, Nr. 9, S. 27–39

[Sta05] *StarUML, Version 5*. Webseite. `http://staruml.sourceforge.net/`. Version: Dezember 2005

[Ste00] Steinmetz, R.: *Multimedia-Technologie: Grundlagen, Komponenten und Systeme*. 3. Auflage. Springer-Verlag, 2000

[Ste09] Steyer, Ralph: *Einstieg in JavaFX: Dynamische und interaktive Java-Applikationen mit JavaFX*. Addison-Wesley, 2009

[Sto10] Stolee, Kathryn T.: Kodu language and grammar specification / Microsoft Research. 2010. – Forschungsbericht

[Str00] Stroustrup, Bjarne: *Die C++-Programmiersprache*. 4. Auflage. Addison-Wesley, 2000

[SV07] Syriani, Eugene ; Vangheluwe, Hans: Programmed Graph Rewriting with DEVS. In: [SNZ08], S. 136–151

[SV08] Syriani, Eugene ; Vangheluwe, Hans: Programmed Graph Rewriting with Time for Simulation-Based Design. In: Vallecillo, Antonio (Hrsg.) ; Gray, Jeff (Hrsg.) ; Pierantonio, Alfonso (Hrsg.): *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT'08)* Bd. 5063, Springer-Verlag, 2008 (Lecture Notes in Computer Science), S. 91–106

[SVG11] World Wide Web Consortium: *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. `http://www.w3.org/TR/SVG11/`. Version: August 2011

[TEG+05] Taentzer, Gabriele ; Ehrig, Karsten ; Guerra, Esther ; Lara, Juan D. ; Levendovszky, Tihamer ; Prange, Ulrike ; Varro, Daniel ; Varro-Gyapay, Szilvia: Model Transformations by Graph Transformations: A Comparative Study. In: *Proceedings of the Workshop on Model Transformations in Practice at (co-located with MoDELS'05)*, 2005

[TK07] Tekušová, Tatiana ; Kohlhammer, Jörn: Applying Animation to the Visual Analysis of Financial Time-Dependent Data. In: Banissi, Ebad (Hrsg.): *Proceedings of the 11th International Conference on Information Visualization (IV'07)*, IEEE Computer Society, 2007, S. 101–108

[TK09] Tolvanen, Juha-Pekka ; Kelly, Steven: MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In: Arora, Shail (Hrsg.) ; Leavens, Gary T. (Hrsg.): *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*, ACM, 2009, S. 819–820

[TKH99] Thirunarayan, Krishnaprasad ; Kniesel, Günter ; Hampapuram, Haripriyan: Simulating multiple inheritance and generics in Java. In: *Computer Language* 25 (1999), Nr. 4, S. 189–210

[TMB02] Tversky, Barbara ; Morrison, Julie B. ; Bétrancourt, Mireille: Animation: can it facilitate? In: *International Journal of Human-Computer Studies* 57 (2002), Nr. 4, S. 247–262

[Tog08] Borland: *Together, Version 2008*. Webseite. `http://borland.com/products/Together/`. Version: 2008

[TRMR97] Tamiosso, Fabiana S. ; Raposo, Alberto B. ; Magalhães, Léo Pini ; Ricarte, Ivan Luiz M.: Building Interactive Animations using VRML and Java. In: *Proceedings of the 1997 Brazilien Symposium on Computer Graphics and Image Processing (SIBGRAPI'97)*, IEEE Computer Society, 1997, S. 42–48

[Tur12] *Turtle Art*. Webseite. `http://wiki.sugarlabs.org/go/Activities/TurtleArt`. Version: Mai 2012

[TVT07] Tiella, Roberto ; Villafiorita, Adolfo ; Tomasi, Silvia: FSMC+, a tool for the generation of Java code from statecharts. In: Amaral, Vasco (Hrsg.) ; Marcelino, Luis (Hrsg.) ; Veiga, Luís (Hrsg.) ; Cunningham, H. C. (Hrsg.): *PPPJ* Bd. 272, ACM, 2007 (ACM International Conference Proceeding Series), S. 93–102

[Ull11] Ullenboom, Christian: *Java ist auch eine Insel*. 10. Auflage. Galileo Computing, 2011 `http://openbook.galileocomputing.de/javainsel/`

[UML11] Object Management Group: *Unified Modeling Language, Superstructure, 2.4.1*. `http://www.omg.org/spec/UML/2.4.1/`. Version: August 2011

[UMo10] Altova: *UModel, Version 2010 Rel. 3*. Webseite. `http://www.altova.com/umodel.html`. Version: Mai 2010

[Usm09] Usman, Nadeem A. Muhammad: Automatic Generation of Java Code from UML Diagrams using UJECTOR. In: *International Journal of Software Engineering and Its Applications* 3 (2009), April, Nr. 2, S. 21–37

[Var02] Varró, Dániel: A Formal Semantics of UML Statecharts by Model Transition Systems. In: [CEKR02], S. 378–392

[Vic08] Victor, Bret: *Alligator Eggs! a puzzle game*. Webseite. `http://worrydream.com/AlligatorEggs/`. Version: 2008

[Vis12] *Visual Paradigm for UML, Version 10.0*. Webseite. `http://www.visual-paradigm.com/product/vpuml/`. Version: Juni 2012

[Vit05] Vitzthum, Arnd: SSIML/Behaviour: Designing Behaviour and Animation of Graphical Objects in Virtual Reality and Multimedia Applications. In: *Proceedings of the 7th IEEE International*

Symposium on Multimedia (ISM'05). Los Alamitos, CA, USA : IEEE Computer Society, 2005, S. 159–167

[Vod97] Vodislav, Dan: A Visual Programming Model for User Interface Animation. In: *P*roceedings of the 1997 IEEE Symposium on Visual Languages (VL'97)[IEE97], S. 344–351

[VP04] Varró, Dániel ; Pataricza, András: Generic and Meta-transformations for Model Transformation Engineering. In: Baar, Thomas (Hrsg.) ; Strohmeier, Alfred (Hrsg.) ; Moreira, Ana M. D. (Hrsg.) ; Mellor, Stephen J. (Hrsg.): *UML 2004 – The Unified Modelling Language, Modelling Languages and Applications, 7th International Conference* Bd. 3273, Springer-Verlag, 2004 (Lecture Notes in Computer Science), S. 290–304

[War94] Ward, Martin P.: Language-Oriented Programming. In: *Software – Concepts and Tools* 15 (1994), Nr. 4, S. 147–161

[WD11] Wilke, Claas ; Demuth, Birgit: UML is still inconsistent! How to improve OCL Constraints in the UML 2.3 Superstructure. In: *Electronic Communications of the EASST* 44 (2011)

[Wil12] Wilensky, Uri: *NetLogo User Manual – Version 5.0.3*. Webseite. `http://ccl.northwestern.edu/netlogo/docs/`. Version: Oktober 2012

[WTEK08] Winkelmann, Jessica ; Taentzer, Gabriele ; Ehrig, Karsten ; Küster, Jochen M.: Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. In: *Electronic Notes in Theoretical Computer Science* 211 (2008), S. 159–170

[XMI05] Object Management Group: *XML Metadata Interchange Specification, 2.0.1*. `http://www.omg.org/spec/XMI/`. Version: Juli 2005

[XML08] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. `http://www.w3.org/TR/xml/`. Version: November 2008

[Zei76] Zeigler, Bernard P.: *Theory of Modeling and Simulation*. John Wiley, 1976

[Zei84] Zeigler, Bernhard: *Multifacetted Modeling and Discrete Event Simulation*. Academic Press, 1984

[Zim09] Zimmermann, Frank: Entwicklung von graphischen Editoren am Beispiel von Petri-Netzen / Nordakademie. 2009 (Arbeitspapiere der Nordakademie 1). – Forschungsbericht

[ZV06] Zündorf, Albert (Hrsg.) ; Varró, Dániel (Hrsg.): *Proceedings of the 3rd International Workshop on Graph Based Tools (GraBaTs'06)*. Bd. *1*. 2006 (Electronic Communications of the EASST)

[ZZ98] Zhang, Da-Qian ; Zhang, Kang: VisPro: A Visual Language Generation Toolset. In: *P*roceedings of the 1998 IEEE Symposium on Visual Languages (VL'98)[IEE98], S. 195–202