Konzeption und Implementierung eines Generators für Simulationsumgebungen

Ingo Weinhold

Diplomarbeit

am Fachbereich Informatik der Technischen Universität Berlin,
betreut von Prof. Dr. H. Ehrig, Dr. G. Taentzer und Dr. R. Bardohl

2. Oktober 2001

Zusammenfassung

In dieser Arbeit wird die Entwicklung eines Generators für visuelle Simulationsumgebungen beschrieben, wobei auf einen gegebenen Alphabet- und Grammatikeditor aufgesetzt wird.

Eine visuelle Simulationsumgebung ist eine grafische Umgebung zum Editieren und Simulieren von Elementen einer visuellen Sprache. Visuelle Sprachen werden als eine Menge von visuellen Sätzen verstanden, die, ist ein visuelles Alphabet gegeben, durch eine visuelle Grammatik erzeugt werden können. Die Beschreibung eines Alphabets erfolgt mit den Mitteln der universellen Algebren und grafischer Constraints, eine Grammatik wird im weiteren Sinne als algebraische Graphgrammatik aufgefasst.

Das Editieren der visuellen Sätze kann sowohl syntaxgesteuert, also regelbasiert, als auch Freihand erfolgen, wobei zusätzlich ein Mittel zur Parsierung, ebenfalls eine Grammatik, benötigt wird. Für die Simulation muss eine Schrittsemantik definiert werden, deren Schritte auf Regelanwendungen basieren, die das Fundament für mächtigere Konstrukte, die Simulationsausdrücke, bilden.

Die selbständige und eigenhändige Anfertigung
dieser Diplomarbeit versichere ich an Eides statt.

Berlin, den 25. September 2001

Unterschrift

Mein Dank gilt ...

Roswitha Bardohl ...	für die großartige Betreuung,
Olga Runge ...	für die hilfreichen Tipps zu AGG,
Stephan Aßmus ...	für das Durchkämpfen durch meinen Text und das Verständnis dafür, dass der eine oder andere Kinoabend geopfert werden musste,
jenen ...	die einen Beitrag zu GenGEd geleistet haben oder noch leisten werden, sowie
allen ...	die meinen Dank verdient haben und die ich trotzdem vergaß hier namentlich zu erwähnen.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Visuelle Sprachen 1

1.2 Gliederung 3

2 Konzepte **5**

2.1 Alphabet 5

2.1.1 Grafische Primitiva und Datentypen 6

2.1.2 Constraints 6

2.1.3 Symbole und Links 8

2.1.4 Alphabet, Typgraph, Signatur 10

2.1.5 Visueller Satz, Algebra 11

2.2 Grammatik 12

2.2.1 Morphismen 12

2.2.2 Regeln 12

2.2.3 Anwendungsbedingungen 13

2.2.4 Grammatiken 14

2.3 Parsierung 16

2.3.1 Layerfunktion 17

2.3.2 Kritische Paare 17

2.4 Simulation 18

2.4.1 Simulationsausdruck 18

2.4.2 Simulationsschritt 22

2.5 VL-Spezifikation, VL-Umgebung 24

3 Ausgangssituation **25**

3.1 Java 25
3.2 Alphabeteditor 26
3.2.1 Funktion 26
3.2.2 Implementierung 27
3.3 Grammatik- und VL-Editor 32
3.3.1 Funktion 32
3.3.2 Implementierung 33
3.4 ParCon und ParCon-Daemon 35
3.5 Agg 36

4 Anforderungen **37**
4.1 Integration von Alphabet-, Grammatik- und VL-Editor 37
4.2 Überarbeitung von Grammatik- und VL-Editor 38
4.3 Überarbeitung der Schnittstelle zu Agg 38
4.4 XML-basiertes Speichern und Laden 38
4.5 VL-Spezifikation, VL-Spezifikationseditor 39
4.6 VL-Umgebung 39

5 Entwurf und Implementierung **41**
5.1 Hilfsmittel 42
5.2 Struktur des Systems 42
5.2.1 Verzeichnisstruktur 42
5.2.2 Packagestruktur 43
5.3 Grammatik 44
5.3.1 Signatur 44
5.3.2 Algebra 47
5.3.3 Grammatik 54
5.4 Transformation 57
5.4.1 Allgemeine Schnittstelle 57
5.4.2 Agg-Anbindung 59
5.5 Parsierung 64
5.5.1 Allgemeine Schnittstelle 64
5.5.2 Agg-Anbindung 68

5.6 Simulation 69
5.6.1 Allgemeine Schnittstelle 69
5.6.2 Simulationsausdrücke 72
5.7 VL-Spezifikation 73
5.8 Persistenz 74
5.8.1 XML 74
5.8.2 DTDs 75
5.8.3 Implementierung 84
5.9 Editoren 89
5.9.1 Grammatikeditor 90
5.9.2 VL-Editor 94
5.9.3 VL-Spezifikationseditor 95
5.9.4 Simulationsumgebung 98
5.9.5 VL-Umgebung 100

6 Spezifikation hierarchischer Statecharts 103
6.1 Einführung 103
6.2 Alphabet und Syntaxgrammatik 105
6.3 Parsierung 110
6.4 Simulation 116

7 Ergebnisse und Ausblick 123
7.1 Ergebnisse 123
7.2 Ausblick 124

Literaturverzeichnis 126

A Benutzerhandbuch 129
A.1 Installation und Start 129
A.1.1 `configure` 130
A.1.2 Kompilation 131
A.1.3 Programmstart 131
A.2 Alphabeteditor 132
A.3 Grammatikeditor 132

A.4 VL-Spezifikationseditor 134
A.4.1 Menüleiste 134
A.4.2 Syntax 135
A.4.3 Parsierung 136
A.4.4 Simulation 137
A.4.5 Allgemeines 139
A.5 VL-Umgebung 140
A.5.1 Menüleiste 140
A.5.2 Editor 141
A.5.3 Simulation 142

Kapitel 1

Einleitung

1.1 Visuelle Sprachen

Für den Begriff der visuellen Sprache (visual language, VL) gibt es bisher keine umfassende und allgemein anerkannte Definition. Schon der allgemeinere Begriff der Sprache stellt sich als nicht leicht greifbar dar. Im Brockhaus ([Gmb86]) findet sich folgende Erklärung:

> Sprache [ahd. sprahha »Sprache«; »Aussage«, »Rede«, »Beratung«], im weitesten Sinne von Semiotik und Informationstheorie ein konventionelles System von Zeichen zu Kommunikationszwecken; dazu gehören neben den natürlichen Sprachen auch künstliche Sprachsysteme (Welthilfssprachen, Systeme logisch-mathemat. Kalküle, Programmiersprachen und Sprachsynthesen), Kommunikationssysteme von Tieren sowie allg. ein Symbolverständnis voraussetzende Zeichensysteme (z. B. Gebärden-S., Trommel-S., Flaggensignale, Morsealphabet).

Visuelle Sprachen fallen demnach am ehesten in die letzte der Kategorien, die angesprochenen Symbole sind visueller Natur. Eine weitere Konkretisierung des Begriffs führt unweigerlich zu ungewollten Einschränkungen, doch auch die Klassifizierung fällt schwer. Unterscheidet man etwa zwischen visuellen Programmiersprachen, visuellen Modellierungssprachen und sonstigen visuellen Sprachen, so ist die Zuordnung nicht immer einfach: Wird zum Beispiel mit einem Schaltplan eine Schaltung programmiert oder modelliert? Gehört eine Bauzeichnung zu einer Modellierungssprache oder nicht?

Der Begriff der Modellierung ist sicherlich viel zu dehnbar, um auf die zweite Frage eine eindeutige Antwort zu geben, doch auch der Übergang von Programmier- zu Modellierungssprachen ist fließend. Sicherlich müssen visuelle Programme, wie Programme generell, formal und ausführbar sein. Jedoch es gibt keinen Grund, warum eine Modellierungssprache diese Kriterien nicht ebenfalls erfüllen darf. Formalheit ist zweifellos wünschenswert und Ausführbarkeit sollte bei Sprachen, mit denen zeitlich dynamische Aspekte modelliert werden, einer Semantik entsprechen.

Denken wir etwa an Statecharts ([Har87]), so gibt es formalisierte Varianten (z.B. [Joh99]), für die eine Schrittsemantik definiert ist. Auf einer Plattform, die das Editieren solcher Statecharts erlaubt und die Schrittsemantik implementiert, lassen diese sich sozusagen ausführen. Kritiker dieses Standpunkts mögen einwerfen, dass Statecharts von den meisten mit ihnen Vertrauten eindeutig zu den Modellierungssprachen gezählt werden und dass das „Ausführen" am ehesten einer Simulation entspricht.

Glücklichweise ist es weder Aufgabe der vorliegenden Arbeit, eine exakte Bestimmung der Begriffe visuelle Sprache, visuelle Programmier- und visuelle Modellierungssprache vorzunehmen, noch Argumente für die Zugehörigkeit der Statecharts, Schaltpläne oder Bauzeichnungen zu der einen oder anderen Kategorie zu finden. Es genügt in diesem Rahmen völlig, allgemein von visuellen Sprachen zu sprechen und sich auf jene Untermenge zu beschränken, die sich entsprechend der in [Bar00] vorgestellten Konzepte – diese verwenden wohlfundierte Theorien, wie die der universellen Algebren, der Constraint Satisfaction Probleme und der algebraischen Graphtransformation – formalisieren lassen.

Die Vermeidung der Zuordung der Statecharts oder anderer Sprachen zu den visuellen Programmier- oder Modellierungssprachen ist jedoch weniger eine Kapitulation vor den Begrifflichkeiten, als eine pragmatisch anwendungsorientierte Sichtweise. Denn für den Anwender ist weniger relevant zu welcher Klasse eine Sprache gehört, sondern vielmehr welche Ausdrucksmöglichkeiten sie bietet und ob es eine Applikation gibt, mit der sie sich ausreizen lassen.

Die Frage der Existenz einer adäquaten Applikation wird insbesondere für Soft- und Hardwareentwickler immer wichtiger. Der Entwurf eines System erfolgt meist mit Hilfe einer oder mehrerer Modellierungssprachen, zum Beispiel der Unified Modelling Language (UML, [UML01]), von der auch in dieser Arbeit Gebrauch gemacht wurde. Da Soft- und Hardwaresysteme mittlerweile Größen erreicht haben, die auf dem Papier unmöglich noch gehandhabt werden können, sind Modellierungsprogramme unerlässlich.

Die meisten der visuellen Modellierungssprachen sind in den letzten zwei Jahrzehnten entstanden und werden noch immer weiterentwickelt. Weiterhin kommen immer neue Sprachen hinzu. Eine Applikation, die also auf eine oder mehrere visuelle Sprachen spezialisiert ist, muss bei jeder Änderung an einer Sprache oder Aufnahme einer neuen angepasst werden. Da diese Anpassungen des Quellcodes meist aufwendig sind und nur von jenen durchgeführt werden können, die im Besitz des Quellcodes sind und auch die softwaretechnischen Fähigkeiten dazu haben, liegt es nahe, eine Applikation zu entwickeln, die von vornherein generisch ausgelegt ist, das heißt mit beliebigen visuellen Sprachen umgehen kann. Bevor ein Endanwender mit einer konkreten Sprache arbeiten kann, muss ein VL-Spezifikateur eine

VL-Umgebung für diese definieren.

Eine solche generische Applikation, genannt GenGEd (abgekürzt für GENeration of Graphical EDitors), wurde im Rahmen zweier Diplomarbeiten von Manuel Schwarze ([Sch99]) und Magnus Niemann ([Nie99]) basierend auf theoretischen Konzepten von Roswitha Bardohl ([Bar00]) an der Technischen Universität Berlin entwickelt. Der Ansatz von GenGEd ist regelbasiert: Die Symbole einer visuellen Sprache, zusammengefasst zu einem Alphabet, können beliebig definiert werden, doch die VL-Diagramme, die Elemente der Sprache, werden durch Regeln einer Grammatik, einer sogenannten Syntaxgrammatik, erzeugt. Der VL-Spezifikateur definiert sowohl Alphabet als auch Syntaxgrammatik, der Endanwender editiert mit Hilfe der Syntaxgrammatik visuelle Sätze.

Neben diesem syntaxgesteuerten Ansatz ist auch ein weiterer denkbar: Der Anwender darf die Alphabetsymbole nach Belieben zu einem visuellen Satz zusammenfügen. Die Freihand-editierten visuellen Sätze müssen allerdings nicht zwangsläufig auch VL-Diagramme, also Elemente der visuellen Sprache, sein, weshalb eine Überprüfung des Editierergebnisses vorzusehen ist. Dies kann durch Parsierung mit Hilfe einer Parsierungsgrammatik erfolgen, die ihrerseits vom VL-Spezifikateur zu erstellen ist.

Abgesehen vom Erzeugen und Verändern von VL-Diagrammen sollte es, wenn das für die visuelle Sprache sinnvoll ist, auch möglich sein, mit ihnen eine Simulation durchzuführen. Die Definition, welcher Art die Simulation sein soll, obliegt ebenfalls dem VL-Spezifikateur. Eine Erweiterung der Simulation stellt die Animation dar. Während der Simulation eine Art Schrittsemantik zugrunde liegt, ist die Animation kontinuierlich, das heißt, sie ermöglicht die Visualisierung beliebig vieler Zwischenphasen für jeden Simulationsschritt.

1.2 Gliederung

Diese Arbeit beginnt (nach diesem einleitenden Kapitel) mit einer Einführung in die GenGEd-Konzepte. Die Begriffe Alphabet, visueller Satz und Grammatik werden erläutert, eine Beschreibung der grundlegenden Ideen für die Parsierung schließt sich an und es folgt eine ausführliche Erklärung der Simulationskonzepte Simulationsausdruck und Simulationsschritt.

Kapitel 3 gibt einen Einblick in das GenGEd-System, das Ausgangspunkt für diese Arbeit war. Alphabet-, Grammatik- und VL-Editor werden analysiert, wobei insbesondere die wesentlichen Mechanismen in den Implementierungen beleuchtet werden und auf Probleme hingewiesen wird. Abschnitte zu Java, ParCon und Agg vervollständigen das Bild.

Es folgt das Kapitel 4, welches auf die von der Implementierung zu erfüllenden Anforderungen eingeht. Dies ist neben der Überarbeitung und Integration vorhandener Komponenten die Entwicklung neuer Datenstrukturen für die Transformation, Parsierung und Simulation sowie der zugehörigen grafischen Benutzerschnittstellen. Hinzu kommt die Implementation für das Speichern und Laden der Datenstrukturen unter Verwendung eines XML-Formats.

Das umfangreiche Kapitel 5 beschreibt den Entwurf und die Implementierung der geforderten Komponenten. Systematisch werden die Datenstrukturen für Grammatik, Transformation, Parsierung und Simulation erläutert, gefolgt von Abschitten über die Datenpersistenz und den grafischen Schnittstellen.

Das Kapitel 6 erklärt am Beispiel hierarchischer Statecharts, wie eine VL-Umgebung für eine visuelle Sprache spezifiziert werden kann. Neben dem Alphabet werden sowohl eine Syntax- als auch eine Parsierungsgrammatik und eine Simulationsspezifikation angegeben.

Kapitel 7 fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick, in welcher Weise das GenGEd-System in Zukunft verbessert und erweitert werden könnte.

Im Anhang befindet sich ein Benutzerhandbuch für GenGEd. Dieses ist nicht vollständig, sondern baut auf die Benutzerhandbücher aus den vorangegangenen Diplomarbeiten auf. In HTML-Form liegt eine mittels Javadoc erzeugte Dokumentation aller Pakete und Klassen vor.

Kapitel 2

Konzepte

Dieses Kapitel diskutiert die konzeptionellen Grundlagen, die dem ursprünglichen GenGEd-System ([Bar00]) und dessen Erweiterung in dieser Arbeit zugrunde liegen.

Die ersten beiden Abschnitte führen die Begriffe Alphabet und Grammatik ein. Im weiteren Sinne entspricht ihre Semantik jener, die sie auch bezüglich einfacher nicht visueller Sprachen besitzen, doch benötigen visuelle Sprachen selbstverständlich eine grafische Ebene, die hier mit konkreter Syntax – die logische Ebene entsprechend mit abstrakter Syntax – bezeichnet wird. Ausführlichere Darstellungen der Alphabet- und Grammatikkonzepte finden sich in [Bar00, Sch99, Nie99]

Es folgen Abschnitte über Parsierung und Simulation. Die Graph-Parsierung im Agg-System, welche für GenGEd verwendet wird, wurde von Thorsten Schultzke im Rahmen einer Diplomarbeit konzipiert und implementiert ([Sch01]). Die Ideen zur Simulation stammen vom Autor der vorliegenden Arbeit und werden ausführlicher und präziser behandelt. Der letzte Abschnitt führt den Begriff der VL-Spezifikation ein und zeichnet ein Gesamtbild des GenGEd-Systems.

2.1 Alphabet

Ein visuelles Alphabet enthält die Grundbestandteile einer visuellen Sprache, die Symbole. Sie sind aufgebaut aus grafischen Primitiva, deren Lagebeziehungen durch grafische Constraints beschrieben werden. Desweiteren können Beziehungen zwischen Symbolen durch Links ausgedrückt werden, die wiederum weitere grafische Constraints zulassen. Die Zuordnung von Attributen, hier Datentypen, erfolgt ebenfalls mittels Links.

2.1.1 Grafische Primitiva und Datentypen

Ein Symbol kann aus einer Reihe grafischer Primitiva, wie Quadraten, Rechtecken, Kreisen, Ellipsen, Linien, Polygonen, statischen Texten, ..., zusammengesetzt werden. Wird eines der Objekte verwendet, so ist es durch einen Namen identifizierbar und besitzt mehrere Eigenschaften. Neben typabhängigen Eigenschaften, wie Linien-, Füll- oder Texteigenschaften, existieren universelle Eigenschaften, die jedes Objekt aufweist (Abb. 2.1). Dazu zählen insbesondere Position und Größe sowie eine Tiefenposition, die angibt, durch welche anderen grafischen Primitiva eines Symbols ein Primitivum verdeckt wird, beziehungsweise welche es verdeckt. Die Gesamtheit der Eigenschaften eines Objekts definiert sein Default-Layout.

Als besondere Primitiva seien Gruppen erwähnt, deren einzige Aufgabe darin besteht, andere Primitiva zu beinhalten. Sie geben dem Spezifikateur einer visuellen Sprache Mittel an die Hand, die Bestandteile der Symbole geeignet hierarchisch zu strukturieren. Auch Gruppen besitzen einen Namen; ihre Position und Größe resultiert aus dem kleinsten Rechteck, welches alle enthaltenen Objekte umschließt (Bounding-Box-Konzept). Die Namen der in einer Gruppe enthaltenen Primitiva müssen, um deren eindeutige Identifizierung sicherzustellen, paarweise verschieden sein.

Neben den statischen Primitiva werden zur Definition einer visuellen Sprache auch Objekte benötigt, die einen veränderbaren Wert repräsentieren, die so genannten Datentypen. Auch sie haben einen Namen, Position und Größe sowie datentypspezifische Eigenschaften.

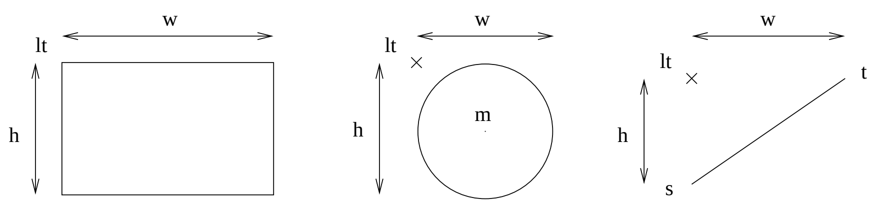

Abbildung 2.1: Eigenschaften der Primitiva.

2.1.2 Constraints

Zur Definition der Lagebeziehungen zwischen den benannten Bausteinen werden grafische Constraints verwendet. Ein einfaches Constraint (Low Level Constraint, LLC) ist durch eine Gleichung oder Ungleichung über Eigenschaften von beteiligten Objekten identifizierende Variablen gegeben. In Abbildung 2.2 etwa wird festgelegt, dass der Mittelpunkt des Kreises auf der Geraden durch die rechte Seite des Rechtecks liegen soll. Die Notation ist für mit gängigen objektorientierten Sprachen Vertraute intuitiv: Auf den Bezeichner des Objekts folgt nach einem Punkt der Name des entsprechenden Attributs, eine Eigenschaft jenes wird durch den entsprechenden Bezeichner nach einem weiteren Punkt annotiert usw. Ist ein Objekt in einer unter Verwendung von Gruppen strukturierten Menge von Primitiva zu identifizieren, so geschieht dies durch Angabe des mittels Punkten getrennten Pfads der Namen von der obersten Gruppe bis zum gewünschten Primitivum (auch bezeichnet als „Path Expression").

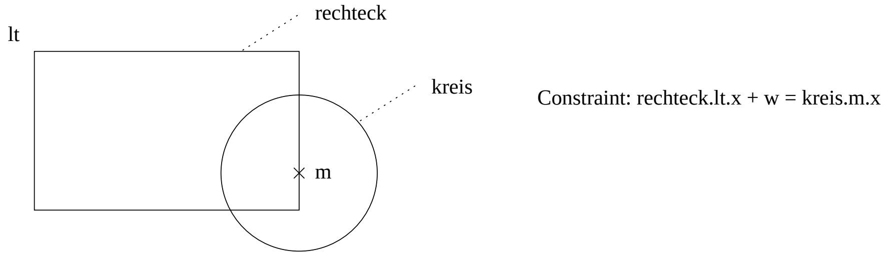

Abbildung 2.2: Beispiel eines LLCs.

Da die Definition komplexerer Lagebeziehungen zwischen Objekten mittels LLCs recht mühselig ist, können High Level Constraints (HLCs) definiert werden, um diese Aufgabe zu vereinfachen. HLCs sind durch Objekte parametrisiert und enthalten ein oder mehrere LLCs, die durch logisches Und oder Oder verknüpft sind. Beispiele für High Level Constraints zeigt die Abbildung 2.3.

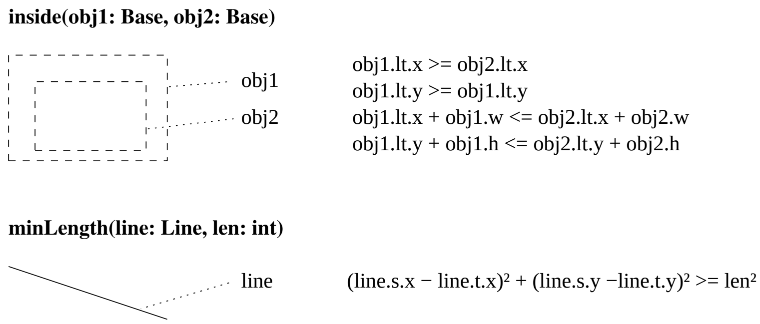

Abbildung 2.3: Beispiele zweier HLCs.

Da HLCs durch die sie definierenden LLCs ersetzt werden können, seien für die kurze Einführung der folgenden Begriffe nur letztere gemeint, ohne dass dadurch die Anwendbarkeit der Termini für die Konzepte, welche High Level Constraints zulassen, eingeschränkt wäre.

Constraint Network: Eine Kombination von Constraints über gemeinsame Variablen heißt Constraint Network (CN).

Constraint Satisfaction: Eine Belegung der in einem Constraint vorkommenden Variablen, bei der die Auswertung des Constraints *true* liefert, heißt Lösung des Constraints. Eine Belegung der Variablen eines Constraint Networks, die alle Constraints löst, heißt Constraint Satisfaction.

Constraint Satisfaction Problem: Ein Constraint Satisfaction Problem (CSP) ist gegeben durch ein Constraint Network, für das eine Constraint Satisfaction gesucht ist.

2.1.3 Symbole und Links

Ein Symbol besteht aus einem Namen, welcher das Symbol auf der Ebene der abstrakten Syntax repräsentiert, und einer Grafik auf der konkreten Syntaxebene. Die Symbolgrafik ist gegeben durch eine Menge von grafischen Primitiva – es gilt analog das in Abschnitt 2.1.1 über Gruppen Geschriebene – und ein CSP induziert durch ein Constraint Network über den Variablen dieser Primitiva.

Beispiel 2.1.1 *In der Abbildung 2.4 ist ein Beispiel gegeben, wie das Klassensymbol für Klassendiagramme der UML spezifiziert werden kann: Es besteht aus drei übereinander liegenden Rechtecken, die Platz für den Klassennamen, eine Attribut- und eine Methodenliste bereit stellen sollen. Die sechs gegebenen LLCs sorgen dafür, dass die Rechtecke an der gleichen x-Position liegen, gleich breit sind und dass das zweite Rechteck in y-Richtung an das erste anschließt, ebenso wie das dritte an das zweite.*

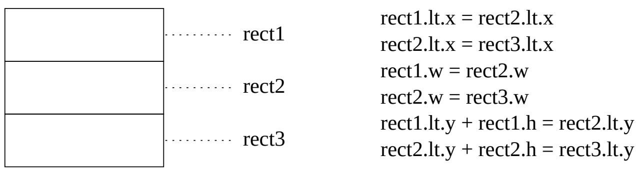

Abbildung 2.4: Definition des Klassensymbols.

Beispiel 2.1.2 *Die weiteren Konzepte werden am Beispiel einfacher, nicht hierarchischer Statecharts erläutert. An dieser Stelle seien schon einmal die verwendeten Symbole dargestellt (Abb. 2.5): Zum einen wird ein State-Symbol benötigt, zum anderen ein Symbol das eine Transition repräsentiert. Beide bedürfen im Gegensatz zum zuvor gezeigten Klassensymbol nicht wirklich Constraints, doch sind hier nichtsdestotrotz welche gegeben, die spätere pathologische Fälle vermeiden sollen. Das State-Symbol besteht aus einem einfachen Rechteck, dem eine Mindestgröße von* $(3, 2)$ *Einheiten zugesichert wird. Für die Transition wird eine Linie mit einem Pfeil an deren Ende verwendet; sie besitzt eine minimale Länge von* 2 *Einheiten.*

Symbole werden zueinander durch Links in Beziehung gebracht. Ein Link ist auf der abstrakten Ebene gegeben durch einen Namen und die Bezeichnung eines Quell- und eines Zielsymbols sowie auf der konkreten Ebene durch ein CSP über den Variablen von Quell- und Zielsymbol. Die Richtung der Links ist von nicht

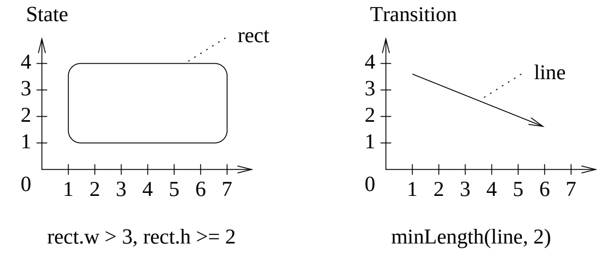

Abbildung 2.5: Symbole der einfachen Statecharts.

unerheblicher Bedeutung: Geht von einem Symbol ein Link aus, so darf später (in einem visuellen Satz) eine Instanz dieses Symbols nur dann existieren, wenn auch eine Instanz des Zielsymbols des Links vorhanden ist, mit dem erstere durch eine Instanz des Links verbunden ist.

Beispiel 2.1.3 *Im Beispiel der einfachen Statecharts werden zwei Links, begin und end, definiert, um den Anfangs- bzw. Endzustand einer Transition festzulegen. Abbildung 2.6 visualisiert diese, wobei die abstrakte von der konkreten Syntax separiert wird; die gepunkteten Linien setzen die entsprechenden Elemente der beiden Ebenen in Verbindung. Die adhoc angegebenen Constraints startPointOnBorder und endPointOnBorder – von deren Implementierung in LLCs wollen wir hier absehen – seien erfüllt, wenn der Linienstart- respektive -endpunkt auf dem Rand eines zweiten Objekts, hier dem Rechteck, liegt. Die Richtung der beiden Links impliziert, dass eine Transition nur dann vorhanden sein kann, wenn sowohl ein Zustand, an dem sie beginnt, als auch einer, an dem sie endet, (diese können auch identisch sein) existieren. Dies ist zweifellos wünschenswert, denn im umgekehrten Fall wären etwa keine isolierten Zustände modellierbar, sehr wohl aber frei hängende Transitionen.*

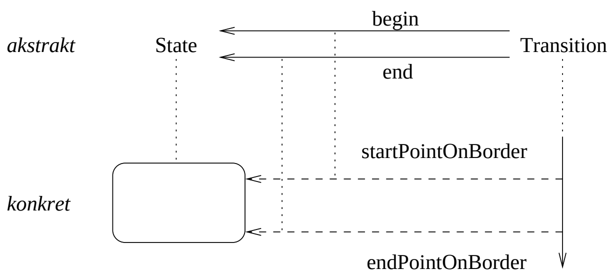

Abbildung 2.6: Links der einfachen Statecharts.

Die in Abschnitt 2.1.1 eingeführten Datentypen – in [Bar00] auch als Attributsymbole bezeichnet (im Gegensatz zu lexikalischen Symbolen) – werden durch Links, so genannte Attributierungslinks, an ein Symbol gebunden. Zu jedem Datentyp muss genau ein Attributierungslink existieren. Diese Besonderheit lässt den

Datentypen die Bedeutung von Attributen der Symbole zukommen.

Beispiel 2.1.4 *Für die einfachen Statecharts sei als Datentyp ein Zustandsname vorgesehen, der mittels eines inside Constraints im Rechteck des State-Symbols platziert wird. Abbildung 2.7 veranschaulicht den Sachverhalt.*

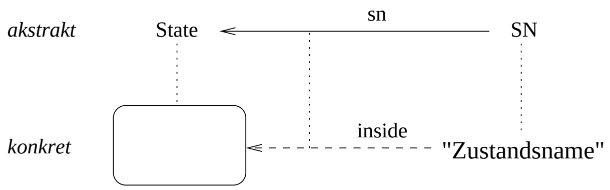

Abbildung 2.7: Attributierungslink zwischen SN und State.

2.1.4 Alphabet, Typgraph, Signatur

Ein visuelles Alphabet besteht aus Symbolen, Datentypen und Links wie sie in den vorangegangenen Abschnitten eingeführt wurden, es müssen jedoch einige weitere Bedingungen erfüllt sein:

- Die Namen der Symbole, Datentypen und Links müssen paarweise verschieden sein.
- Jeder Datentyp muss durch genau einen Attributierungslink an ein Symbol gebunden sein.
- Das Alphabet muss azyklisch sein.

Der erste Punkt dient der eindeutigen Identifizierbarkeit der Elemente, der zweite, zuvor schon erwähnt, gibt den Datentypen den Charakter von Attributen, der dritte Punkt bedarf einer näheren Erläuterung: Fasst man Symbole und Datentypen als Knoten eines Graphen auf und die Links als Kanten zwischen diesen, so lässt sich das Alphabet als Graph darstellen. Und von eben diesem Graphen wird Zyklenfreiheit gefordert. Der Grund für diese Forderung liegt im Wesen des später noch beschriebenen Algorithmus, welcher zu einem Alphabet Grammatikregeln generiert.

Angesichts des Attributcharakters der Datentypen liegt es nahe, diese nicht als Knoten im Graphen zu verstehen, sondern als Attribute der Symbolknoten. Das Alphabet wird damit zum attributierten Graphen.

Schließlich sei noch eine letzte Sichtweise auf ein Alphabet erwähnt, die insbesondere für die formale Fundierung der Konzepte von Bedeutung ist (siehe dazu [Bar00]). Werden nämlich die Symbole als Sortensymbole und die Links als

Operationssymbole verstanden, so ist ein Alphabet nichts weiter als eine algebraische Signatur, ob der ausschließlich unären Operationssymbole genauer eine Graphstruktur-Signatur.

Beispiel 2.1.5 *Die abstrakte Syntax des Alphabets zu den einfachen Statecharts ist in Abbildung 2.8 dargestellt, Abbildung 2.9 zeigt es als algebraische Signatur.*

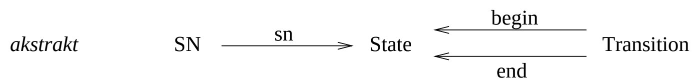

Abbildung 2.8: Das Alphabet der einfachen Statecharts.

sorts:	State, Transition, SN	
opns:	begin:	Transition $\rightarrow$ State
	end:	Transition $\rightarrow$ State
	sn:	SN $\rightarrow$ State

Abbildung 2.9: Die Signature der einfachen Statecharts.

2.1.5 Visueller Satz, Algebra

Ein visueller Satz ist eine Instanz eines visuellen Alphabets, das heißt je nach Interpretation des Alphabets entweder als Typgraph oder als algebraische Signatur, kann ein visueller Satz entweder als über dem Alphabet-Graphen getypter Graph oder als Algebra zur Signatur aufgefasst werden. Damit ergibt sich folgende Bedingung, die ein visueller Satz erfüllen muss: Existiert eine Instanz eines Alphabet-Symbols, so müssen auch

- Instanzen aller Links, die bei diesem Symbol beginnen, und Instanzen der Zielsymbole der Links sowie
- Instanzen aller zugehörigen Datentypen und Attributierungslinks

vorhanden sein. Anders formuliert, wird sowohl die Totalität der Algebra-Operationen als auch eine totale Attributierung gefordert.

Neben der Erfüllung dieser strukturellen Bedingungen ist jedoch auch die Lösbarkeit des zugehörigen CSPs zu fordern, um eine gültige graphische Repräsentation sicherzustellen.

Weiteren Aufschluss über die formalen Hintergründe gibt [Bar00].

Ist eine visuelle Sprache, eine Menge von visuellen Sätzen über einem Alphabet, gegeben, so bezeichnen wir die Elemente der Sprache als VL-Diagramme.

2.2 Grammatik

Da ein Alphabet in den meisten Fällen noch nicht hinreichend einschränkend sein wird, um eine visuelle Sprache zu definieren – die Menge aller visuellen Sätze über einem Alphabet enthält gewöhnlich auch unerwünschte Exemplare – liegt es nahe, endliche Produktionensysteme, Grammatiken, zu verwenden, um diese Menge geeignet auf die Elemente der visuellen Sprache zu reduzieren.

Eine solche Grammatik besteht im Wesentlichen aus einem visuellen Startsatz und einer Menge von Produktionen (Regeln). Grundlage der Regeln und des Ersetzungsprozesses sind Morphismen zwischen visuellen Sätzen.

2.2.1 Morphismen

Ein Morphismus zwischen zwei visuellen Sätzen ist nur auf der Ebene der abstrakten Syntax definiert. Je nach Sichtweise handelt es sich dabei um einen partiellen Graphmorphismus oder einen partiellen Graphstrukturhomomorphismus. In jedem Falle werden Symbole sortentreu und verträglich mit den Links abgebildet.

2.2.2 Regeln

Eine Regel besteht aus einer linken Regelseite (left hand side, LHS) und einer rechten Regelseite (right hand side, RHS) und beschreibt die Ersetzung der linken durch die rechte Regelseite. Bei der Wortersetzung werden dazu keine weiteren Informationen benötigt, da dort ein einfaches Entfernen des Wortes der linken und Einfügen des Wortes der rechten Seite durchgeführt werden kann. Für Regeln auf graphartigen Strukturen, wie sie hier vorliegen, muss ein Morphismus zwischen linker und rechter Regelseite die zusätzlich erforderlichen Informationen tragen, konkret ein Morphismus, wie er im vorangegangenen Abschnitt beschrieben wurde.

Die Anwendung einer solchen Regel auf einen visuellen Satz erfolgt in mehreren Schritten, im Folgenden mit der Sicht der abstrakten Syntax der visuellen Sätze als Graphen und unter Verwendung der üblichen Graphtransformations-Nomenklatur erklärt:

1. Neben dem zu transformierenden visuellen Satz, dessen abstrakte Syntax als Hostgraph bezeichnet wird, und der anzuwendenden Regel muss ein Ansatz (Match) gegeben sein, der die Stelle der Regelanwendung im Hostgraphen markiert. Dies geschieht durch einen Morphismus (Matchmorphismus), von dem gewöhnlich zusätzliche Eigenschaften, etwa Totalität, gefordert werden.

2. Das Pushout von Regel- und Matchmorphismus wird gebildet. Der Pushoutgraph erfüllt nicht zwangsläufig das Kriterium der Totalität der Algebraoperationen. Um dies sicherzustellen werden sukzessive alle Knoten (und etwaige

inzidente Kanten) entfernt, denen zumindest eine für die Totalität notwendige Kante fehlt.

3. Zum Ergebnisgraphen wird, wenn möglich, ein visueller Satz gebildet:
   - Knoten und Kanten, die auch schon im Hostgraphen vorhanden waren, werden die Symbole und Links des ursprünglichen visuellen Satzes zugeordnet, neuen Knoten und Kanten Instanzen der entsprechenden Symbole bzw. Links aus dem Alphabet.
   - Das CSP des visuellen Satzes wird gelöst.

   Ist das Ergebnis der Transformation kein korrekter visueller Satz, weil zum Beispiel das CSP nicht gelöst werden konnte so ist die Regel nicht anwendbar.

Vernachlässigt wurde, dass es sich bei den zu transformierenden Graphen um attributierte Graphen handelt. Auf weitere Details hierzu soll in dieser Arbeit verzichtet werden, sodass hier lediglich auf entsprechende Literatur verwiesen sei: [LKW93].

Zu erwähnen bleibt, dass in einer Regel Variablen an Stelle der konkreten Attributwerte vorkommen dürfen. Die in der linken Regelseite enthaltenen Variablen werden durch einen gegebenen Match entsprechend der Attribute im Hostgraphen belegt. Auf der rechten Regelseite dürfen nur Variablen der linken Regelseite vorhanden sein; an ihrer Stelle wird bei der Transformation mit den Werten, mit denen sie belegt wurden, gearbeitet.

Desweiteren darf eine Regel Parameter vom Typ der im Alphabet vorhandenen Datentypen besitzen. Die Parameterbezeichner dürfen in der Regel beliebig verwendet werden. Alle Parameter müssen vor der Transformation belegt werden; es wird danach wie mit belegten Variablen verfahren.

2.2.3 Anwendungsbedingungen

Um eine spezifischere Anwendung einer Regel zu erzwingen, können zwei Arten von Anwendungsbedingungen formuliert werden: Zum einen negative Anwendungsbedingungen (negative application conditions, NACs), zum anderen Attributbedingungen.

Eine NAC besteht aus einem Graphen und einem Morphismus von der linken Regelseite in diesen Graphen. Kann der NAC-Graph im Hostgraphen „wiedergefunden“ werden, so darf die Regel nicht angewandt werden. Im NAC-Graphen dürfen sowohl Variablen (auch welche, die nicht in der linken Regelseite vorkommen) als auch Parameterbezeichner verwendet werden.

Eine Attributbedingung ist ein boolescher Ausdruck über den Variablen und Parametern der Regel. Nach der Belegung dieser, kann er zu einem Wahrheitswert ausgewertet werden. Die Regel darf nur angewandt werden, wenn die Auswertung aller Attributbedingungen *true* liefert.

Zusammengenommen darf eine Regel also nur angewandt werden, wenn keine der NACs, jedoch alle Attributbedingungen erfüllt sind.

Neben den Anwendungsbedingungen gibt es noch eine zweite Möglichkeit die Anwendbarkeit einer Regel einzuschränken. Das im Unterabschnitt 2.2.2 erläuterte Verfahren zu Regelanwendung verwendet für die Graphtransformation den so genannten Single-Pushout-Ansatz (SPO-Ansatz). Ohne auf die Details einzugehen (siehe dazu [EHK+97]), unterscheidet er sich im Ergebnis der Anwendung vom Double-Pushout-Ansatz (DPO-Ansatz) im Wesentlichen beim Löschen von Knoten. Kommt ein Knoten auf der linken, jedoch nicht auf der rechten Regelseite vor, so wird der entsprechende Knoten im Regelansatz bei der Anwendung der Regel aus dem Hostgraphen gelöscht. Beim SPO-Ansatz kann eine Regel immer angewandt werden, wenn ein adäquater Match gegeben ist, was nach sich zieht, dass nicht nur der Knoten, sondern automatisch auch seine inzidenten Kanten sowie alle Knoten und Kanten, die die Totalität der Algebraoperationen verhindern, entfernt werden. Im DPO-Ansatz können nur solche Kanten gelöscht werden, die auch im Regelansatz liegen. Das heißt insbesondere, dass nicht zu entfernende zu einem zu löschenden Knoten inzidente Kanten im Ergebnisgraphen quasi „in der Luft hängen“ würden. Eine solche Regelanwendung ist beim DPO-Ansatz deshalb von vornherein nicht zulässig. Um trotz der Verwendung des SPO-Ansatzes NACs einzusparen, die ein DPO-ähnliches Verhalten erzwingen sollen – gewöhnlich ist dieses nämlich einsichtiger und deshalb erwünscht – wird ein zusätzlicher Transformationsparameter, die so genannte Dangling-Edge-Condition (kurz: Dangling-Condition), eingeführt. Wird sie verwendet, so darf eine Regel nicht angewandt werden, wenn es zu normalerweise automatisch zu entfernenden „hängenden Kanten“ kommen würde.

2.2.4 Grammatiken

Eine Grammatik besteht aus einem visuellen Satz und einer endlichen Menge von Regeln, wie sie in den vorangegangenen Abschnitten beschrieben wurden. Jede der Regeln darf endlich viele NACs und Attributbedingungen besitzen. Die Dangling-Condition kann optional für die Grammatik eingeschaltet werden. Auf kanonische Weise induziert eine Grammatik eine visuelle Sprache: Diese Sprache enthält alle aus dem visuellen Satz der Grammatik mit Hilfe der Regeln ableitbaren visuellen Sätze. Dabei heißt ein visueller Satz mittels einer Regelmenge aus einem anderen Satz ableitbar, wenn er aus letzterem durch null- oder mehrfache Anwendung von einer oder mehreren der Regeln erzeugbar ist.

Im Folgenden bezeichnen wir Grammatiken, deren Verwendungszweck die Erzeugung einer visuelle Sprache ist, als Syntaxgrammatik. Weitere Grammatiktypen sind Parsierungsgrammatik und Simulationsgrammatik. Auf diese wird in den

Abschnitten 2.3 und 2.4 näher eingegangen.

Eine einfache Syntaxgrammatik kann durch den im Folgenden beschriebenen Algorithmus erzeugt werden. Sie enthält für jedes Symbol eine Regel, die ein solches Symbol einfügen und eine, die es löschen kann, sowie für jeden Datentypen eine Regel, um Werte dieses Typs zu ändern. Das Startdiagramm der Grammatik ist der leere visuelle Satz. Einfüge- und Löschregeln werden wie folgt generiert:

1. Ausgehend von der leeren Regel wird für die Löschregel auf der linken und für die Einfügeregel auf der rechten Seite das entsprechende Symbol eingefügt.
2. - Zu jedem mit dem Symbol assoziierten Datentypen muss in der Einfügeregel ein Parameter addiert werden, der auf der rechten Regelseite entsprechend verwendet wird.
   - Wird die Dangling-Condition nicht verwendet, so muss zur Löschregel für jeden (im Alphabet) zum Symbol führenden Link eine NAC hinzugefügt werden. Im NAC-Graphen wird das Symbol, der Link und das Quellsymbol des Links eingefügt. Der NAC-Morphismus enthält vorerst nur das Mapping des gegebenen Symbols der LHS auf jenes im NAC-Graphen.
3. Die Regel samt ihren NACs wird, wie später beschrieben, vervollständigt und total attributiert.

Eine Regel zum Ändern eines Datentypwerts enthält auf beiden Regelseiten das zum Datentyp gehörende Symbol – der Regelmorphismus bildet diese aufeinander ab – sowie auf der linken Seite eine Variable zum Datentypen und auf der rechten Seite ein Regelparameterbezeichner. Die Regel wird ebenfalls vervollständigt und total attributiert.

Die Regelvervollständigung soll die Totalität der Algebraoperationen der linken und rechten Regelseite und der NACs sicherstellen. Dieser Teilalgorithmus arbeitet rekursiv: Hat ein Symbol in LHS oder RHS bezüglich einer Algebraoperation kein Bild, so wird ein solches eingefügt (zusammen mit dem zugehörigen Link). Parallel dazu wird es (wenn möglich ebenfalls mit Link) auch in den verbleibenden Algebren (aus LHS, RHS, NACs) ergänzt, ebenso wie ein entsprechendes Mapping in den betroffenen Morphismen (Regelmorphismus, NAC-Morphismen). Da Alphabete azyklisch sind, ist die Termination dieses Algorithmus sichergestellt, RHS und LHS werden also nach endlicher Zeit in diesem Sinne vollständig sein. Anschließend sind noch die NAC-Algebren zu vervollständigen. Dies geschieht analog, jedoch ausschließlich lokal in der jeweils betroffenen Algebra, also ohne Auswirkungen auf LHS, RHS oder irgendwelche Morphismen.

Der letzte Schritt besteht in der totalen Attributierung aller visuellen Sätze einer Regel. Fehlt auf der linken Regelseite ein Attribut, so wird seiner statt eine neue Variable gesetzt. Existiert auf der rechten Regelseite oder in einer NAC das

Bild des mit dem Attribut assoziierten Symbols, jedoch nicht das Attribut selbst (oder ein entsprechender Parameter), so wird dort die gleiche Variable verwendet. Ist die linke Seite auf diese Weise total attributiert worden, so ist es auch die rechte Seite; es müssen deshalb nur noch die fehlenden Attributwerte in den NACs durch neue Variablen ergänzt werden.

Die Folgerung lässt sich ableiten, jedoch ohne dies hier zu beweisen, dass die von einer mittels des angegebenen Algorithmus generierten Grammatik erzeugten Sprache alle visuellen Sätze zu dem gegebenen Alphabet enthält. Mehr noch, mittels der Löschregeln kann aus jedem visuellen Satz das leere Diagramm und damit mit Hilfe aller Regeln aus einem beliebigen Satz ein beliebiger anderer abgeleitet werden. Die algorithmisch erzeugten Regeln (auch: Alphabetregeln) bilden damit eine Grundlage zum syntaxgesteuerten Editieren von visuellen Sätzen.

Beispiel 2.2.1 *Abbildung 2.10 zeigt die Alphabetregeln für die einfachen Statecharts bei Verwendung der Dangling-Condition.*

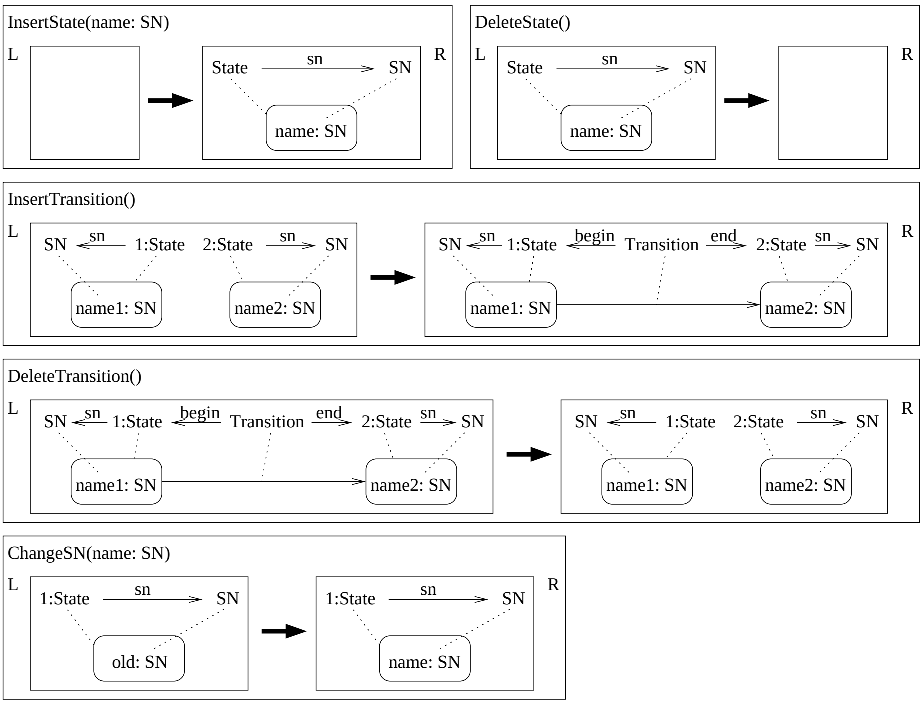

Abbildung 2.10: Alphabetregeln für die einfachen Statecharts.

2.3 Parsierung

Unter Parsierung soll hier die Prüfung der Zugehörigkeit eines visuellen Satzes zu einer visuellen Sprache verstanden werden. Die visuelle Sprache wird durch eine Parsierungsspezifikation beschrieben, deren wichtigster Bestandteil eine Parsierungsgrammatik ist. Im Gegensatz zu einer Syntaxgrammatik, durch die eine visuelle Sprache als Menge der aus dem Startdiagramm ableitbaren visuellen Sätze definiert wird, beinhaltet die durch eine Parsierungsgrammatik bestimmte Sprache all jene visuellen Sätze, aus denen mittels der Grammatikregeln das Stoppdiagramm der Grammatik (Uminterpretation des Startdiagramms) ableitbar ist. Optional kann eine Parsierungsspezifikation eine Layerfunktion und das Resultat einer Analyse der kritischen Paare enthalten. Diese zusätzlichen Informationen dienen in der Anwendung primär der Verbesserung der Effizienz der Parsierung.

Wie auch schon bei der Transformation visueller Sätze auf die Transformation attributierter Graphen zurückgegriffen wurde, soll auch für die Parsierung visueller Sätze die Graphparsierung verwendet werden ([BST01]).

2.3.1 Layerfunktion

Eine Layerfunktion, letztlich bestehend aus drei Funktionen, ordnet jeder Regel der Parsierungsgrammatik eine (Rule-Layer) und jedem Symbol des Alphabets zwei natürliche Zahlen (Creation- und Deletion-Layer) zu. Durch die Rule-Layers wird die Menge der Regeln quasi in Schichten eingeteilt. Wurde bei der Parsierung eine Regel angewandt, so wird danach nur eine Regel der gleichen oder einer höheren Schicht gewählt. Aufgrund weiterer Bedingungen, die von einer Layerfunktion zu erfüllen sind – Näheres siehe [Sch01] – sichert dies die Termination der Parsierung zu und hilft sie zu beschleunigen.

Es bleibt anzumerken, dass eine Layerfunktion für die Graphparsierung Creation- und Deletion-Layer sowohl für die Knoten als auch für die Kanten zu definieren hat. Wie im vorangegangenen Absatz beschrieben, ordnet eine Layerfunktion für eine Parsierungsgrammatik für visuelle Sprachen jedoch nur den Symbolen, das heißt den Knoten, die entsprechenden Layers zu. Für die Rückführung auf die Graphparsierung muss die Layerfunktion deshalb so erweitert werden, dass auch den Kanten eine Creation- und ein Deletion-Layer zugewiesen wird, was geschieht, indem eine Kante die gleichen Layers wie ihr Quellknoten bekommt.

2.3.2 Kritische Paare

Können in einem Hostgraphen zwei verschiedene, sich überlappende Ansätze einer oder zweier Regeln gefunden werden, so besteht die Möglichkeit, dass aus den beiden resultierenden Graphen disjunkte Mengen von Graphen ableitbar sind. Da bei der Parsierung die Ableitbarkeit eines bestimmten, nämlich des Stoppgraphen untersucht wird, ist dies von besonderer Relevanz, da in einem solchen Fall beide Varianten untersucht werden müssen. Um den Arbeitsaufwand bei der eigentlichen Parsierung zu verringern, können die Regeln der Grammatik schon im Vorfeld untersucht werden. Dabei werden alle möglichen Regelpaare dahingehend analysiert, welche Überlappungen ihrer linken Regelseiten denkbar sind. Aus jedem Überlappungsgraphen gewinnt man durch Anwendung der entsprechenden Regeln zwei Graphen, ein so genanntes kritisches Paar – häufig wird als kritisches Paar auch das Paar der Regelanwendungen bezeichnet.

Wie das Ergebnis der Analyse der kritischen Paare für die eigentliche Parsierung nutzbar gemacht werden kann, ist in [Sch01] nachzulesen.

2.4 Simulation

Da viele in der Informatik gängige visuelle Sprachen der Modellierung von Software- oder Hardwaresystemen dienen, welche gewöhnlich ein zeitlich dynamisches Verhalten aufweisen, bietet ein großer Teil davon Möglichkeiten zur Beschreibung dieser Dynamik. Denken wir zum Beispiel an endliche Automaten, Statecharts oder Petrinetze, so fällt ins Auge, dass zumeist eine Schrittsemantik (in manchen Fällen sogar eine ganze Reihe verschiedener Semantiken) definiert ist oder wenigstens denkbar wäre.

Um eine Schrittsimulation zu umfassen, müssen die bisher vorgestellten Konzepte lediglich um eine Spezifikation erweitert werden, die festlegt, welches VL-Diagramm in einer Simulation auf ein gegebenes Diagramm folgt bzw. welche Diagramme folgen können – denn Nichtdeterminismus soll nicht ausgeschlossen werden. Das einzige bisher vorgestellte Mittel, mit dem Transformationen von Diagrammen beschrieben werden können, sind Regeln. Wie schon im Abschnitt 2.2.4 erwähnt, soll eine als Simulationsgrammatik bezeichnete Grammatik die Grundlage für eine Simulationsspezifikation sein. Für manche Beispiele mag es schon genügen, die Schritte einer Simulation durch einfache Regelanwendungen zu beschreiben, doch es lassen sich auch Beispiele finden, bei denen die Anzahl der Regelanwendungen für einen Schritt der Simulation von der Zahl bestimmter Elemente im Diagramm abhängt, etwa wenn ein Attribut aller im Diagramm vorhandenen Symbole eines bestimmten Typs zu verändern ist. Offenbar wird also ein mächtigeres Konzept zur Beschreibung eines Simulationsschritts benötigt, welches naheliegend als Simulationsschritt bezeichnet und im Abschnitt 2.4.2 eingeführt wird. Kern eines Simulationsschritts ist ein Simulationsausdruck, über den der vorangehende Abschnitt 2.4.1 informiert.

2.4.1 Simulationsausdruck

Da der Begriff des Simulationsausdrucks semiformal definiert werden soll, beginnen wir mit der kurzen Einführung einer Notation für Begrifflichkeiten aus den Abschnitten 2.1 und 2.2 sowie einigen Definitionen.

- Die Menge der natürlichen Zahlen inklusive 0 sei mit $\mathbb{N}$ bezeichnet, die Menge der Wahrheitswerte mit $\mathbb{B} := \{true, false\}$.

- Sei A ein visuelles Alphabet, so bezeichne DT_A die Menge der Datentypen des Alphabets, $U = (U_t)_{t \in DT_A}$ die Mengenfamilie aller Datentypwerte und $VS(A)$ die Menge der visuellen Sätze über A.

- Eine Grammatik $G = (A, S, R)$ besteht aus einem Alphabet A, einem Startdiagramm $S \in VS(A)$ und einer Regelmenge R.

- Für eine Regel $r \in R$ bezeichne $P_r = (P_{r,t})_{t \in DT_A}$ die Datentyp-indizierte Familie der Parametermengen der Regel sowie $LHS_r, RHS_r \in VS(A)$ die linke bzw. rechte Regelseite.

- Eine Familie $pa = (pa_t)_{t \in DT_A}$ von Abbildungen $pa_t \colon P_{r,t} \to U_t$ heißt Parameterbelegung für die Regel r. Die Menge aller Parameterbelegungen zu einer Regel r bezeichne $PA(r)$.

- Zu einer Regel $r \in R$, einer Parameterbelegung $pa \in PA(r)$ und einem visuellen Satz $H \in VS(A)$ bezeichne $M(r, H, pa)$ die Menge aller gültigen Matches von r in H unter der Belegung pa.

- Jede Familie $MS = (MS_{r,H})_{r \in R, H \in VS(A)}$ von partiellen Abbildungen $MS_{r,H} \colon PA(r) \multimap (LHS_r \multimap H)$ für die $M(r, H, pa) \neq \emptyset \implies pa \in dom(MS_{r,H}) \wedge MS_{r,H}(pa) \in M(r, H, pa)$ gilt, heißt Matchselektor.

- Der aus der Anwendung einer Regel $r \in R$ bei einer Belegung $pa \in PA(r)$ auf einen visuellen Satz $H \in VS(A)$ mit einem Match $m \in M(r, H, pa)$ resultierende visuelle Satz sei mit $T(r, pa, H, m)$ bezeichnet.

Definition 2.4.1 *(Simulationsausdrücke) Die Menge der Simulationsausdrücke $SE(G, X)$ über einer visuellen Grammatik $G = (A, S, R)$ und einer Familie $X = (X_t)_{t \in DT_A}$ von abzählbaren Variablenmengen ist wie folgt induktiv definiert:*

1. ***true**, **false** $\in SE(G, X)$.*

2. *Sind $r \in R$ eine Regel und $p = (p_t)_{t \in DT_A}$ eine Familie von Abbildungen $p_t \colon P_{r,t} \to X_t \cup U_t$, so ist **rule**$(r, p) \in SE(G, X)$.*

3. *Sind $e, f, g \in SE(G, X)$, so auch*
   (a) *(e **and** f) $\in SE(G, X)$,*
   (b) *(e **andc** f) $\in SE(G, X)$,*
   (c) *(e **or** f) $\in SE(G, X)$,*
   (d) *(e **orc** f) $\in SE(G, X)$,*
   (e) ***if** e **then** f **else** g **fi** $\in SE(G, X)$,*
   (f) ***while** e **do** f **done** $\in SE(G, X)$.*

4. *$SE(G, X)$ enthält keine weiteren Elemente.*

□

Die induktive Definition von $SE(G, X)$ ist offensichtlich frei, weshalb eine induktive Definition einer Auswertungsfunktion für Simulationsausdrücke eine wohldefinierte Abbildung liefert. Die Semantik, die die Namen der Ausdrücke nahe legen, soll von dieser selbstverständlich implementiert werden. Insbesondere ist mit einem While-Ausdruck eine Schleife intendiert, deren Termination von der Auswertung des ersten Unterausdrucks abhängig ist. Da die Auswertung von nicht terminierenden Schleifen wenig sinnvoll ist und bei einer induktiven Definition der Auswertungsfunktion zu Problemen führt, soll dafür nur die größte Untermenge von $SE(G, X)$ berücksichtigt werden, für deren Elemente die Auswertung terminiert. Diese Menge ist abhängig von den Auswertungsparametern (Grammatik G, visuellem Satz $H \in VS(A)$, Matchselektorfolge $(MS_n)_{n\in\mathbb{N}}$ und Variablenbelegung ass) und wird im Folgenden mit $ESE(G, (MS_n)_n, ass, H)$ bezeichnet werden.

Zum besseren Verständnis sei vor der Definition der Auswertung der Simulationsausdrücke kurz erläutert, welche Anwendung sie finden sollen. Ein Simulationsausdruck kann als iteratives Programm verstanden werden, seine Auswertung entspricht der Ausführung dieses Programms. Damit insbesondere **if then else fi**- und **while do done**-Ausdrücke sinnvoll sind, muss die Auswertung ihrer Bedingung einen booleschen Wert liefern. Für diese Bedingungen kommen beliebige Simulationsausdrücke infrage, sie alle müssen also zu einem booleschen Wert ausgewertet werden können. Damit wird auch klar, wie **and** und **or** und deren konditionale Varianten zu verstehen sind. Von entscheidender Bedeutung sind die **rule**()-Ausdrücke. Ihre Auswertung entspricht einer Regelanwendung, die Anwendbarkeit definiert das Auswertungergebnis. Dazu muss natürlich ein visueller Satz gegeben sein, doch das ist gerade die Intention: Quasi als Nebeneffekt der Auswertung eines Simulationsausdrucks wird ein visueller Satz transformiert, die Anzahl und Reihenfolge der Regelanwendungen werden durch das iterative Programm definiert, das durch den Ausdruck gegeben ist.

Definition 2.4.2 *(Auswertung von Simulationsausdrücken) Seien $G = (A, S, R)$ eine Simulationsgrammatik, $X = (X_t)_{t\in DT_A}$ eine Familie von abzählbaren Variablenmengen, $(MS_n)_{n\in\mathbb{N}}$ eine Folge von Matchselektoren, $ass = (ass_t)_{t\in DT_A}$ mit $ass_t \colon X_t \to U_t$ eine Variablenbelegung und $H \in VS(A)$ ein visueller Satz. Definiere die Auswertungsfunktion $eval(G, (MS_n)_n, ass, H) \colon ESE(G, (MS_n)_n, ass, H) \to VS(A) \times \mathbb{B}$ für beliebiges $e \in SE(G, X)$ induktiv über $SE(G, X)$.*
$eval(G, (MS_n)_n, ass, H)(e) := (\tilde{H}, \tilde{b})$, wobei $\tilde{H}$ und $\tilde{b}$ wie folgt definiert werden:

- *$e =$ **true**:*
  Definiere $\tilde{H} := H$ und $\tilde{b} := true$.

- *$e =$ **false**:*
  Definiere $\tilde{H} := H$ und $\tilde{b} := false$.

- $e = \textbf{\textit{rule}}(r, pa)$:
  $pa' = (pa'_t)_{t \in DT_A}$ *mit* $pa'_t \colon P_r \to U_t$ *sei gegeben durch:*
  $$pa'_t(p) := \begin{cases} pa_t(p), & \textit{falls } pa_t(p) \in U_t \\ ass_t(pa_t(p)), & \textit{falls } pa_t(p) \in X_t \end{cases}.$$
  Ist $pa' \in dom(MS_{0,r,H})$, *so definiere* $\tilde{H} := T(r, pa', H, MS_{0,r,H}(pa'))$ *und* $\tilde{b} := \text{true}$, *sonst* $\tilde{H} := H$ *und* $\tilde{b} := \text{false}$.

- $e = (f \ \textbf{\textit{and}} \ g)$:
  Seien $(H', b) := eval(G, (MS_{2n})_n, ass, H)(f)$ *und* $(H'', b') := eval(G, (MS_{2n+1})_n, ass, H')(g)$. *Definiere* $\tilde{H} := H''$ *und* $\tilde{b} := b \wedge b'$.

- $e = (f \ \textbf{\textit{andc}} \ g)$:
  Sei $(H', b) := eval(G, (MS_{2n})_n, ass, H)(f)$. *Ist* $b = \text{false}$, *so definiere* $\tilde{H} := H'$ *und* $\tilde{b} := b$, *sonst* $(H'', b') := eval(G, (MS_{2n+1})_n, ass, H')(g)$ *sowie* $\tilde{H} := H''$ *und* $\tilde{b} := b \wedge b'$.

- $e = (f \ \textbf{\textit{or}} \ g)$:
  Seien $(H', b) := eval(G, (MS_{2n})_n, ass, H)(f)$ *und* $(H'', b') := eval(G, (MS_{2n+1})_n, ass, H')(g)$. *Definiere* $\tilde{H} := H''$ *und* $\tilde{b} := b \vee b'$.

- $e = (f \ \textbf{\textit{orc}} \ g)$:
  Sei $(H', b) := eval(G, (MS_{2n})_n, ass, H)(f)$. *Ist* $b = \text{true}$, *so definiere* $\tilde{H} := H'$ *und* $\tilde{b} := b$, *sonst* $(H'', b') := eval(G, (MS_{2n+1})_n, ass, H')(g)$ *sowie* $\tilde{H} := H''$ *und* $\tilde{b} := b \vee b'$.

- $e = \textbf{\textit{if}} \ f \ \textbf{\textit{then}} \ g \ \textbf{\textit{else}} \ h \ \textbf{\textit{fi}}$:
  Sei $(H', b) := eval(G, (MS_{3n})_n, ass, H)(f)$. *Ist* $b = \text{true}$, *so definiere* $(\tilde{H}, \tilde{b}) := eval(G, (MS_{3n+1})_n, ass, H')(g)$, *sonst* $(\tilde{H}, \tilde{b}) := eval(G, (MS_{3n+2})_n, ass, H')(h)$.

- $e = \textbf{\textit{while}} \ f \ \textbf{\textit{do}} \ g \ \textbf{\textit{done}}$:
  Sei $(H', b) := eval(G, (MS_{3n})_n, ass, H)(f)$. *Ist* $b = \text{true}$, *so definiere* $(H'', b') := eval(G, (MS_{3n+1})_n, ass, H')(g)$, $(\tilde{H}, b'') := eval(G, (MS_{3n+2})_n, ass, H'')(e)$ *und* $\tilde{b} = \text{true}$, *sonst* $\tilde{H} := H'$ *und* $\tilde{b} := \text{false}$.

$\square$

Die Definition der Auswertung der Simulationsausdrücke sollte bis auf die Verwendung der Folge der Matchselektoren wenig überraschend sein. Da es für die Auswertung eines Rule-Ausdrucks bei gegebener Regel r, visuellem Satz H und Parameterbelegung pa' notwendig ist, aus der Menge der Matches $M(r, pa, H)$ den zu verwendenden auszuwählen, wird ein Matchselektor benötigt. Würde man nun fordern, für alle Unterausdrücke eines Ausdrucks den selben Matchselektor zu verwenden, könnte dies eine zu starke Einschränkung darstellen. Deshalb wird eine Folge von Matchselektoren verwendet, die bei der Auswertung von Ausdrücken mit Unterausdrücken in indexdisjunkte Teilfolgen aufgespaltet wird, sodass für die Auswertung eines jeden Rule-Ausdruck in einem Simulationsausdruck ein anderes Element der Folge verwendet wird. In der Praxis heißt das, dass die Auswertung eines Simulationsausdrucks in Interaktion mit dem Benutzer geschehen kann, der in einer Situation, in dem mehrere Matches verwendbar wären, aufgefordert werden könnte, von diesen einen auszuwählen.

Schließlich sollen noch einige notationelle Verkürzungen definiert werden, die später bei der Beschreibung der Implementation (Abschnitt 5.6.2) ihre Anwendung finden: Seien $n \in \mathbb{N}$ beliebig und $e_i \in SE(G, X)$ für $i \in \mathbb{N}, 1 \leq i \leq n$, so bezeichnen $\mathbf{AND}_{i=0}^{n} e_i$, $\mathbf{ANDC}_{i=0}^{n} e_i$, $\mathbf{OR}_{i=0}^{n} e_i$, $\mathbf{ORC}_{i=0}^{n} e_i$ folgende Ausdrücke:

- $n = 0$:
  $\mathbf{AND}_{i=0}^{n} e_i = \mathbf{ANDC}_{i=0}^{n} e_i = \mathbf{true}$, $\mathbf{OR}_{i=0}^{n} e_i = \mathbf{ORC}_{i=0}^{n} e_i = \mathbf{false}$,
- $n = 1$:
  $\mathbf{AND}_{i=0}^{n} e_i = \mathbf{ANDC}_{i=0}^{n} e_i = \mathbf{OR}_{i=0}^{n} e_i = \mathbf{ORC}_{i=0}^{n} e_i = e_1$,
- $n \leq 2$:
  $\mathbf{AND}_{i=0}^{n} e_i = (\mathbf{AND}_{i=0}^{n-1} e_i \textbf{ and } e_n)$,
  $\mathbf{ANDC}_{i=0}^{n} e_i = (\mathbf{ANDC}_{i=0}^{n-1} e_i \textbf{ andc } e_n)$,
  $\mathbf{OR}_{i=0}^{n} e_i = (\mathbf{OR}_{i=0}^{n-1} e_i \textbf{ or } e_n)$,
  $\mathbf{ORC}_{i=0}^{n} e_i = (\mathbf{ORC}_{i=0}^{n-1} e_i \textbf{ orc } e_n)$.

2.4.2 Simulationsschritt

Sei $G = (A, S, R)$ eine Simulationsgrammatik, so ist ein Simulationsschritt $s = (X, e)$ gegeben durch eine Familie $X = (X_t)_{t \in DT_A}$ von abzählbaren Variablenmengen und einen Simulationsausdruck $e \in SE(G, X)$.

Zur Anwendung des Simulationsschritts auf einen visuellen Satz $H \in VS(A)$ müssen eine Variablenbelegung $ass = (ass_t)_{t \in DT_A}$ mit $ass_t \colon X_t \to U_t$ und eine Matchselektorfolge $(MS_n)_{n \in \mathbb{N}}$ gegeben sein und es muss $e \in ESE(G, (MS_n)_n, ass, H)$ gelten. Ergebnis der Anwendung des Simulationsschritts ist ein visueller Satz $S(s, ass, H, (MS_n)_n) := \tilde{H}$, wobei $(\tilde{H}, \tilde{b}) := eval(G, (MS_n)_n, ass, H)(e)$.

Beispiel 2.4.1 *Um die eher trockene Theorie der vorgestellten Simulationskonzepte zu illustrieren, folgen schließlich eine Simulationsgrammatik sowie drei darüber definierte Simulationsschritte zu den einfachen Statecharts. Der Einfachheit halber wird der Zustandsname durch ein optional angehängtes* `"_active"` *zum Indikator der Aktiviertheit des Zustands. Die Simulationsgrammatik (Abb. 2.11)* *besitzt zwei Regeln – eine, die einen Zustand als aktiv markiert, und eine, welche einen Zustand vor einer Transition deaktiviert und den nachfolgenden Zustand aktiviert. Beide benötigen eine Attributbedingung um sicherzustellen, dass der zu aktivierende Zustand nicht schon aktiv ist bzw. dass der Zustand vor der zu schaltenden Transition auch wirklich aktiv ist. Die in den rechten Regelseiten zu findenden Ausdrücke verwenden die Java-Notation. Die drei über den Regeln definierten Simulationsschritte (Abb. 2.12) sind ebenfalls sehr einfach: Der Schritt „start“ dient der Initialisierung der Simulation – mit Hilfe der ersten Regel wird ein Zustand aktiviert – der Schritt „step“ schaltet entlang einer Transition weiter und „run“ schaltet Transitionen so lange wie möglich – bei einem zyklischen Statechart würde die Simulation also nicht terminieren.*

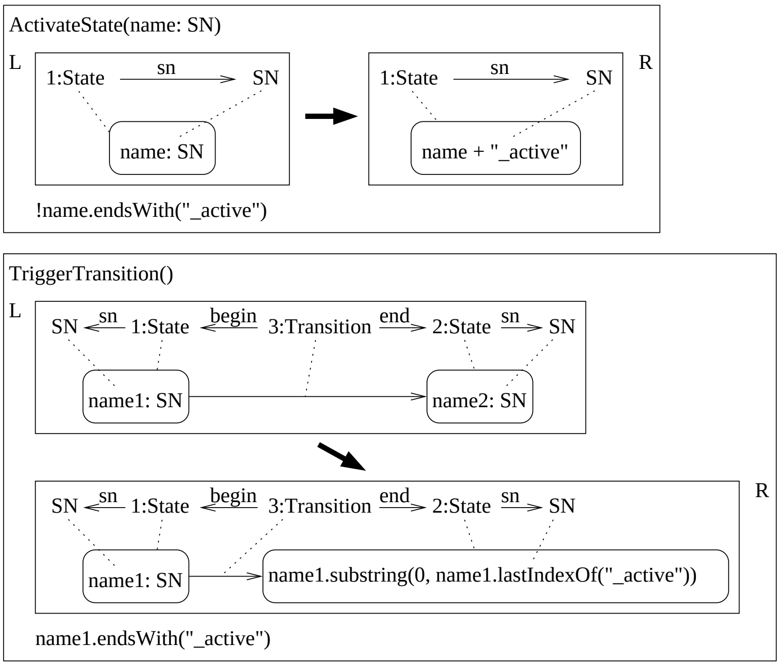

Abbildung 2.11: Regeln der Simulationsgrammatik für die einfachen Statecharts.

$$
\begin{array}{ll}
\underline{start} & = (name, startExpr) \\
startExpr & = \mathbf{rule}(ActivateState, \{name \mapsto name\}) \\
\underline{step} & = (\emptyset, stepExpr) \\
stepExpr & = \mathbf{rule}(TriggerTransition, \emptyset) \\
\underline{run} & = (\emptyset, runExpr) \\
runExpr & = \mathbf{while}\ \mathbf{rule}(TriggerTransition, \emptyset)\ \mathbf{do\ true\ done}
\end{array}
$$

Abbildung 2.12: Beispiele von Simulationsschritten für die einfachen Statecharts.

2.5 VL-Spezifikation, VL-Umgebung

Nachdem die Konzepte Syntaxgrammatik sowie Parsierungs- und Simulationsspezifikation eingeführt wurden, kann genauer besprochen werden, was die Aufgabe des GenGEd-Systems ist.

GenGEd soll von zwei Gruppen von Benutzern verwendet werden. Zum einen sind dies die Spezifikateure einer visuellen Sprache, zum anderen jene Anwender, die Diagramme einer visuellen Sprache zeichnen und simulieren wollen.

Spezifikateure möchten mit GenGEd eine visuelle Sprache definieren, um eine VL-Umgebung bereitzustellen, mit der dann die andere Benutzergruppe arbeiten kann. Wie der Arbeitsfluss eines VL-Spezifikateurs aussieht, zeigt Abbildung 2.13. Wie darin zu sehen ist, ist das Ziel seiner Arbeit eine VL-Spezifikation, die eine generische VL-Umgebung seinen Vorstellungen entsprechend konfiguriert.

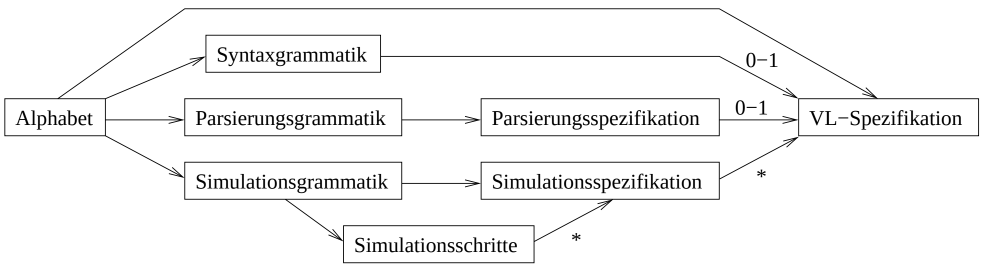

Abbildung 2.13: Arbeitsfluss bei der Definition einer VL-Spezifikation.

Im ersten Schritt erstellt der Spezifikateur ein visuelles Alphabet, es folgen Syntax-, Parsierungs- und Simulationsgrammatiken. Die Parsierungsgrammatik ist die Grundlage einer Parsierungsspezifikation. Optional kann eine Layerfunktion hinzugefügt werden und das Ergebnis einer Analyse der kritischen Paare. Desweiteren können Simulationsspezifikationen, jeweils bestehend aus einer Simulationsgrammatik und darüber definierten Simulationsschritten, definiert werden. Schließlich werden Alphabet, optional eine Syntaxgrammatik, ebenfalls optional eine Parsierungsspezifikation und beliebig viele Simulationsspezifikationen zu einer VL-Spezifikation zusammengefasst.

Eine VL-Umgebung bietet je nach VL-Spezifikation verschiedene Möglichkeiten. Ist eine Syntaxgrammatik gegeben, so soll dem Anwender das syntaxgesteuerte Editieren erlaubt werden, anderenfalls ist nur Freihandeditieren möglich. Enthält die VL-Spezifikation eine Parsierungspezifikation, können syntaxgesteuert oder freihand editierte visuelle Sätze auf Korrektheit geprüft werden. Mit den Simulationsschritten der Simulationsspezifikationen kann ein Diagramm schrittweise simuliert werden.

Kapitel 3

Ausgangssituation

Die Basis, die der Autor bei Beginn dieser Arbeit vorfand, bestand aus einer theoretischen Konzeption des GenGEd-Systems durch Roswitha Bardohl ([Bar00]) und den darauf fußenden Implementierungen eines Alphabet- und Diagrammeditors durch Manuel Schwarze ([Sch99]) und eines Grammatikeditors durch Magnus Niemann ([Nie99]) in der Programmiersprache Java. Zur Lösung der CSPs wurde der externe Constraintsolver ParCon ([Gri96]) über einen Daemon (ParCon-Daemon) angebunden. Die Transformation visueller Sätze war auf die vom Agg-System ([ERT99]) implementierte attributierte Graphtransformation zurückgeführt worden.

3.1 Java

Java ist eine von der Firma Sun Microsystems entwickelte interpretierte, objektorientierte Programmiersprache. Dadurch, dass es sich um eine interpretierte Sprache handelt, können Java-Programme auf jeder Plattform ausgeführt werden, die einen Interpreter, eine sogenannte Java Virtual Machine (JVM), zur Verfügung stellt. Mittlerweile gibt es JVM-Implementierungen für alle gängigen Betriebssysteme und sogar für verschiedene Hardware.

Neben einer JVM und Programmen zum Übersetzen und Archivieren gehört zu Java eine umfassende Klassenbibliothek. Ist auch diese vollständig für eine Plattform implementiert, so ermöglicht dies die Ausführung von Java-Programmen, die Zugriff auf Dateisystem- und Netzwerkfunktionalität haben und auch grafische Benutzeroberflächen (Graphical User Interfaces, GUIs) bereitstellen können.

Bevor ein Java-Programm ausgeführt werden kann, muss es vom Java-Compiler erst in einen Bytecode übersetzt werden, dessen Interpretation durch die JVM effizienter möglich ist. Spezielle JVMs, sogenannte JIT-Compiler (Just-In-Time-Compiler) übersetzen den Bytecode zur Laufzeit in schneller ausführbaren Plattform-nativen Code. Nichtsdestotrotz erreichen Java-Programme nicht annähernd die gleiche Geschwindigkeit wie Plattform-native Applikationen. Den Erfahrungen des Autors zufolge gilt dies insbesondere für GUI-Anwendungen, bei denen die Swing-Klassen verwendet werden.

Beim Abwägen der Vor- und Nachteile von Java stehen der Plattformunabhängigkeit und kurzen Entwicklungszeiten durch Merkmale wie Objektorientiertheit, einer umfangreichen Klassenbibliothek und Garbage-Collection hauptsächlich eine geringere Ausführungsgeschwindigkeit im Vergleich zu nativen Programmen und ein immenser Speicherhunger gegenüber. Für weniger zeitkritische Anwendungen, insbesondere, wenn sie plattformunabhängig sein sollen, ist Java also durchaus geeignet.

Die Entscheidung das GenGEd-System in Java zu implementieren, fiel im Wesentlichen aufgrund der Tatsachen, dass einerseits für den zeitkritischen Teil des Constraintsolvers der ohnehin externe ParCon angedacht war, und zum anderen, da zur Graphtransformation Agg verwendet werden sollte, welches vollständig in Java geschrieben ist.

Sowohl Alphabet- als auch Grammatikeditor wurden unter Verwendung der Java-Version 1.1 implementiert. Seitdem entwickelte Sun Microsystems Java kontinuierlich weiter, sodass die nun aktuelle Versionsnummer 1.3 lautet. Obwohl – soweit dem Autor bekannt – vollständige Rückwärtskompatibilität gewahrt wurde, stellt die Version 1.2 einen größeren Einschnitt dar. Seit dieser ist die offizielle Bezeichnung *Java 2 Platform*, statt wie bis dahin *Java Platform*. Mit der *Java 2 Platform* sind die Swing-Klassen in die Java-Klassenbibliothek aufgenommen worden und größere Teile des APIs (Application Programming Interfaces) wurden „aufgeräumt". Als positiver Effekt ist unter anderem das Verschwinden einiger Probleme mit GUIs bei der Portierung auf verschiedene Plattformen (insbesondere seit Java 1.3) zu bemerken.

3.2 Alphabeteditor

3.2.1 Funktion

Der Alphabeteditor dient dem Erstellen und Editieren von visuellen Alphabeten. Er besteht aus fünf Untereditoren – Symbol-, Verbindungs-, Constraint-, Diagrammeditor (eigentlich etwas unglücklich als Testsystem bezeichnet, soweit möglich verwenden wir, um spätere Verwirrung zu vermeiden, im Folgenden die Bezeichnung Diagrammeditor) und einem Hilfesystem – von denen Constrainteditor und Hilfesystem zwar vorgesehen, jedoch nicht implementiert sind und deshalb nicht weiter betrachtet werden.

Folgende Alphabetkonzepte (Abschnitt 2.1) wurden in Symbol- und Verbindungseditor implementiert:

- Symbole können definiert werden. Es stehen ein Reihe grafischer Primitva für die Definition der Symbolgrafik zur Verfügung. Deren Lagebeziehungen kann durch HLCs spezifiziert werden.

- Über einhundert HLCs sind in einer Datei vordefiniert; der Benutzer kann weitere hinzufügen.

- Datentypen können definiert werden. Dafür werden Java-Klassen benötigt, die ein spezielles Interface implementieren. Klassen für Strings, ganze und Fließkommazahlen und Stringlisten sind schon im Paket enthalten. Weitere kann der Benutzer selbst implementieren.

- Die Links zwischen Symbolen sowie die Attributierungslinks können definiert werden; es ist möglich, HLCs über den beteiligten Objekten anzugeben.

Im Diagrammeditor kann zu einem Alphabet ein visueller Satz definiert werden. Dies erfolgt durch sukzessives Einfügen von Symbol-, Datentyp- und Linkinstanzen. Die meisten Zwischenergebnisse bei einem solchen Editiervorgang sind keine visuellen Sätze. Nichtsdestotrotz können sie wie visuelle Sätze abgespeichert und später wieder geladen werden.

Symbol- und Verbindungseditor bearbeiten das selbe Alphabet. Auch hier sind Stadien möglich – etwa wenn ein Datentyp noch nicht durch einen Attributierungslink an ein Symbol gebunden wurde – die im Sinne der Konzepte kein korrektes visuelles Alphabet darstellen. Das Speichern und Laden ist auch hier möglich.

Die für Alphabete und visuelle Sätze verwendeten Dateiformate sind, wie später noch genauer beschrieben wird, nicht textueller Natur, insbesondere also nicht menschenlesbar.

Der Alphabeteditor bietet neben den genannten speziellen Fähigkeiten auch allgemeine, die in modernen GUIs nicht fehlen sollten. Dies ist zum einen ein Undo-Mechanismus, das heißt, die Möglichkeit, dass der Benutzer der Applikation Editierschritte rückgängig machen oder sie wiederholen kann, und zum anderen Internationalisierung, also Mehrsprachigkeit. Zurzeit werden die Sprachen Deutsch und Englisch unterstützt, doch das Hinzufügen weiterer Sprachen ist auch Nichtprogrammierern möglich, da es dazu nur notwendig ist, separate Textdateien zu erweitern.

3.2.2 Implementierung

3.2.2.1 Packagestruktur

Bevor näher auf spezielle Aspekte der Implementierung des Alphabeteditors eingegangen wird, soll ein kurzer Überblick über dessen wesentliche Packages gegeben werden.

`genged`: Das Wurzelpackage. Es enthält alle anderen Packages sowie die Klasse `GenGEd`, welche Methoden zur Verwaltung der Properties und Resourcen zur Verfügung stellt.

`genged.alphabeteditor`: Enthält die Implementierung des GUIs des Alphabeteditors sowie Basis- und Hilfsklassen für die Subeditoren.

`genged.alphabeteditor.symeditor`: Enthält die Implementierung des GUIs des Symboleditors.

`genged.alphabeteditor.coneditor`: Enthält die Implementierung des GUIs des Verbindungseditors.

`genged.alphabeteditor.testsystem`: Enthält die Implementierung des GUIs des Diagrammeditors.

`genged.constsolver`: Enthält alle Klassen, die mit der Implementierung der HLC-Sprache und der Anbindung an den Constraintsolver ParCon zu tun haben.

`genged.gui`: Enthält GUI-Klassen, die in mehreren Packages benötigt werden.

`genged.language`: Enthält die Implementierung des Alphabets sowie all seiner Komponenten (von den Constraints abgesehen), wie grafischer Primitiva, Symbole, Links und Datentypen.

`genged.language.datatypes`: Enthält die Implementierung der speziellen Datentypen (Strings, ganze und Fließkommazahlen und Stringlisten).

`genged.utilities`: Enthält diverse Klassen, die Funktionalitäten von allgemeinem Nutzen zur Verfügung stellen, etwa für die Mehrsprachigkeit der Applikation, für das Undo-System und Ähnlichem.

Nicht erwähnt wurden die `event` Unterpackages in den Packages `genged.language` und `genged.gui`, welche Klassen beinhalten, die das Event-Listener-Designpattern für das darüberliegende Package implementieren, sowie diverse `commands` Unterpackages, die für das jeweilige Superpackage Kapselungen von Editierfunktionen zum Zwecke der Unterstützung eines Undo-Mechanismus zur Verfügung stellen.

3.2.2.2 Alphabet, visueller Satz

Das Alphabet ist recht nahe an den Konzepten implementiert. Die Klasse `Alphabet` repräsentiert ein visuelles Alphabet. Für Symbole und Datentypen sind ebenfalls eigene Klassen, `SymbolObject` und `DatatypeObject`, vorgesehen, die beide das Interface `AlphabetObject` implementieren, welches eine Reihe von Methoden sammelt, die für die Verwaltung dieser Objekte von Relevanz sind. Symbole und Datentypen werden in der `Alphabet`-Klasse als Vector von `AlphabetObject`-Objekten verwaltet.

Ein Link wird von einem Objekt der Klasse `Operation` repräsentiert, welches Referenzen auf zwei `AlphabetObject`-Objekte, Quelle und Ziel des Links, besitzt. Es wird also nicht zwischen Attributierungslinks und Links zwischen lexikalischen Symbolen unterschieden. Ein `Operation`-Objekt hält desweiteren einen Vector von `ConstraintObject`-Objekten, die zum Link gehörenden Constraints.

Die Constraints eines Symbols sind nicht mit dem entsprechenden `SymbolObject`-Objekt assoziiert, sondern werden in einem `Operation`-Objekt gespeichert. In einem `Alphabet` gibt es deshalb zwei Typen von `Operation`-Objekten, externe und interne – solche, die für Links des Alphabets stehen, und die zuvor genannten.

Für jeden Typ der grafischen Primitiva existiert eine eigene Klasse, direkt oder indirekt abgeleitet von `GraphicObject`. Ein Gruppenprimitivum gibt es ebenfalls, implementiert durch `GroupObject`. `SymbolObject` leitet sich von dieser Klasse ab und `DatatypeObject` ist ein indirekter Abkömmling von `GraphicObject`. Die Klasse `GraphicObject` implementiert die im Abschnitt 2.1.1 angesprochenen universellen Eigenschaften wie Position und Größe und zwingt abgeleitete Klassen, eine Zeichenmethode (`paint()`) zu implementieren. Durch einen Aufruf dieser kann somit ein beliebiges Primitivum, ein Symbol oder Datatyp grafisch dargestellt werden.

Die Klassen `Alphabet` stellt drei umfassende Listener-Mechanismen zur Verfügung: Durch Implementierung der Interfaces `AlphabetListener`, `AlphabetAlphabetListener`, `AlphabetOperationListener` und Registrierung beim `Alphabet`-Objekt erfolgt eine Benachrichtigung bei Änderungen von Daten des Alphabets selbst (nicht seiner Unterobjekte), seiner Symbole, Datentypen und grafischer Primitiva oder den Links.

Ein visueller Satz zu einem Alphabet wird durch die Klasse `DiagramAlgebra` repräsentiert. Sie besitzt eine Referenz auf das Alphabet, über das der Satz getypt ist und leitet sich direkt von `Alphabet` ab, um dessen Funktionalität zur Verwaltung der Symbole, Datentypen und Links zu erben. Entsprechend werden auch die selben Klassen für die Instanzen der Elemente eines visuellen Satzes verwendet. Die Basisklasse `GraphicObject` besitzt ein Feld zur Unterscheidung, ob es sich um ein Objekt, das zu einem Alphabet gehört, oder um eine Instanz eines solchen handelt.

3.2.2.3 Persistenz

Das Speichern und Laden der Daten von Alphabeten und visuellen Sätzen wird dadurch ermöglicht, dass `Alphabet`, `DiagramAlgebra` sowie alle weiteren beteiligten Klassen das von Java vorgesehene `Externalizable`-Interface implementieren – sie können dann einfach mittels der Methoden `ObjectOutputStream.writeObject()` und `ObjectInputStream.readObject()` geschrieben bzw. gelesen werden.

Die Einfachheit dieses Verfahrens hat seinen Preis: Zum einen sind die so erzeugten Dateien nicht menschenlesbar, zum anderen ergeben sich Probleme, wenn eine Klasse um Felder erweitert wird, aber dennoch ältere Daten gelesen werden sollen.

3.2.2.4 Subeditoren

Die Untereditoren des Alphabeteditors implementieren das GUI für die Manipulation der `Alphabet`- und `DiagramAlgebra`-Instanzen. Sie verwenden ein gemeinsames Rahmenwerk (`genged.alphabeteditor`) bestehend aus den Klassen/Interfaces `AbstractEditor`, `EditorManager`, `EditorAction` und `ActionManager`. Vereinfacht formuliert, können Klassen, die das `EditorManager`-Interface implementieren, Subeditoren verwalten, die von `AbstractEditor` abgeleitet sind. Und tatsächlich implementiert `AlphabetEditor` `EditorManager`, während sich `SymbolEditor`, `ConnectionEditor` und `TestSystem` von `AbstractEditor` ableiten. Ein `AbstractEditor` muss eine Menü- und eine Werkzeugleiste sowie einen `JTree` und eine weitere Komponente mit beliebiger Funktionalität zur Verfügung stellen, `AlphabetEditor` realisiert die Fähigkeit zwischen den Untereditoren hin- und herzuschalten. Das `JTree`-Objekt stellt in den drei Subeditoren eine Strukturansicht eines Aspekts des Alphabets bzw. des visuellen Satzes dar, die weitere Komponente beinhaltet jeweils eine spezifische Werkzeugleiste und eine Zeichenfläche, auf der das Alphabet, der visuelle Satz oder Teile dieser dargestellt werden.

Die Klasse `EditorAction` stellt eine Erweiterung der Java-Klasse `AbstractAction`, eine partielle Implementation des Interfaces `Action`, welches zusammen mit der Klasse `ActionListener` das Listener-Designpattern implementiert, dar. Jede mögliche vom Benutzer initiierbare Aktion wird durch genau eine `EditorAction`-Instanz repräsentiert. Dieser sind alle GUI-Komponenten, wie Menüpunkte, Buttons einer Werkzeugleiste oder ähnlicher Elemente, bekannt, die die Aktion auslösen können. Insbesondere lässt sich durch `EditorAction`-Methoden steuern, ob die Aktion für den Benutzer verfügbar sein soll, bestimmte Eigenschaften können geändert werden und es ist möglich, die Aktion programmatisch auszulösen. Beim Auslösen der Aktion wird eine Methode eines Objekts aufgerufen – welche Methode welches Objekts ist für jede `EditorAction`-Instanz individuell setzbar. Zum Zeitpunkt des Programmstarts werden die `EditorAction`-Objekte aus Resourcedateien generiert, die für jeden Subeditor sämtliche dafür notwendigen Parameter enthalten.

Die noch nicht weiter erwähnte Klasse `ActionManager` dient der Verwaltung der `EditorAction`-Instanzen, sie ordnet eindeutigen Aktionsnamen das entsprechende Objekt zu.

3.2.2.5 Undo-Mechanismus

Ein von Java vorgesehener Undo-Mechanismus wird vom Alphabeteditor in erweiterter Form verwendet. Das Prinzip ist recht einfach: Jeder Editierschritt wird in einer Klasse kodiert, die das `UndoableEdit`-Interface implementiert. In einem Objekt einer solchen Klasse werden vor der Ausführung des Editierschritts alle für sein Rückgängigmachen benötigten Daten gesammelt, sodass auf Anfrage der ursprüngliche Zustand wieder hergestellt werden kann. Eine Instanz der Klasse `UndoManager` sammelt die `UndoableEdit`-Objekte und ermöglicht so das schrittweise Rückgängigmachen oder erneute Ausführen.

Für den Alphabeteditor wurde zum einen `UndoableEdit` durch `Command` implementiert, welches eine Reihe von Funktionalität zur Verfügung stellt, die ein schnelles Kreiieren von abgeleiteten Klassen für spezielle Editieraufgaben ermöglicht. Zum anderen wurde der `UndoManager` zum `GenGEdUndoManager` erweitert, der noch etwas komfortabler handhabbar ist und bei dem `EditorAction`-Objekte für „Undo“ und „Redo“ registriert werden können, die automatisch angepasst werden, wenn ein neuer Editierschritt hinzugekommen ist oder ein alter rückgängig gemacht oder erneut ausgeführt wurde.

3.2.2.6 Internationalisierung

Der Alphabeteditor wurde entworfen, um mehrere Sprachen zu unterstützen. Die zentrale dafür verantwortliche Klasse ist `LanguageText`. Ein Aufruf der Methode `getText()` liefert zu einem Schlüssel einen Text in der aktuell eingestellten Sprache zurück. Zudem können noch String-Parameter übergeben werden, die, wenn der Text dies vorsieht, an markierten Stellen in diesen eingefügt werden. Seine Daten liest ein `LanguageText`-Objekt aus Resourcedateien, die aus einer Liste von Einträgen aus Schlüssel und je einem Text für jede Sprache aufgebaut sind.

Es ist möglich, die Sprache des Alphabeteditors während einer Sitzung zu ändern. `LanguageText` leitet sich deshalb von `Observable` ab – beliebig viele Objekte von Klassen, die das `Observer`-Interface implementieren, können sich registrieren und werden bei einer Änderung der Sprache unverzüglich informiert. Im GUI des Alphabeteditors wird dies nahezu konsequent angewandt.

Der Genauigkeit wegen sei noch angemerkt, dass `LanguageText` meistens nicht direkt angesprochen wird: Statt `LanguageText.getText()` wird `GenGEd.getString()` verwendet und das Anmelden, um bei einer Sprachänderung informiert zu werden, erfolgt mittels `GenGEd.addLanguageChangeListener()`. Die letzteren Methoden setzen allerdings direkt auf jenen von `LanguageText` auf.

3.2.2.7 Probleme, Kritik

Obwohl der Alphabeteditor weitestgehend geschickt entworfen wurde, lassen sich dennoch einige Unschönheiten entdecken:

- Die Tatsache, dass eine Datentypklasse das Interface `Datatype` implementieren muss, ist sehr einschränkend und verhindert eine einfache Nutzung schon vorhandener, etwa der Java-Klassen. Sinnvoll wäre eine Trennung der eigentlichen Daten von der Funktionalität zu deren grafischer Darstellung.

- Dass `DiagramAlgebra` direkt von `Alphabet` abgeleitet ist, spart zwar augenscheinlich Arbeit, entspricht jedoch nicht den Konzepten. Eine gemeinsame Basisklasse, in der die Verwaltung der Elemente sowie Teile des Listener-Mechanismus implementiert sind, böte eine Alternative. Bei konsequenter Betrachtung müsste man auch Symbole, Datentypen und Links von deren Instanzen trennen. Eine generische Instanzklasse, die (etwa in einer Hashtable) nur die Parameter sammelt, in denen sich die Instanz vom Element des Alphabets unterscheidet, könnte eine adäquate Lösung sein.

- Die Constraints der Symbole in die sogenannten „internen Operationen“ zu verlegen, hinterlässt einen bitteren Nachgeschmack. Mit Rücksicht auf die Erweiterbarkeit, ist es geschickter, zusammengehörige Constraints in einer Klasse `ConstraintNetwork` zusammenzufassen. Sowohl Links als auch Symbole besitzen dann ein `ConstraintNetwork`-Objekt und registrieren es, wenn sie zu einem Alphabet oder Diagramm gehören, bei diesem. Legt man die Klasse hierarchisch an, so bestünde das Constraint Network eines Alphabets oder Diagramms aus den Constraint Networks der Symbole und Links sowie möglicherweise weiteren Constraints.

3.3 Grammatik- und VL-Editor

3.3.1 Funktion

GramEd (so sei die Gesamtimplementierung von Magnus Niemann nachfolgend bezeichnet) besteht aus drei separaten Applikationen: Grammatik-, Testdiagramm- und Diagrammeditor. Der Grammatikeditor dient dem Erstellen und Editieren von visuellen Grammatiken und der Testdiagrammeditor (um Verwirrung vorzubeugen, wird er im Folgenden als VL-Editor bezeichnet) dem Erstellen und syntaxgesteuerten Editieren von VL-Diagrammen. Der Diagrammeditor stellt eine nicht generische Version des VL-Editors für ein Statechart-Beispiel dar. Er ist hier nicht von Relevanz und wird deshalb nicht weiter betrachtet.

Die folgenden Grammatikkonzepte (Abschnitt 2.2) wurden implementiert:

- Zu einem Alphabet kann eine Grammatik generiert werden, die aus den Alphabetregeln und dem leeren visuellen Satz besteht.

- Es können neue leere Regeln und NACs hinzugefügt und (wie auch das Startdiagramm) mit den Alphabetregeln und den neuen schon fertig gestellten Regeln syntaxgesteuert editiert werden. Die Morphismenmappings können definiert werden, ebenso wie Regelparameter.

- Mit Hilfe einer visuellen Grammatik können im VL-Editor Diagramme syntaxgesteuert editiert werden.

VL-Diagramme und Grammatiken werden in Binärdateien abgespeichert. Ein einfacher Undo-Mechanismus ist vorgesehen (es kann nur ein Transformationsschritt rückgängig gemacht werden), jedoch keine Internationalisierung.

3.3.2 Implementierung

3.3.2.1 Packagestruktur

`gramed`: Das Wurzelpackage. Es enthält alle anderen Packages sowie die Klasse `GramEd`, welche Methoden zur Verwaltung der Properties und Resourcen zur Verfügung stellt.

`gramed.display`: Enthält die Implementierung der GUI-Elemente für die Visualisierung von Diagrammen und Regeln.

`gramed.gui`: Enthält die Implementierung des GUIs des Grammatikeditors.

`gramed.interfaces`: Enthält einige Interfaces.

`gramed.objects`: Enthält Klassen für die Teilstrukturen der Grammatik, nicht jedoch die Grammatikklasse selbst.

`gramed.seditor`: Enthält die Implementierung des GUIs des VL-Editors.

`gramed.grammar`: Enthält die Klasse für die Grammatik und ein Klasse für die Erzeugung der Alphabetregeln.

`gramed.util`: Enthält eine Reihe von Hilfsklassen.

Die Packages `gramed.gui` und `gramed.seditor` enthalten jeweils ein Unterpackage `actions`, in welchen sich Klassen befinden, die die Benutzeraktionen kapseln.

3.3.2.2 Grammatik, Transformation

Wie schon erwähnt, wird die Transformation der visuellen Sätze auf die Graphtransformation des AGG-Systems abgebildet. Die Grammatikstrukturen enthalten deshalb jeweils eine entsprechende AGG-Klasse. `GEDiagram` repräsentiert ein VL-Diagram und besitzt eine Referenz auf ein `DiagramAlgebra`-Objekt (siehe Abschitt 3.2.2.2), welches das eigentliche Diagramm enthält, eine Instanz der AGG-Klasse `Graph` zur Darstellung der abstrakten Syntax des Diagramms sowie Tabellen, um Symbolen Knoten und Links Kanten zuzuordnen und umgekehrt. `GEMorphism` besteht aus zwei `GEDiagram`-Referenzen für Quelle und Ziel des Morphismus sowie einem `OrdinaryMorphism`-Objekt als Repräsentation des Graphmorphismus zwischen Quell- und Zielgraphen. `GERule` wiederum setzt sich aus `GEMorphism`-Objekten (für Regelmorphismus und NACs) und einer Instanz der AGG-Klasse `Rule` (einer Graphregel) zusammen. `VLGrammar`, die Repräsentation einer VL-Grammatik, besitzt eine Menge von Regeln (`GERule`) und ein Startdiagramm (`GEDiagram`).

Die Transformation eines als `GEDiagram` gegebenen Diagramms erfolgt durch das AGG-System – alle dafür notwendigen Strukturen liegen in den `GE*`-Klassen vor. Das `Graph`-Objekt wird dabei verändert, es muss nur noch die konkrete Syntax, also das `DiagramAlgebra`-Objekt angepasst werden. Wie dies zu geschehen hat, wurde in Abschnitt 2.2.2 erläutert.

3.3.2.3 Persistenz

Alle zur Grammatik gehörenden Klassen implementieren, wie es auch beim Alphabet der Fall ist (Abschnitt 3.2.2.3), das `Externalizable`-Interface, ebenso wie die jeweils eingebetteten AGG-Klassen. Das Abspeichern und Laden erfolgt analog.

3.3.2.4 Editoren

Grammatik- und VL-Editor sind sich von der Funktionalität her sehr ähnlich. In beiden können syntaxgesteuert, durch Regelanwendung, Diagramme bearbeitet werden. Dem Rechnung tragend leiten sich beide Klassen, `GrammarEditor` und `TestDiagramEditor`, von der gemeinsamen Basisklasse `AbstractVLEditor` ab. Benutzeraktionen sind nicht wie im Alphabeteditor durch eine generische Klasse realisiert, deren Parameter aus separaten Resourcedateien gelesen werden, sondern es existiert für jede Aktion eine eigene Klasse in `gramed.gui.actions` und `gramed.seditor.actions`.

3.3.2.5 Undo-Mechanismus

Der Undo-Mechanismus sieht nur vor, einen Schritt rückgängig machen zu können, genauer die letzte Diagrammtransformation. Ein `GEDiagram`-Objekt speichert vor einer Transformation seinen aktuellen Zustand in dafür vorgesehene Felder. Das Ausführen des Undo-Schritts stellt diesen dann wieder her.

3.3.2.6 Probleme, Kritik

Die meisten der im Folgenden aufgelisteten Punkte resultieren vermutlich direkt oder indirekt aus der Tatsache, dass Alphabeteditor und GramEd zur gleichen Zeit entwickelt wurden:

- Das GenGEd-System sollte ein integriertes Werkzeug sein, jedoch sind Grammatik- und VL-Editor separate Applikationen.
- Die Action-Mechanismen von Alphabeteditor und GramEd (Abschnitte 3.2.2.4 und 3.3.2.4) sind verschieden.
- Die Undo-Mechanismen von Alphabeteditor und GramEd (Abschnitte 3.2.2.5 und 3.3.2.5) sind verschieden. Insbesondere bezüglich seiner Fähigkeiten ist letzterer nur rudimentär.
- GramEd ist nicht internationalisiert.
- Die Strukturen zur Repräsentation der Grammatik sind mit Agg-Strukturen verwoben. Dies ist von daher ungeschickt, dass Agg eigentlich nur zur Transformation benötigt wird und deshalb nur über eine abstrahierende Schnittstelle angesprochen werden sollte. Dies ermöglichte zum Beispiel das problemlose Ersetzen von Agg durch ein anderes Transformationssystem, ohne hierbei in die Datenstrukturen oder deren Verwendung eingreifen zu müssen.

3.4 ParCon und ParCon-Daemon

ParCon ist ein 1995 von Peer Griebel an der Universität-GH Paderborn entwickeltes, in der Programmiersprache C geschriebenes Kommandozeilenprogramm zum Lösen von Constraint Satisfaction Problemen, das heißt ein Constraintsolver. Das System ist insbesondere dafür ausgelegt, zweidimensionale grafische Constraints verarbeiten zu können. So ist es zum Beispiel möglich, in den in einer speziellen Syntax anzugebenden Low Level Constraints direkt Punktvariablen zu verwenden.

Nachdem alle zum Constraint Network gehörenden Constraints und eine initiale Variablenbelegung an ParCon übergeben sind, löst dieser das CSP (wenn möglich) und gibt die neuen Werte der Variablen zurück. Anschließend können einzelne oder alle Variablen neu belegt werden, woraufhin ParCon das CSP erneut löst. Dabei wird eine „Least Astonishment"-Strategie angewandt, das heißt, es wird versucht, die neuesten Variablenwerte nicht zu verändern, sondern mit den ältesten zu beginnen. Da es häufig Constraints unterschiedlicher Wichtigkeit gibt, ist eine Zuordnung von Prioritäten zu den an ParCon übermittelten Constraints vorgesehen.

Abgesehen davon, dass es sich um ein Kommandozeilenprogramm handelt, was für die Anbindung an eine Java-Applikation nicht hilfreich ist, besitzt ParCon alle Eigenschaften, die es für das Lösen der CSPs in GenGEd geeignet machen. Das Anbindungsproblem wurde 1997 durch die Implementation eines Daemons gelöst. Die Aufgabe des Daemons ist es, beliebig vielen Clients eine Kommunikation mit ParCon über ein auf TCP/IP aufsetzendes Protokoll zu ermöglichen. Eine Beschreibung dieses Protokolls ist in [Sch99] nachzulesen. Da die Java-Bibliothek auch Klassen für den Datentransfer per TCP/IP bereitstellt, ist die Verwendung beliebig vieler paralleler ParCon-Prozesse zur Lösung der in GenGEd anfallenden CSPs möglich.

3.5 Agg

Agg wurde 1991 an der Technischen Universität Berlin als Editor für Graphen in der Programmiersprache Eiffel entwickelt ([LB93]), 1997 in Java reimplementiert ([Rud97, Mel98, ERT99]) und seitdem stetig erweitert. Es ermöglicht die Definition von attributierten Graphen und Graphregeln und implementiert verschiedene Strategien zur Matchsuche und Transformation. Als Attributtypen der Knoten und Kanten sind beliebige Java-Klassen erlaubt, für die Attributwerte dürfen sogar erweiterte Java-Ausdrücke (in Regeln auch mit Variablen) verwendet werden, die das System auszuwerten vermag. NACs können ebenso definiert werden wie Attributbedingungen.

Zum Zeitpunkt der Entwicklung des Grammatikeditors war Agg, besonders die Attributkomponente, leider noch recht fehlerbehaftet. Bis zu dieser Arbeit konnte jedoch nicht nur die Stabilität und Zuverlässigkeit stark verbessert werden, es sind auch neue Komponenten, wie zum Beispiel ein Parser, hinzugekommen.

Im Zuge der Weiterentwicklung wurden auch einige Änderungen am API und den Vorbedingungen für die Transformation vorgenommen. So muss ein zu transformierender Graph nun vollständig attributiert sein, während dies früher nicht nötig war und in GenGEd auch nicht sichergestellt wurde.

Kapitel 4

Anforderungen

Dieses Kapitel konkretisiert die mit der Aufgabenstellung dieser Arbeit verbundenen Anforderungen. Die in Kapitel 3 wiedergegebene Analyse des bestehenden GenGEd-Systems lag zu Beginn der Arbeit selbstverständlich noch nicht in diesem Umfang vor – einen tieferen Einblick konnte der Autor erst nach intensiver Einarbeitung in den Quellcode und während der Implementierung gewinnen. Die Formulierung der Anforderungen orientierte sich deshalb primär an der Funktionalität, nicht der Implementierung des vorhandenen GenGEd-Systems. Auf welche Weise die Anforderungen angesichts der Ausgangssituation umgesetzt wurden, beschreibt das Kapitel 5.

4.1 Integration von Alphabet-, Grammatik- und VL-Editor

Wie in Kapitel 3 beschrieben, ist der Alphabeteditor ein monolithisches System bestehend aus Subeditoren, während Grammatik- und VL-Editor separate Applikationen sind. Im Sinne einer integrierten Umgebung zur Spezifikation einer visuellen Sprache sind deshalb alle Teileditoren in einer Applikation zusammenzufassen.

Im Rahmen dieser Aufgabe müssen eine Reihe von Teilaspekten bearbeitet werden. So sollte eine gemeinsame Packagestruktur gefunden werden, in der die Packages `genged` und `gramed` aufgehen, Grammatik- und VL-Editor müssen sich von `AbstractEditor` ableiten, um sie zu Untereditoren des Alphabeteditors zu machen, das GUI von Alphabet- und Grammatik-/VL-Editor muss angeglichen werden. Außerdem sollte die im Alphabeteditor implementierte Internationalisierung auf den GramEd-Teil übertragen werden, ebenso wie der Undo-Mechanismus. Es ist des Weiteren sinnvoll, den von GramEd genutzten Action-Mechanismus durch den generischen des Alphabeteditors zu ersetzen.

4.2 Überarbeitung von Grammatik- und VL-Editor

Wie fast jedes Softwaresystem enthält auch GenGEd Fehler. Eine Reihe von Fehlern, die seit der Implementation bekannt geworden sind, sollen behoben werden. Dies gilt insbesondere für Grammatik- und VL-Editor. Ein Teil dieser Aufgabe besteht darin, die ursprünglich verwendete Agg-Version durch die aktuellste zu ersetzen, um einerseits vom Korrigieren der Fehler in dieser und zum anderen von deren neuen Fähigkeiten zu profitieren.

Desweiteren bietet GramEd mehrere Ansatzpunkte, dessen Benutzerfreundlichkeit zu erhöhen, etwa durch neue Funktionen, wie dem Kopieren von Regeln und NACs, oder einer eleganteren Lösung für den Export von Regeln.

4.3 Überarbeitung der Schnittstelle zu Agg

Wie im Abschnitt 3.3.2.6 beschrieben, sind die Strukturen zur Repräsentation einer visuellen Grammatik mit Strukturen aus Agg verschränkt. Es sollen neue Datenstrukturen für Grammatiken entwickelt werden, die keine Agg-Strukturen mehr enthalten. Für die Transformation von visuellen Sätzen soll eine Schnittstelle konzipiert werden, die von der konkreten Implementierung der Transformation abstrahiert. Eine Implementation dieser Schnittstelle stellt dann die Anbindung der Agg-Graphtransformation dar.

Die Gründe für diese Kapselung wurden schon genannt: Es besteht die Möglichkeit andere Transformationstools einfach anzubinden oder gar eine neue, auf GenGEd -Strukturen zugeschnittene Transformation zu implementieren. Zudem sind dann bei etwaigen Änderungen im Agg-API nur sehr lokale Anpassungen in GenGEd erforderlich.

4.4 XML-basiertes Speichern und Laden

Das Speichern und Laden der Alphabete, Diagramme und Grammatiken erfolgt mittels des `Externalizable`-Protokolls von Java. Der Vorteil der Einfachheit der Verwendung dieses Mechanismus wird aufgewogen durch Inkompatibilitäten, die entstehen, wenn die betreffenden Strukturen eine Veränderung, Erweiterung oder gar Neuimplementierung (Abschnitt 4.3) erfahren. Desweiteren sind die gespeicherten Daten weder menschenlesbar noch als Austauschformat zwischen verschiedenen Applikationen geeignet.

Seit Anfang 2000 gibt es internationale Bestrebungen, ein auf XML basierendes Austauschformat für Graphen und Graphgrammatiken zu entwickeln. Zwar werden in GenGEd nicht wirklich Graphen, sondern am ehesten Graphstrukturen benutzt, doch sollte es nichtsdestotrotz möglich sein, ebenfalls dieses Format zu verwenden.

Die Bemühungen zur Entwicklung des Formats sind noch im Gange und es wird voraussichtlich noch einige Zeit dauern, bis die endgültige Spezifikation feststeht. Dennoch soll schon jetzt die Umstellung auf das Speichern in XML stattfinden. Als Basis ist dabei eine an der TU Berlin entwickelte DTD (Document Type Definition) zu verwenden ([GXL01]). Es kann davon ausgegangen werden, dass, sobald ein endgültiges Format feststeht, eine Anpassung an dieses keinen sonderlich großen Aufwand verursachen dürfte.

4.5 VL-Spezifikation, VL-Spezifikationseditor

Es sollen Datenstrukturen entwickelt werden, um eine VL-Spezifikation zu repräsentieren. Dazu gehören insbesondere Strukturen für Parsierungsspezifikationen und ihre Komponenten sowie für Simulationsspezifikationen, -schritte und -ausdrücke entsprechend der im Kapitel 2 beschriebenen Konzepte. Das Speichern der Datenstrukturen soll in XML-Dateien geschehen, deren genaues Format zu spezifizieren ist.

Zum Erstellen und Bearbeiten der VL-Spezifikationen wird ein Editor, der VL-Spezifikationseditor, benötigt, der als Subeditor in das GenGEd-System integriert werden soll. Er muss alle Mittel bereitstellen, um die in einer VL-Spezifikation enthaltenen Parsierungs- und Simulationsspezifikationen zu erstellen und zu modifizieren. Dies beinhaltet die Definition einer Layerfunktion, die Analyse der kritischen Paare und die Definition von Simulationsschritten.

4.6 VL-Umgebung

Es muss eine VL-Umgebung entwickelt werden, die die in Abschnitt 2.5 erläuterten Konzepte implementiert. Das heißt, sie soll sich nach dem Laden einer VL-Spezifikation automatisch entsprechend dieser konfigurieren, also Freihand- oder syntaxgesteuertes Editieren anbieten, optional Parsierung und Simulation.

Für das Freihandeditieren kann der Diagrammeditor und für das syntaxgesteuerte Editieren der VL-Editor angepasst werden. Die Parsierung, die bisher in keiner der GenGEd-Komponenten vorgesehen ist, kann auf die nunmehr in Agg integrierte Graphparsierung abgebildet werden. Die Simulationskomponente hingegen muss völlig neu entwickelt werden. Sie soll eine benutzergesteuerte Schrittsimulation unter Verwendung von Simulationsschritten zur Verfügung stellen.

Die VL-Umgebung soll einerseits als weiterer Untereditor – in dem der Spezifikateur seine gerade definierte VL-Spezifikation testen kann – andererseits als „Stand-Alone-System“ für den Endanwender verfügbar sein.

Kapitel 5

Entwurf und Implementierung

In diesem Kapitel wird die Leserin/der Leser über die Details der Umsetzung der in Kapitel 4 beschriebenen Arbeitspunkte informiert. Nach einleitendenen Abschnitten über vom Autor für den Entwurf und die Implementierung verwendete Hilfsmittel und die Struktur des neuen GenGEd-Systems bauen die folgenden Abschnitte systematisch in „Bottom-Up-Manier“ aufeinander auf. Es wird mit der Beschreibung der neu entwickelten Strukturen für die Grammatik begonnen, es folgen das Transformationsrahmenwerk, die Strukturen für die Parsierung, die Simulation und schließlich die VL-Spezifikation. In Abschnitt 5.8 werden die Mechanismen zum Speichern und Laden der zuvor vorgestellten Datenstrukturen präsentiert.

Nachdem damit das nicht grafische Fundament abgehandelt wurde, beschließt ein Abschnitt über GUI-Komponenten, genauer die veränderten und die neu entworfenen Editoren, dieses Kapitel.

Der Aufbau des Kapitels ist, wie erwähnt, systematisch, nicht chronologisch. So begann der Autor tatsächlich damit, die vorhandenen Editoren zu integrieren (entsprechend Abschnitt 4.1). Erst nachdem dies bis auf das Angleichen der internen Mechanismen abgeschlossen und mit einer genaueren Analyse der Datenstrukturen für die Implementation der XML-Ein- und -Ausgabe begonnen worden war, wurde deutlich, dass die an Grammatik- und VL-Editor durchzuführenden Änderungen so umfangreich sein würden, dass es sinnvoller erschien, sie völlig neu zu entwickeln. Die Beschreibung dieser und anderer Arbeiten, die letztlich keinen Eingang in das Ergebnis dieser Arbeit fanden, wird dieses Kapitel der Leserin/dem Leser vorenthalten. Vielmehr soll es möglich sein, hier effizient Informationen über die praktische Umsetzung der GenGEd-Konzepte nachzuschlagen.

5.1 Hilfsmittel

Zum Entwurf und zur Implementation wurden vom Autor verschiedene Werkzeuge eingesetzt:

UML: Die Unified Modelling Language ([UML01]) ist die heute am weitesten verbreitete Sprache zur Modellierung insbesondere von objektorientierten Systemen. Genau genommen ist sie eine Sammlung von verschiedenen Modellierungssprachen, wie Klassen-, Objekt- oder Sequenzdiagrammen, von denen in dieser Arbeit vorrangig Klassendiagramme wiederzufinden sind.

Designpatterns: Ein Designpattern (Entwurfsmuster) beschreibt einen Lösungsansatz für ein verallgemeinertes entwurfs- oder implementierungstechnisches Problem. In der Java-Bibliothek finden sich eine große Menge von Implementationen verschiedener Designpatterns. Auch diese Arbeit macht Gebrauch von ihnen, jedoch sollen die verwendeten hier nicht näher eingeführt werden. Es sei auf [GHJV95] verwiesen.

kfte: Alle Implementierungsarbeiten wurden unter Linux durchgeführt, zum Editieren des Quelltexts wurde der äußerst flexible Editor kfte verwendet.

CVS: Das Concurrent Versions System (CVS) ist ein Versionskontrollsystem, verfügbar auf Unix-Systemen und allen heute gängigen Plattformen. CVS kann allerdings nicht nur verschiedene Versionen von Dateien verwalten, sondern ermöglicht auch das parallele Arbeiten mehrerer Entwickler an den gleichen Dateien und unterstützt beliebig viele Entwicklungszweige (Branches).

5.2 Struktur des Systems

5.2.1 Verzeichnisstruktur

Die Verzeichnisstruktur des GenGEd-Systems wurde völlig neu gestaltet. Alle zum System gehörenden Dateien wurden unter CVS-Versionsverwaltung gestellt.

`genged`: Das Wurzelverzeichnis. Enthält sämtliche zum GenGEd-System gehörenden Dateien.

`genged/bin`: Enthält generierte ausführbare Dateien (den ParCon-Daemon), in den Unterverzeichnissen `Linux` und `SunOS` plattformspezifische ParCon-Executables, sowie die kompilierten Java-Klassen.

`genged/doc`: Enthält die Dokumentationen zu GenGEd, neben dieser Arbeit zum Beispiel die DTDs, die das Dateiformat beschreiben.

genged/examples: Enthält die Unterverzeichnisse `alphabets`, `diagrams`, `vlgrammars` und `vlspecs` mit Beispielen für Alphabete, Diagramme, Grammatiken und VL-Spezifikationen.

genged/lib: Enthält die von GenGEd verwendeten Drittbibliotheken, wie `agg.jar`, `colim.jar`, `jgl.jar` und `xerces.jar`.

genged/resources: Enthält die GenGEd-Resourcedateien. Neben Icons gehören dazu Dateien, in denen die HLCs und Actions definiert sind, sowie solche, die die Texte für die Internationalisierung beinthalten.

genged/src: Enthält die Unterverzeichnisse `doc`, `genged` und `parcond` mit den Quellen für Dokumentationen (zum Beispiel für diese Arbeit), für GenGEd selbst und für den ParCon-Daemon.

5.2.2 Packagestruktur

Die neue Packagestruktur basiert auf jener des alten Alphabeteditors. Einige Pakete wurden umbenannt und eine Reihe neuer sind hinzugekommen:

genged: Das Wurzelpackage. Es enthält alle anderen Packages sowie die Klasse `GenGEd`, welche Methoden zur Verwaltung der Properties und Resourcen zur Verfügung stellt.

genged.alphabet: Enthält die Implementierung des Alphabets sowie all seiner Komponenten (von den Constraints abgesehen), wie grafischer Primitiva, Symbole, Links und Datentypen.

genged.alphabet.datatypes: Enthält die Implementierung der speziellen Datentypen (Strings, ganze und Fließkommazahlen und Stringlisten).

genged.constsolver: Enthält alle Klassen, die mit der Implementierung der HLC-Sprache und der Anbindung an der Constraintsolver ParCon zu tun haben.

genged.gengededitor: Enthält die Implementierung des GUIs der GenGEd-Applikation sowie Basis- und Hilfsklassen für die Subeditoren (ehemals `genged.alphabeteditor`).

genged.gengededitor.symeditor: Enthält die Implementierung des GUIs des Symboleditors.

genged.gengededitor.coneditor: Enthält die Implementierung des GUIs des Verbindungseditors.

genged.gengededitor.diagrameditor: Enthält die Implementierung des GUIs des Diagrammeditors (ehemals `genged.alphabeteditor.testsystem`).

genged.grammar: Enthält die Implementierung der Grammatik sowie all ihrer Komponenten.

genged.gui: Enthält GUI-Klassen, die in mehreren Packages benötigt werden.

genged.parsing: Enthält die Implementierung der Parsierungsspezifikation und Interfaces für die Abstraktion der Parsierung und das Unterpackage `agg` mit einer auf der AGG-Graphparsierung basierenden Implementation.

genged.simulation: Enthält die Implementierung der Simulationsspezifikation, -schritte und -ausdrücke.

genged.transformation: Enthält Interfaces für die Abstraktion der Transformation und das Unterpackage `agg` mit einer auf der AGG-Graphtransformation basierenden Implementation.

genged.utilities: Enthält diverse Klassen, die Funktionalitäten von allgemeinem Nutzen zur Verfügung stellen, etwa für die Mehrsprachigkeit der Applikation, für das Undo-System und ähnlichem.

genged.vlspec: Enthält die Implementierung der VL-Spezifikation.

5.3 Grammatik

Die Datenstrukturen für die Repräsentation visueller Grammatiken wurden völlig neu entwickelt. Dies war aufgrund der Verschränkung der bisherigen Strukturen mit AGG-Klassen notwendig. Doch auch für Alphabet, visuelle Sätze und deren Elemente wurden neue Klassen geschrieben. Diese ersetzen allerdings nicht `Alphabet` und `DiagramAlgebra`, sondern „wrappen" sie nur, das heißt, verwenden sie intern, stellen jedoch eine andere Schnittstelle zur Verfügung, um geeigneter auf sie zuzugreifen. Die Gründe für dieses Unterfangen liegen hauptsächlich darin, Implementierungsdetails zu verstecken – etwa die Verwendung von temporären Instanzen im Alphabet oder die Mischung von Symbolen und Datentypen – und eine leichtere Handhabung des Rahmenwerks zu ermöglichen. So ist insbesondere das Layout, also die konkrete Syntax, für eine Grammatik nicht relevant.

Die neuen Klassen sind näher an der Theorie der algebraischen Signaturen und Algebren konzipiert, was auch den Unterschied zu den attributierten Graphen, wie sie für die Transformation mittels AGG verwendet werden, deutlicher hervortreten lässt. Einzig die Datentypen werden nicht theoriegemäß repräsentiert, sondern aufgrund ihrer speziellen Eigenschaften wie Attribute behandelt.

5.3.1 Signatur

Die Klasse `GGSignature` implementiert eine neue Sicht auf ein Alphabet. Ein Konstruktor, der ein `Alphabet`-Objekt als Parameter bekommt, kreiert ein

`GGSignature`-Objekt, das das selbe Alphabet als algebraische Signatur repräsentiert. Eine solche Signatur besteht aus Sorten und Operationssymbolen, die durch `GGSort`- und `GGOperationSymbol`-Instanzen verkörpert werden. Eine Sorte steht für ein Symbol und kann mit beliebig vielen Datentypen (`GGAttributeType`) assoziiert sein – Attributierungslinks werden durch das Rahmenwerk nicht explizit dargestellt. Links zwischen Symbolen finden ihre Entsprechungen in den `GGOperationSymbol`-Objekten, die den Konzepten in Abschnitt 2.1.4 folgend eine Quell- und eine Zielsorte referenzieren, also unär sind. Die Beziehung der neuen Klassen zu den alten zeigt Abbildung 5.1.

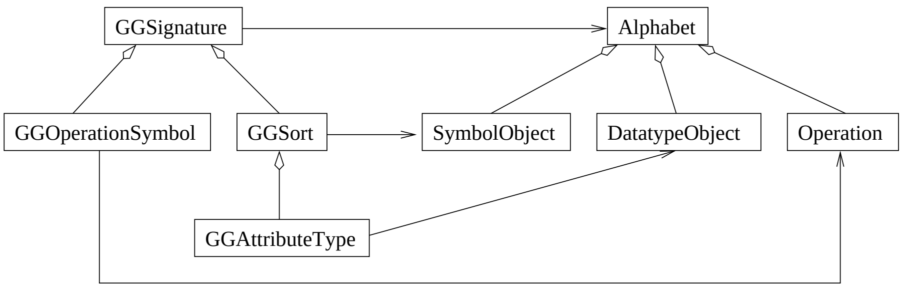

Abbildung 5.1: Beziehungen zwischen `GGSignature` und `Alphabet` und assozierten Klassen.

Im Folgenden sollen einige Methoden dieser Klassen vorgestellt werden. Gemein ist allen öffentlichen Methoden, dass sie keine Veränderungen verursachen, die Objekte also gewissermaßen „read only“ sind. Dies macht Sinn, da an den Stellen, wo das Rahmenwerk eingesetzt wird, ausschließlich visuelle Sätze, Regeln oder Grammatiken verändert werden, das Alphabet aber längst fixiert ist. Daraus resultiert der große Vorteil, dass während der Konstruktion und Initialisierung eines `GGSignature`-Objekts diverse Hilfsdaten angelegt werden können, die zwar redundante Informationen darstellen, aber einen effizienten Zugriff erlauben.

5.3.1.1 `GGSignature`

Die Klasse `GGSignature` besitzt neben privaten implementierungsinternen Methoden und jenen, die der Implementierung der `GGStorable`- und `XMLObject`-Interfaces dienen – auf diese wird im Abschnitt 5.8 näher eingegangen – und `getAlphabet()`, die das zur Signatur gehörende `Alphabet`-Objekt zurückliefert, ausschließlich solche, die Zugriff auf die Elemente der Signatur bieten. So liefern `getSorts()`, `getOperationSymbols()` und `getAttributeTypes()` in Form von `Collection`-Objekten die Mengen aller Sorten, Operationssymbole oder Attributtypen.

Einzelne Objekte lassen sich durch die Methoden `getSortFor()`, `getOperationSymbolFor()`, und `getAttributeTypeFor()` finden. Von diesen gibt es jeweils eine Version, die einen `String`, den Bezeichner des gesuchten

Elements, bekommt, sowie eine, deren Parameter das entsprechende Objekt aus dem `Alphabet` ist, also vom Typ `SymbolObject`, `Operation` respektive `DatatypeObject`.

Oftmals werden alle Operationssymbole gesucht, die eine bestimmte Quell- und/oder Zielsorte haben. `getOperationSymbolsWithSourceTarget()`, `getOperationSymbolsWithSource()` und `getOperationSymbolsWithTarget()` liefern die gewünschten Mengen als `Collection`.

Letztlich bleiben noch die Methode `getSortIndegree()` und `getSortOutdegree()` zu erwähnen. Da es sich bei den Signaturen um Graphstruktur-Signaturen handelt, man sie also als Graphen und damit die Sorten als Knoten betrachten kann, ist es teilweise hilfreich auf Graphalgorithmen zurückzugreifen. So kann zum Feststellen, ob die Signatur zyklenfrei ist, ein linearer Algorithmus zur topologischen Sortierung adaptiert werden. Dieser etwa benötigt den Eingangsverzweigungsgrad der Knoten, den `getSortIndegree()` liefert.

5.3.1.2 `GGSort`

`GGSort` besitzt diverse interne Methoden, doch für Benutzer des Frameworks sind wenige andere von Interesse. `getName()` liefert den Namen der Sorte, `getSymbolObject()` das korrespondierende `SymbolObject`-Objekt, `getSignature()` und `getAlphabet()` Signatur und Alphabet.

`getAttributeTypes()` und `getAttributeTypeFor()` haben, abgesehen davon, dass sie sich nur auf die Attributtypen der entsprechenden Sorte beziehen, die selbe Funktion wie die gleichnamigen Methoden der Signatur.

`getSourceOperationSymbols()` und `getTargetOperationSymbols()` geben die Operationssymbole zurück, die diese Sorte als Quelle oder Ziel haben.

5.3.1.3 `GGOperationSymbol`

Sieht man auch hier von den internen Operationen ab, bleiben noch `getSignature()`, `getAlphabet()`, die Signatur und Alphabet liefern, `getDiagramOperation()` das korrespondierende `Operation`-Objekt und `getName()` den Namen des Operationssymbols.

`getSource()` und `getTarget()` geben die Quell- bzw. Zielsorte des Operationssymbols zurück.

5.3.1.4 `GGAttributeType`

Auch `GGAttributeType` besitzt Methoden, um auf Alphabet, Signatur, das korrespondierende `DatatypeObject`-Objekt sowie den Namen des Attributtypen und die Sorte, zu dem er gehört, zuzugreifen. Zudem liefern `getDatatypeClassname()` den Namen der Datentypklasse und `createDefaultValue()` eine neu kreierte Instanz dieser zurück.

5.3.2 Algebra

Die neuen Klassen, die einen visuellen Satz verkörpern sollen, erfüllen zwei wesentliche Eigenschaften: Zum einen stellen sie einen Adapter für `DiagramAlgebra`-Objekte und ihre Elemente (Abb. 5.2) und zum anderen Instanzklassen der Signaturstrukturen (Abb. 5.3) dar. Ein visueller Satz wird als Algebra über einer Signatur verstanden und durch ein `GGAlgebra`-Objekt repräsentiert.

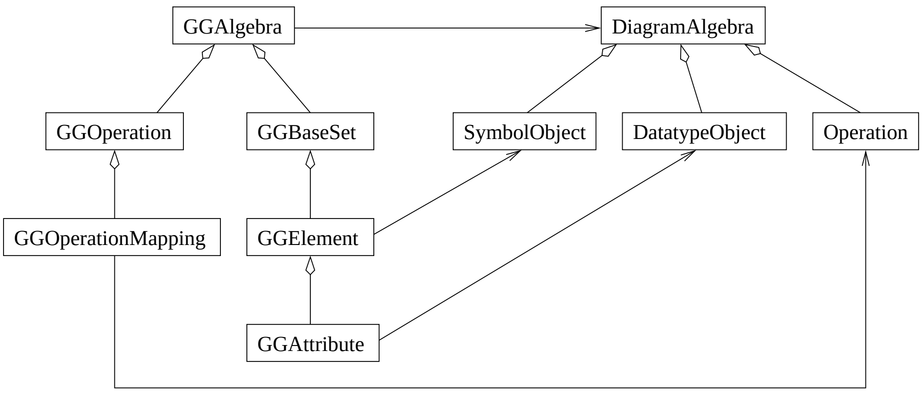

Abbildung 5.2: Beziehungen zwischen `GGAlgebra` und `DiagramAlgebra` und assozierten Klassen.

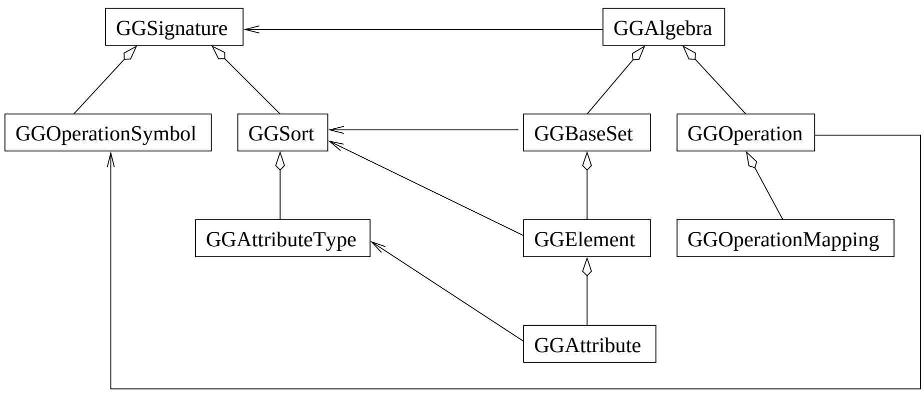

Abbildung 5.3: Beziehungen zwischen `GGAlgebra` und `GGSignature` und assozierten Klassen.

Eine Algebra besitzt zu jeder Sorte der Signatur eine Trägermenge (`GGBaseSet`) und zu jedem Operationssymbol eine Operation (`GGOperation`), die einer partiellen Abbildung zwischen den mit Quell- und Zielsorte des Operationssymbols assoziierten Trägermengen entspricht. Jede Trägermenge kann beliebig viele Elemente (`GGElement`) haben, die wiederum für jeden Attributtypen der zur Trägermenge gehörenden Sorte genau ein Attribut (`GGAttribute`) besitzen. Somit korrespondieren Attribute zu `DatatypeObject`- und Elemente zu `SymbolObject`-Instanzen. Auch für die Links, also `Operation`-Objekte, muss eine Entsprechung existieren, weshalb die Mappings einer Operation durch eine eigene Klasse, `GGOperationMapping`, dargestellt werden.

Wie erwähnt, müssen die Operationen nicht total sein. Außerdem besitzt zwar jedes Element einer Trägermenge alle Attribute, doch diese Attribute müssen nicht unbedingt einen Wert haben. Es sind also nicht alle `GGAlgebra`-Instanzen auch Repräsentationen gültiger visueller Sätze. Der Grund liegt darin, dass das Rahmenwerk dadurch deutlich angenehmer zu benutzen ist – so ist es zum Beispiel nicht nötig im gleichen Zuge mit dem Einfügen eines neuen Elements auch alle Operationen entsprechend zu vervollständigen.

Im Gegensatz zu den Signaturen sind die Algebren sehr wohl veränderbar. Neben den diversen `get*()` Methoden, die die Klassen analog zu den Signaturstrukturen besitzen, stellen sie also modifizierende Methoden zur Verfügung. Dies macht die Implementation verschiedener Mechanismen notwendig – wie etwa einen Observer- oder Listener-Mechanismus, damit andere Klassen auf Änderungen einer Algebra reagieren können – auf die nach einer Kurzbeschreibung aller Klassen näher eingegangen wird. Dabei soll auf die weitere Erwähnung einiger Methoden verzichtet werden: In jeder Klasse wird ein Objekt der in der Hierarchie darüberliegenden Klasse sowie der korrespondierenden Klassen der `GGSignature`- und `DiagramAlgebra`-Hierarchie (vergleiche Abb. 5.2 und 5.3) referenziert und es sind für jedes dieser Objekte entsprechende `get*()` Methoden implementiert. Teilweise gibt es solche Methoden sogar für Klassen, die erst über mehrere Schritte zu erreichen sind, zum Beispiel besitzt jede Klasse eine `getDiagramAlgebra()` Methode und `GGAttribute` stellt ein `getSort()` zur Verfügung.

5.3.2.1 `GGAlgebra`

Parameter für die Konstruktion eines `GGAlgebra`-Objekts sind ein Algebraname, eine Signatur und ein `DiagramAlgebra`-Objekt. Ist letzteres gegeben, so repräsentiert die neu konstruierte Algebra den selben visuellen Satz wie die `DiagramAlgebra`-Instanz, ansonsten wird eine leere Algebra über der Signatur kreiert. In jedem Fall werden `GGBaseSet`-Objekte für alle Sorten und `GGOperation`-Objekte für alle Operationssymbole angelegt.

Auf diese kann mittels `getBaseSets()` und `getOperations()` als `Collection` oder selektiver per `getBaseSetFor()` und `getOperationFor()` zugegriffen werden. Die ersten beiden Methoden liefern eine `Collection` aller Trägermengen bzw. Operationen zurück, die anderen beiden jene Trägermenge oder Operation, die zu einer übergebenen Sorte oder einem Operationssymbol korrespondiert.

Obwohl die bisher aufgezählten Methoden in Anbetracht der Tatsache, dass in den anderen Klassen alle benötigten Akzessor- und manipulativen Methoden implementiert werden, genügen, stellt `GGAlgebra` zur Bequemlichkeit noch weitere zur Verfügung. Diese wurden sporadisch hinzugefügt, als sie hilfreich erschienen, sodass ihre Sammlung etwas unvollständig wirkt.

`getElements()` und `getElementsOrdered()` liefern eine `Collection` aller Elemente der Algebra, letztere in der Reihenfolge, in der die zugehörigen `SymbolObject`-Objekte in der `DiagramAlgebra`-Instanz erscheinen. `getElementFor()` bekommt einen Elementnamen oder ein `SymbolObjekt`-Objekt und gibt das entsprechende Element zurück; `getAttributeFor()` funktioniert analog für Attribute. Mit `getZeroIndegreeElements()` steht eine Methode für Graphalgorithmen zur Verfügung, die alle Elemente sammelt, die nicht im Bild einer der Operation liegen. `getOperationMappings()` liefert eine `Collection` aller Operationsmappings und `isValid()` prüft, ob alle Operationen total sind.

Für die Manipulation der Algebra gibt es nur drei Methoden: `addElement()` fügt ein neues Element hinzu und `removeElement()` entfernt ein vorhandenes; `removeOperationMapping()` löscht ein Operationsmapping.

5.3.2.2 GGBaseSet

Die Methoden `getElements()` und `getElementFor()` liefern alle Elemente der Trägermenge bzw. das zu einem `SymbolObject`-Objekt korrespondierende Element.

`addElement()` fügt ein neues Element zur Trägermenge hinzu, `removeElement()` entfernt ein bestimmtes und `removeAllElements()` sämtliche enthaltene Elemente. `GGBaseSet` fungiert als Factory (Factory-Designpattern) für `GGElement`-Objekte: `createElement()` kreiert ein völlig neues Element, während `cloneElement()` ein existierendes Element der gleichen Sorte kloniert.

5.3.2.3 GGElement

Ein `GGElement`-Objekt wird immer zu einer gegebenen Sorte konstruiert, was ermöglicht, Attribute für alle Attributtypen der Sorte zu kreieren. Sie besitzen dann allerdings noch keine Werte.

`getAttributes()` liefert eine `Collection` aller Attribute, `getSetAttributes()` nur solcher, die einen Wert besitzen. Die zwei Versionen von `getAttributeFor()` geben das `GGAttribute`-Objekt zu einem Attributtyp bzw. zu einer `DatatypeObject`-Instanz zurück. `setAttribute()` setzt nicht, wie vielleicht zu erwarten, den Wert eines Attributs, sondern ersetzt das entsprechende `Attribute`-Objekt durch das übergebene.

5.3.2.4 GGAttribute

Der Wert für ein Attribut kann zweierlei sein: Einerseits ein Datentypwert, das heißt ein Objekt der Datentypklasse, oder andererseits ein `String`, der einen Ausdruck enthält. Beides kann von einem `DatatypeObject`-Objekt verwaltet werden.

Die Methode `isSet()` prüft, ob überhaupt ein Wert (also ein `DatatypeObject`-Objekt) für das Attribut gesetzt ist, `valueIsDatatype()` und `valueIsExpression()` testen zusätzlich, ob der Wert ein Datentypwert bzw. ein Ausdruck ist. Für den jeweiligen Fall liefern `getDatatypeValue()` und `getExpressionValue()` diesen Wert zurück. Da es eine Möglichkeit gibt, Datentypwerte in konstante Ausdrücke umzuwandeln (`Datatype.getStringRepresentation()`), ist es, so das Attribut überhaupt gesetzt ist, mittels `getValueAsExpression()` immer möglich dessen Wert als Ausdruck zu bekommen.

Gesetzt werden kann der Attributwert durch eine der `setValue()` Versionen: Es wird entweder ein `DatatypeObject`-Objekt, ein Datentypwert oder ein Ausdruck übergeben. `unsetValue()` entfernt, wenn vorhanden, den Wert das Attributs, sodass nachfolgende Aufrufe von `isSet()` *false* zurückliefern. Die Methode `createDefaultValue()` kreiert einen Defaultdatentypwert; das Attribut bleibt unverändert.

5.3.2.5 GGOperation

Auf die Trägermengen, die Domain und Codomain einer Operation darstellen, kann mittels `getDomain()` und `getCodomain()` zugegriffen werden. Eine `Collection` aller Mappings liefert `getMappings()`. Hat ein Element ein Bild unter der Operation, so zeigt `hasImage()` dies an und `getImageFor()` und `getMappingForSource()` geben das zugehörige Bild bzw. das Operationsmapping zurück. `getImages()` kreiert eine `Collection` aller Bilder, also das Bild der Operation.

So wie `GGBaseSet` für `GGElement` ist auch `GGOperation` eine Factory für `GGOperationMapping`-Objekte. Es gibt zwar eine `addMapping()` Methode, mit der explizit ein `GGOperationMapping`-Objekt hinzugefügt werden kann, doch wird dies gewöhnlich implizit durch `createMapping()` unter Angabe von Original- und Bildelement geschehen. `removeMapping()` entfernt ein bestimmtes Mapping, `removeAllMappings()` alle vorhandenen.

5.3.2.6 GGOperationMapping

Die Klasse `GGOperationMapping` trägt nur sehr wenige Informationen. Das Original- und das Bildelement eines Mappings liefern die Methoden `getSource()` und `getTarget()`.

5.3.2.7 `GGException`

In vielen der Methoden der verschiedenen Klassen können Fehler auftreten, die nicht vernachlässig- oder korrigierbar sind. Zum einen kann das daran liegen, dass `Alphabet`/`DiagramAlgebra` Methoden verwendet werden, die fehlschlagen und eine Exception (`AlphabetException`) werfen können, zum anderen daran, dass Parameter übergeben wurden, mit denen eine korrekte Abarbeitung der Methode nicht möglich ist. Ein weiterer Grund – vorrangig in der Implementierungs- und Testphase – ist die Möglichkeit, dass Objekte aufgrund von Design- oder Implementierungsfehlern in einen invaliden Zustand geraten sind. In jedem der Fälle wird eine Exception vom Typ `GGException` geworfen. Dies gilt für alle Signatur-, Algebra- und Grammatikklassen, ebenso wie für das Transformationsrahmenwerk.

Dem `GGException`-Konstruktor kann eine für die zu kreierende Exception ursächliche Exception übergeben werden, sodass es möglich ist, die auslösende Exception durch eine `GGException` zu kapseln. Konsequenterweise wirft keine Methode des Rahmenwerks – wir sehen von den Methoden zum Speichern und Laden von XML-Daten ab – eine Exception eines anderen Typs.

5.3.2.8 `GGObservable`-Mechanismus

Mehrere Klassen des Rahmenwerks implementieren das `GGObservable`-Interface, zumeist durch direktes oder indirektes Ableiten von `GGObservableImpl`. Das Anliegen ist es, einen Notifikationsmechanismus zur Verfügung zu stellen, der ähnlich einfach zu verwenden ist wie das Observer-Designpattern, jedoch so flexibel und übersichtlich ist wie das Listener-Designpattern. Bei ersterem stört, dass der Observer nur über eine Methode informiert wird, was bei der Überwachung vieler Aspekte oder gar verschiedener Objekte schnell unübersichtlich werden kann, während letzteres bei Anwachsen der Zahl der zu überwachenden Aspekte entweder zu einem Wust von Listener-Interfaces oder zu Interfaces mit sehr vielen Methoden führt.

Die Philosophie des Mechanismus ist einfach: Bei den zu überwachenden Objekten treten Events (Ereignisse) auf, über die die Observer zu informieren sind. Ein Event wird durch einen `String` identifiziert, die Bekanntgabe des Auftretens eines solchen geschieht durch die `GGObservable`-Methode `triggerEvent()` mit dem `String` als Parameter. Ein Observer wiederum muss kein spezielles Interface implementieren oder von irgendeiner Klasse ableiten, sondern er registriert sich bei einem `GGObservable`-Objekt (`addObserver()`) unter Angabe des Events, bei dem er benachrichtigt werden möchte und der Methode, die in diesem Fall aufgerufen werden soll. Was als Parameter dieser Methode übergeben wird, ist festgelegt: das benachrichtigende `GGObservable`-Objekt, der Event, auf den der Observer wartete, der Event, der ausgelöst wurde, und ein Objekt, das als Parameter des Events an `triggerEvent()` übergeben wurde. Der Typ des ersten Parameters ist nicht genau festgelegt; die wahre Klasse des Observables, eine beliebige Superklasse dieser oder `GGObservable` sind zulässig.

Worin der Unterschied zwischen dem zweiten und dritten Parameter besteht, leuchtet nach der Beschreibung einer Erweiterung des erläuterten Mechanismus ein. Möchte ein Observer bei einer großen Menge von verschiedenen Events benachrichtigt werden, so müsste er sich für jeden einzelnen registrieren. Um dies zu vereinfachen, werden Eventhierarchien eingeführt, repräsentiert durch die Klasse `GGObservableEventHierarchy`. Zu einer Hierarchie lassen sich einzelne Events hinzufügen (`addEvent()`) und es lässt sich für jeden Event definieren, welche anderen Events allgemeiner sind als dieser (`addBaseEvent()`, `addBaseEvents()`). Zwei ausgezeichnete Events sind immer in einer Hierarchie vertreten: `MOST_GENERAL_EVENT` ist allgemeiner und `MOST_SPECIFIC_EVENT` spezieller als jeder andere Event. Ist nun für ein Observable eine Eventhierarchie gegeben, so werden beim Auslösen eines Events nicht nur die Oberver informiert, die auf diesen, sondern auch solche, die auf allgemeinere Events warten.

Eine vollständige Implementation des `GGObservable`-Interface bietet die Klasse `GGObservableImpl` an, von der sich andere Klassen ableiten können und damit automatisch diesen Mechanismus zur Verfügung haben. Da Java keine multiple Vererbung erlaubt, ist es in manchen Fällen nicht möglich diese Klasse zu beerben, zum Beispiel bei GUI-Klassen, die zwangsläufig direkt oder indirekt von `Component` abstammen müssen. Für diesen Fall gibt es die Klasse `GGProxyObservable`, die als Stellvertreter für die „verhinderte“ Klasse fungieren kann. `AlgebraDisplay` zum Beispiel verwendet ein `GGProxyObservable`-Objekt, welches durch die Methode `getObservable()` zugänglich ist und bei dem sich Observer registrieren können.

Neben `GGAlgebra` und `GGBaseSet` verwenden mehrere Grammatikklassen (`GGGrammar`, `GGRule`, `GGMorphism`) den Observable-Mechanismus.

5.3.2.9 `StateEditable`, `Cloneable`, `assign()`

Dieser Abschnitt beschreibt verschiedene Mechanismen, die aber in die gleiche Richtung zielen. `StateEditable` ist ein von Java angebotenes Interface, dass dazu dient, den Zustand eines Objekts einer implementierenden Klasse in eine Hashtable zu speichern und später wieder herstellen zu können. Verwendet wird diese Funktionalität für den Undo-Mechanismus. Alle Algebra- und Alphabetklassen implementieren zu diesem Zweck `StateEditable`.

Objekte von Klassen, die das `Cloneable`-Interface implementieren, sind in der Lage, eine genaue Kopie von sich anzufertigen. Alle Alphabet- und Algebraklassen ebenso wie die Grammatikklassen ausgenommen `GGGrammar` selbst, implementieren diese Schnittstelle.

Viele von diesen Klassen stellen des Weiteren eine oder mehrere verschiedene `assign()`-Methoden zur Verfügung. Einer der Parameter dieser Methoden ist immer ein Objekt der gleichen Klasse, welches der Instanz, auf der die Methode aufgerufen wurde, zugewiesen werden soll. Nach der Abarbeitung ist letztere eine Kopie des übergebenen Objekts. Der Grund, warum kein `Assignable`-Interface vorgesehen ist, liegt darin, dass `assign` weitere klassenspezifische Parameter benötigt. So werden beim Zuweisen eines Morphismus zuerst Domain- und Codomainalgebra zugewiesen, wobei sich gemerkt werden muss, welches Element zu welchem kopiert wurde, damit es auch möglich ist, die Mappings des Morphismus entsprechend zu kopieren. Aus diesem Grund ist der zweite Parameter von `GGAlgebra.assign()` eine `Map`, in der die Zuordung der Elemente gespeichert wird.

5.3.2.10 Synchronisationsmechanismus

Wie in Abschitt 4.6 angedacht, wurde zum Freihandeditieren in der VL-Umgebung der Diagrammeditor des Alphabeteditors wiederverwendet. Während dieser jedoch `DiagramAlgebra`-Objekte bearbeitet, operieren alle anderen Komponenten der VL-Umgebung mit `GGAlgebra`-Instanzen. Zwar besitzt `GGAlgebra` einen Konstruktor, der eine `DiagramAlgebra` „wrappt", und eine Methode `getDiagramAlgebra()`, mit der sich die `DiagramAlgebra` zu einem `GGAlgebra`-Objekt liefern lässt, sodass eine Konvertierung in beide Richtungen möglich ist, doch stellte sich dieses Verfahren in der Praxis als inakzeptabel heraus; zum Beispiel deshalb, weil durch die Umwandlung einer `DiagramAlgebra` in eine `GGAlgebra` Informationen verloren gehen können.

Stattdessen wurde ein Synchronisationsmechanismus implementiert. Ein visueller Satz liegt in der VL-Umgebung immer als `GGAlgebra` vor. Der Diagrammeditor arbeitet auf der internen `DiagramAlgebra` dieser `GGAlgebra` und letztere wird über einen Listener-Mechanismus von den Änderungen informiert. Aus Performancegründen synchronisiert sich die `GGAlgebra` nicht nach jeder Veränderung, sondern merkt nur vor, dass eine Synchronisation notwendig ist – `isSynchronized()` gibt darüber Auskunft. Will man sicher sein, dass bei einem Zugriff auf die `GGAlgebra` auch aktuelle Daten verwendet werden, ruft man zuvor `synchronize()` oder `synchronizeForced()` auf. Ersteres überprüft zunächst, ob überhaupt eine Synchronisation notwendig ist, letzteres führt sie auf jedem Fall durch. Alle Methoden der Algebraklassen, die eine Modifikation der `DiagramAlgebra` zur Folge haben (können), sorgen automatisch für die Synchronisation. Die nur lesenden Methoden tun dies derzeit nicht, was weniger an Performancegründen liegt, sondern daran, dass `synchronize()` eine `GGException` werfen kann, jedoch fast alle Lesemethoden ohne solche auskommen. Wäre auch bei ihnen eine Synchronisation vorgesehen, müsste sämtlicher Code, der Algebraklassen verwendet, auch eine Behandlung der Ausnahmen vorsehen. Stattdessen ist nun lediglich an zwei Stellen im Quellcode der VL-Umgebung ein `synchronize()` aufzurufen.

Da der Synchronisationsmechanismus nur an sehr wenigen Stellen Sinn macht – nämlich nur dort, wo mit verschiedenen Komponenten an einem visuellen Satz sowohl als `GGAlgebra` als auch als `DiagramAlgebra` gearbeitet werden muss – ist er abschaltbar, das heißt genauer, er ist voreingestellt abgeschaltet und kann mit der Methode `setSynchronizationEnabled()` eingeschaltet werden.

5.3.3 Grammatik

Eine Grammatik (`GGGrammar`) besteht aus Regeln (`GGRule`) und einem Startdiagramm (`GGAlgebra`), alle getypt über der selben Signatur. Sie kann vom Typ `SYNTAX_GRAMMAR`, `PARSING_GRAMMAR` oder `SIMULATION_GRAMMAR` sein und hat ein `GGMatchCompletionInfo`-Objekt, das für die Transformation (Abschnitt 5.4) von Interesse ist.

Eine Regel besteht aus einem Regelmorphismus (`GGMorphism`), einer Menge von NACs, repräsentiert durch Morphismen, einer Menge von Regelparametern (`GGParameter`) und Attributbedingungen (boolesche Ausdrücke enthaltende Strings).

Ein Morphismus schließlich besitzt zwei Algebren als Domain und Codomain und eine `Map`, die Elemente des Domains auf die Elemente des Codomains abbildet. Er wird also nicht gemäß der Theorie als sortenindizierte Familie von Abbildungen dargestellt.

Abbildung 5.4.

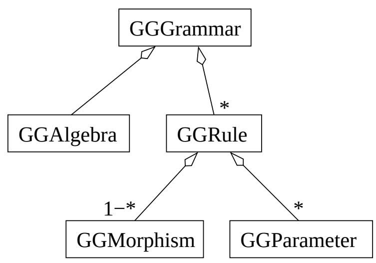

Abbildung 5.4: Struktur der Grammatikklassen.

5.3.3.1 `GGMorphism`

Parameter für die Konstruktion eines `GGMorphism`-Objekts sind ein Name für den Morphismus, die Signatur, über die er getypt sein soll sowie Algebren für Domain und Codomain. Sind Domain oder Codomain nicht gegeben, wird jeweils eine leere Algebra kreiert, die dann dem Morphismus gehört. Übergebene Algebren gehören ihm nicht. Diese Unterscheidung ist von Bedeutung zum einen für die Namensgebung der Algebren – zum Morphismus gehörende bekommen den mit einem Suffix versehenen Morphismusnamen – zum anderen für das Klonieren des Morphismus – Besitztümer des Morphismus werden ebenfalls kloniert, ihm nicht gehörende Algebren einfach übernommen. In jedem Fall hat ein `GGMorphism`-Objekt nach der Konstruktion keine Mappings.

Nachträglich lässt sich mittels `setOwnerFlags()` noch setzen, ob Domain oder Codomain dem Morphismus gehören sollen oder nicht. Parameter ist ein aus `DOMAIN_OWNER` und `CODOMAIN_OWNER` zusammensetzbarer Wert. Mit `getOwnerFlags()` kann auf dessen aktuellen Wert zugegriffen werden, angenehmer zu verwenden sind allerdings `ownsDomain()` und `ownsCodomain()`, die direkt darüber Auskunft geben, ob Domain bzw. Codomain dem Morphismus gehören.

Domain und Codomain selbst liefern `getDomain()` und `getCodomain()`, zu setzen sind sie mittels `setDomain()` und `setCodomain()`. Die Methoden `assignDomain()` und `assignCodomain()` setzen kein neues `GGAlgebra`-Objekt, sondern weisen dem aktuellen per `assign()` (Abschnitt 5.3.2.9) das übergebene zu.

Die Namen des Morphismus selbst oder seines Domains oder Codomains setzen `setName()`, `setDomainName()` und `setCodomainName()`.

Alle weiteren Methoden haben direkt oder indirekt mit den Mappings des Morphismus zu tun. `getMappings()` liefert ein `Map`-Objekt, dass die Elementmappings enthält, `getMappingsFor()` mit einer Sorte als Parameter gibt nur die Mappings zurück, die die Elemente dieser Sorte betreffen. `getMappedElements()` und `getUnmappedElements()` kreieren eine `Collection`, die alle Elemente des Domains enthalten, die abgebildet bzw. jene die nicht abgebildet werden. Die Methode `hasImage()` prüft, ob ein bestimmtes Element bezüglich des Morphismus ein Bild besitzt und wenn dies so ist, gibt `getImageFor()` es zurück.

Auf die Mappings kann mittels mehrerer Methoden Einfluss genommen werden: `setMappings()` bekommt eine `Map` als Parameter, die die Mappings enthält, die der Morphismus haben soll, `addMapping()` installiert ein neues Mapping zwischen zwei Elementen, `removeMapping()` entfernt ein solches und `removeAllMappings()` löscht alle vorhandenen Mappings.

Die Methode `isTotal()` prüft, ob der Morphismus total ist, `isValid()` untersucht seine Validität. Dabei wird die Domain- und die Codomain-Algebra auf Totalität ihrer Operationen geprüft und festgestellt, ob der Morphismus die Homomorphiebedingungen erfüllt.

5.3.3.2 `GGParameter`

Ein `GGParameter`-Objekt repräsentiert einen Regelparameter und besitzt einen Namen und einen Typ (`GGAttributeType`). Für beide gibt es `get*()`- und `set*()`-Methoden. Schließlich bleiben noch `getTypename()` und `getClassname()`, die den Namen des Typs bzw. den Namen seiner Datentypklasse zurückgeben.

5.3.3.3 `GGRule`

Es können ausschließlich über einer gegebenen Signatur leere Regeln ohne NACs, Regelparameter und Attributbedingungen konstruiert werden.

LHS, RHS und den Regelmorphismus erhält man durch die Methoden `getLHS()`, `getRHS()` und `getMorphism()` und darf sie nach Belieben manipulieren.

NACs können hinzugefügt (`addNAC()`), kreiert (`createNAC()`, Factory-Designpattern) und entfernt werden (`removeNAC()`). Ob die Regel eine bestimmte NAC enthält, untersucht `hasNAC()`, `getNACs()` liefert eine `Collection` aller NACs.

Für die Manipulation der Parameter stehen die Methoden `setParameters()` zum Setzen der ganzen Parametermenge, `addParameter()` zum Hinzufügen eines Parameters, `createParameter()` zum Kreieren eines neuen (Factory-Designpattern) und `removeParameter()` und `removeAllParameters()` zum Entfernen eines bzw. aller Parameter. Eine `Collection` der Parameter liefert `getParameters()`, einen Parameter mit einem bestimmten Namen `getParameter()`.

Die Menge der Attributbedingungen kann mit `setAttributeConditions()` gesetzt werden, eine einzelne wird durch `addAttributeCondition()` hinzugefügt und `getAttributeConditions()` gibt eine `Collection` aller Attributbedingungen zurück.

5.3.3.4 `GGGrammar`

Konstruiert wird immer ein `GGGrammar`-Objekt über einer Signatur und ohne Regeln. Der Typ der Grammatik ist mit `SYNTAX_GRAMMAR` voreingestellt, `getType()` liefert ihn zurück und mit `setType()` lässt er sich ändern. Die Methode `getMatchCompletionInfo()` gibt das `GGMatchCompletionInfo`-Objekt der Grammatik zurück; weitere Informationen dazu finden sich im Abschnitt 5.4.

Die Grammatik dient als Factory für ihre Regeln (`createRule()`), sie können jedoch auch per `addRule()` hinzugefügt werden. `removeRule()` dient zum Entfernen einer Regel, `getRules()` liefert einen `Vector` aller Regeln.

Das Startdiagramm wird durch `setStartAlgebra()` gesetzt, `getStartAlgebra()` gibt das aktuelle Startdiagramm zurück.

5.3.3.5 `GGGrammarCreator`

Die Klasse `GGGrammarCreator` dient dem Erzeugen der Alphabetregeln für eine Grammatik und besitzt ausschließlich statische Methoden. Von diesen sind nur zwei öffentlich, die anderen stellen Hilfsmethoden dar, die hier nicht erwähnt werden.

`createGrammar()` bekommt als Parameter eine Signatur, einen Grammatiktyp und einen booleschen Wert, der angibt, ob die Dangling-Condition verwendet werden soll. Die Methode kreiert eine neue Grammatik und alle Alphabetregeln zur Signatur nach dem in 2.2.4 beschriebenen Algorithmus.

Die Methode `adjustGrammarToSignature()` ist zum Beispiel für den Fall implementiert, dass der Spezifikateur einer VL-Spezifikation ein visuelles Alphabet kreiiert hat und schon an einer Grammatik arbeitet, wobei er der Tatsache gewahr wird, im Alphabet ein Symbol vergessen zu haben. Da das Grammatikrahmenwerk davon ausgeht, dass die Signatur ein für alle Mal feststeht, müsste die gesamte Arbeit an der Grammatik verworfen werden. `adjustGrammarToSignature()` bekommt eine Grammatik und eine Signatur und versucht die Grammatik an die Signatur anzupassen und so das Problem zu lösen. Die derzeitigen Fähigkeiten der Methode sind allerdings noch recht beschränkt. So wird bisher nur festgestellt, ob neue Symbole oder Datentypen hinzugekommen sind und die entsprechenden Alphabetregeln werden generiert. Das neue Alphabet muss das alte vollständig enthalten und es dürfen auch keine Links hinzukommen, die neue Symbole als Ziel haben.

5.4 Transformation

Dieser Abschnitt ist in zwei Unterabschnitte unterteilt: Der erste beschreibt Klassen, die für die Transformation benötigte Daten enthalten – dies sind `GGMatch` und `GGMatchCompletionInfo` – sowie die Interfaces, mit denen die Abstraktion von der konkreten Transformationsmaschine realisiert werden. Der zweite erläutert die spezielle Implementierung dieser Schnittstelle zur AGG-Anbindung.

5.4.1 Allgemeine Schnittstelle

Das Rahmenwerk für die Transformation ist relativ einfach aufgebaut. Zum einen gibt es die Klassen `GGMatch` und `GGMatchCompletionInfo`, die die für eine Transformation benötigten Daten bereitstellen, zum anderen das `GGAlgebraTransformer`-Interface, das definiert, welche Funktionalität letztlich zu implementieren ist. `GGMatch` repräsentiert einen Match für eine gegebene Regel und einen visuellen Satz. Dies beinhaltet neben dem Matchmorphismus auch eine Belegung der Regelparameter und optional ein `GGMatchCompletionInfo`. Dieses enthält alle Parameter, die die Matchvervollständigung beeinflussen.

Ein Objekt einer `GGAlgebraTransformer` implementierenden Klasse lässt sich durch die Methode `createAlgebraTransformer()` der globalen Factory `GGFactory` erhalten. Diese implementiert noch weitere Methoden, die vorrangig Objekte für die Parsierung kreieren, weshalb ihre Beschreibung im Abschnitt 5.5 zu finden ist.

5.4.1.1 `GGMatchCompletionInfo`

Der einzige derzeit in GenGEd vorgesehene Parameter für die Matchvervollständigung ist die Dangling-Condition. Der `GGMatchCompletionInfo`-Konstruktor setzt den Wert für diesen Parameter voreingestellt auf *false*, das heißt, die Dangling-Condition würde für einen Match nicht berücksichtigt. `setDanglingCondition()` ändert den Wert, `getDanglingCondition()` liefert ihn zurück.

5.4.1.2 `GGMatch`

Ein Match wird unter Angabe einer Regel und einer Algebra konstruiert, optional kann des Weiteren ein `GGMatchCompletionInfo`-Objekt bereitgestellt werden. Der Matchmorphismus ist nach der Konstruktion leer.

Zum Zugriff auf die Regel, die Algebra oder den Morphismus des Matches existieren die Methoden `getRule()`, `getAlgebra()` und `getMorphism()`. Das `GGMatchCompletionInfo` kann mittels `setCompletionInfo()` gesetzt und von `getCompletionInfo()` geliefert werden.

Die weiteren Methoden, abgesehen von `isTotal()`, welches den Matchmorphismus auf Totalität prüft, dienen der Verwaltung der Regelparameterwerte. `setParameterValue()` setzt den Wert eines bestimmten Parameters, `unsetAllParameterValues()` entfernt die Werte sämtlicher Parameter. Mittels `getParameterValue()` lässt sich der Wert eines Parameters erfahren, `getParameterValueAsString()` liefert ihn repräsentiert als String. Die Methode `getParameters()` gibt alle Regelparameter als `Collection` zurück und `getUnsetParameters()` nur solche, für die kein Wert definiert ist.

5.4.1.3 `GGAlgebraTransformer`

Bevor ein `GGAlgebraTransformer`-Objekt einsatzbereit ist, muss es durch den Aufruf der Methode `init()` mit einer Signatur als Parameter vorbereitet werden. Nachfolgend darf bis zum nächsten `init()`-Aufruf nur mit Objekten, die über dieser Signatur getypt sind, gearbeitet werden. Dies geschieht primär zur Verbesserung der Effizienz der nachfolgenden Methoden.

`checkMorphism()` und `checkMatch()` überprüfen, ob ein partieller Morphismus bzw. Match entsprechend der `GGAlgebraTransformer`-Implementation korrekt sind. Dazu gehört zumindest, dass `GGMorphism.isValid()` den (Match-)Morphismus positiv beurteilt, doch abhängig von den Bedürfnissen der Transformationsmaschine kann auch die Erfüllung weiterer Bedingungen gefordert werden.

Die Methoden `completeMorphism()` und `completeMatch()` versuchen den übergebenen Morphismus/Match zu vervollständigen und liefern *true* zurück, wenn dies gelungen ist. Bei der Vervollständigung des Matches werden, so vorhanden, die durch dessen `GGMatchCompletionInfo` gegebenen Parameter beachtet.

Die eigentliche Transformation führt `apply()` durch. Einziger Parameter ist ein Match, durch den die zu transformierende Algebra, die anzuwendende Regel und der Matchmorphismus gegeben sind. Ist letzterer nicht total, so wird er mittels `completeMatch()` vervollständigt. Kann keine Vervollständigung gefunden werden, ist der Match nicht korrekt (`checkMatch()`) oder sind nicht alle Regelparameter belegt, so kann die Transformation nicht durchgeführt werden. Doch auch anderenfalls kann sie scheitern – zum Beispiel, wenn der Match zwar von `checkMatch()` als korrekt befunden wird, aber, da dort die Werte der Regelparameter nicht beachtet werden, seine Attributkomponente die Korrektheit zunichte macht. Gelingt die Transformation, so wird die Algebra verändert („in place" Transformation, [Rud97]) und `apply()` liefert *true* zurück, im anderen Fall bleibt die Algebra unangetastet und *false* wird zurückgegeben.

5.4.2 Agg-Anbindung

Die Anbindung von AGG zur Transformation erfolgt durch die Implementation des `GGAlgebraTransformer`-Interfaces durch die Klasse `AGGAlgebraTransformer`. Im Wesentlichen werden dafür die GENGED-Strukturen in AGG-Strukturen konvertiert, mit diesen die Transformation durchgeführt und das Resultat zurückkonvertiert. Die Dienste zur Konvertierung sind hauptsächlich in der Klasse `AGGGrammarConverter` gekapselt. Diese verwendet einige Hilfklassen (`AGG*ConversionInfo`), die bei der Konvertierung anfallende und für die Rückkonvertierung notwendige Daten sammeln. Doch bevor auf die einzelnen Klassen eingegangen wird, erläutert der erste Unterabschnitt, wie die Abbildung der GENGED- auf die AGG-Strukturen erfolgt.

5.4.2.1 Konvertierung der Strukturen

Die Tatsache, dass es sich bei den zu konvertierenden Algebren um Graphstrukturen handelt, macht diese Aufgabe prinzipiell recht einfach: Algebraelemente werden zu Knoten, Operationsmappings zu Kanten und die Attribute der Elemente zu Knotenattributen. Zum Zeitpunkt der Implementierung des Rahmenwerks war in GENGED vorgesehen, auch mit nicht total attributierten visuellen Sätzen zu arbeiten. In der Version, die GRAMED verwendete, unterstützte AGG ebenfalls partielle Attributierung, doch seit der Überarbeitung der Attributkomponente müssen zu transformierende Graphen immer vollständig attributiert werden. Es galt also, einen Weg zu finden, partiell attributierte Algebren in total attributierte Graphen umzuwandeln.

Dazu wurde das oben erwähnte Prinzip dahin gehend erweitert, dass ein Knoten zu jedem Attribut ein weiteres, boolesches Attribut bekommt, welches anzeigt, ob ersteres überhaupt gesetzt ist. Ein in der GENGED-Algebra nicht gesetztes Attribut bekommt im AGG-Graphen einen beliebigen Wert und das zugehörige Indikatorattribut wird auf *false* gesetzt, andernfalls wird es *true* und der Attributwert wird übernommen. Abbildung 5.5 veranschaulicht dies am Beispiel eines einfachen Statecharts (wie im Kapitel 2 mehrfach verwendet), welches einen benannten und einen unbenannten Zustand besitzt.

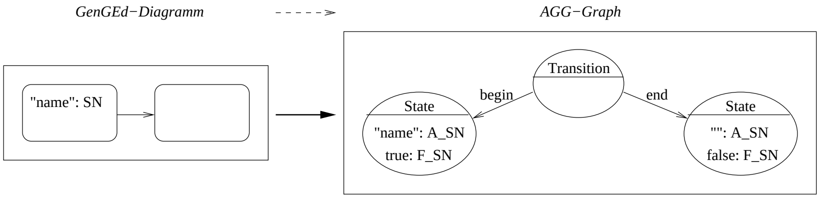

Abbildung 5.5: Beispiel für die Konvertierung von GenGEd- in Agg-Strukturen.

Für Morphismen erfolgt die Konvertierung kanonisch: Domain und Codomain werden konvertiert, ein Knoten wird durch den Graphmorphismus auf einen anderen abgebildet, wenn der Homomorphismus eine Abbildung zwischen den zugehörigen Elementen vorsah, und eine Kante wird abgebildet, wenn der Homomorphismus die zu Quell- und Zielknoten korrespondierenden Elemente abbildete.

Eine Regel zu konvertieren ist ungleich komplizierter, da dies so geschehen muss, dass bei der Transformation auch das gewünschte Verhalten resultiert. Dazu ist festzuhalten, was denn überhaupt gewünscht wird: Ist in der linken Regelseite ein Attribut eines Elements nicht gesetzt, so soll das Element unabhängig von diesem Attribut, auch unabhängig davon, ob es überhaupt gesetzt ist, auf Elemente matchen. Wird das Element in die rechte Regelseite abgebildet und hat das dortige Elemente ebenfalls keinen Wert für das Attribut, so soll das entsprechende Attribut in der zu transformierenden Algebra nicht verändert werden. Ist das Attribut auf der rechten Regelseite gesetzt, so wird der Wert des Attributs in die transformierte Algebra übernommen. Ist ein Attribut auf der linken, jedoch nicht auf der rechten Regelseite gesetzt, so hat auch die resultierende Algebra keinen Wert dafür. Bei der Konvertierung werden im ersten Fall sowohl auf der linken als auch der rechten Seite der Graphregel für das Attribut selbst und auch sein Indikatorattribut Variablen gesetzt (auf beiden Seiten die gleichen), im zweiten Fall geschieht dies nur für die linke Seite, die rechte wird „normal“ übersetzt, und im dritten Fall werden beide Seiten dem Standardverfahren entsprechend konvertiert. In den NACs werden für ein ungesetztes Attribut im NAC-Graphen in jedem Fall sowohl das Attribut selbst als auch das Indikatorattribut mit Variablen belegt – hat das zugehörige Element ein Urbild in der linken Regelseite und ist in diesem das Attribut auch nicht gesetzt, so werden die gleichen verwendet, ansonsten neue.

Da nun entsprechend der Konzepte die visuellen Sätze immer total attributiert sind, könnten auch die Indikatorattribute und die soeben beschriebene aufwendige Behandlung der Sonderfälle aus dem Transformationsrahmenwerk entfernt werden. Doch da es stabil zu funktionieren scheint und auch keine anderen Gründe dagegen sprechen, soll die zusätzliche Funktionalität zur Transformation partiell attributierte visueller Sätze im System verbleiben, auch wenn das GUI nur die

Arbeit mit totaler Attributierung kennt.

5.4.2.2 AGG*ConversionInfo

Die Objekte der Konvertierungsinfoklassen sind Container für Informationen, die bei der Konvertierung der GenGEd- in Agg-Strukturen anfallen. Sie werden unter Angabe, welche Daten von Interesse sind, kreiiert und sammeln später nur diese.

`AGGAlgebraConversionInfo` ist ein Behälter für Informationen über die Konvertierung eines `GGAlgebra`- in ein `Graph`-Objekt. Bei der Umwandlung einer Algebra in einen Graphen, werden Elemente (`GGElement`) zu Knoten (`Node`) und Operationsmappings (`GGOperationMapping`) zu Kanten (`Arc`). Welches Element dabei welchem Knoten und welches Operationsmapping welcher Kante entspricht, speichert das Konvertierungsinfo für Hin- und Rückrichtung in jeweils einer `Map`. Zugriff auf die `Map`-Objekte erhält man über die vier `get*Map()`-Methoden. Angenehmer anzuwenden sind allerdings `getNodeFor()`, `getElementFor()`, `getArcFor()` und `getOperationMappingFor()`, die jeweils zu einem Element, Knoten, Operationsmapping oder einer Kante das entsprechende Objekt liefern. Vier Versionen der `put()`-Methode assozieren solche Paare – sie werden bei der Konvertierung verwendet.

`AGGMorphismConversionInfo` ist relevant für die Konvertierung eines `GGMorphism`-Objekts in einen Graphmorphismus (`OrdinaryMorphism`). Es beinhaltet je ein `AGGAlgebraConversionInfo` für Domain und Codomain des Morphismus. Der Zugriff auf diese erfolgt per `getDomainConversionInfo()` und `getCodomainConversionInfo()`.

Bei der Konvertierung einer Regel (`GGRule` nach `Rule`) werden die NACs zwar berücksichtigt, doch in einem `AGGRuleConversionInfo`-Objekt werden Daten zu diesen nicht benötigt. Dieses enthält nur ein `AGGMorphismConversionInfo` für den Regelmorphismus. Darauf und auf die Algebrakonvertierungsinfos der linken und rechten Regelseite kann mittels `getMorphismConversionInfo()`, `getLHSConversionInfo()` und `getRHSConversionInfo()` zugegriffen werden.

Ein `GGMatchConversionInfo` referenziert neben einem `GGAlgebraConversionInfo` für die zu transformierende Algebra und einem `GGRuleConversionInfo` für die Regel (zur angenehmeren Handhabung) zusätzlich den Graphen, die Graphregel und den Graphmatch, die Ergebnisse der Konvertierung. Die Klasse stellt entsprechende `get*()`- und für letztere auch `set*()`-Methoden zur Verfügung.

5.4.2.3 AGGGrammarConverter

Ein `AGGGrammarConverter`-Objekt (Abb. 5.6) kann Objekte wie Regeln, Morphismen und Algebren in ihre Agg-Gegenstücke konvertieren, jedoch auch eine ganze

Grammatik. Letzteres ist fast so etwas wie ein Nebeneffekt des ersteren, denn intern muss der Konverter ohnehin ein `GraGra`-Objekt (Graphgrammatik) vorhalten, da nur im Kontext eines solchen Knoten- und Kantentypen kreiiert werden können, sodass nur noch die konvertierten Regeln hinzugefügt werden müssen. Vor der Benutzung muss eine Initialisierung mittels einer der `init()`-Methoden erfolgen. Die erste Version bekommt eine Signatur und kreiert entsprechend dieser Knoten-, Kanten- und Attributtypen. Die gegenseitige Zuordnung von GenGEd- und Agg-Typen wird durch sechs `Map`-Objekte realisiert, `get*For()`-Methoden erleichtern den Zugriff. Die zweite `init()`-Version hat eine Grammatik als Parameter; zur Signatur der Grammatik werden die Typen erzeugt und anschließend mittels `convertAlgebra()` das Startdiagramm und mit `convertRule()` alle Regeln konvertiert und zur Graphgrammatik hinzugefügt. Mit `getGraGra()` kann man sich danach das `GraGra`-Objekt abholen.

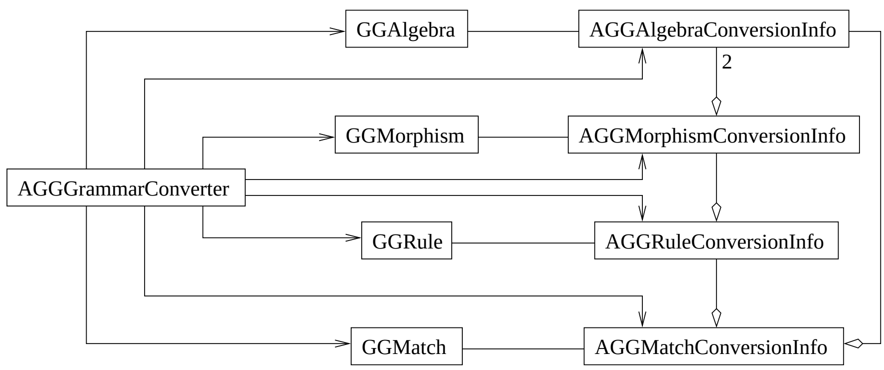

Abbildung 5.6: Struktur der Konvertierungsklassen.

`convertAlgebra()` gibt es in fünf Versionen, die sich in ihren Parametern unterscheiden, letztlich verwenden alle jedoch die gleiche Version. Dieser werden neben der zu konvertierenden Algebra, optional der Graph, in den die Knoten und Kanten eingefügt werden sollen, und `Map`-Objekte für die Element-Knoten- und Operationsmapping-Kanten-Zuordung übergeben (eine der Versionen bekommt statt der `Map`-Objekte sinnvollerweise ein Konvertierungsinfo). Zurückgegeben wird der resultierende Graph.

Für die Konvertierung eines Morphismus werden mehr Parameter benötigt. Die `convertMorphism()`-Methoden unterscheiden sich nur in der Art der Parameter für die Konvertierungsinformation: Eine bekommt ein `AGGMorphismConversionInfo`, eine je ein `AGGAlgebraConversionInfo` für Domain und Codomain und die dritte acht `Map`-Objekte. Weitere Parameter sind neben dem zu konvertierenden Morphismus optional ein `OrdinaryMorphism`-Objekt, Domain- und Codomaingraphen sowie zwei boolesche Parameter, die anzeigen, ob Domain bzw. Codomain schon konvertiert wurden. Diese große Anzahl von Parametern ist nötig, da `convertMorphism()` verschiedene Arten von Aufgaben zu lösen hat: Morphismen müssen komplett mit Domain und Codomain konvertiert werden können (etwa Regelmorphismen), ebenso wie solche, bei denen Domain und/oder Codomain schon konvertiert wurden (Match-/NAC-Morphismen). Der resultierende Graphmorphismus wird zurückgeliefert.

Auch die `convertRule()`-Versionen unterscheiden sich nur darin, welche Container für die Konvertierungsdaten zur Verfügung gestellt werden: keine, ein `AGGRuleConversionInfo` oder acht `Map`-Objekte. Ansonsten ist nur die zu konvertierende Regel zu übergeben und optional eine Graphregel. Die Methode gibt das Ergebnis der Konvertierung zurück. Die GenGEd-Regel und die Graphregel werden außerdem mittels zwei interner `Map`-Objekte miteinander assoziiert, sodass später mit `getAGGRuleFor()` bzw. `getRuleFor()` zur einen die jeweils andere wiedergefunden werden kann.

`convertMatch()` hat keinen Rückgabewert, sondern speichert seine Ergebnisse im zu übergebenden `AGGMatchConversionInfo`, dem zweiten Parameter nach dem zu konvertierenden Match.

Der Vollständigkeit halber soll an dieser Stelle erwähnt werden, dass `AGGGrammarConverter` eine Methode `convertLayerFunction()` zum Konvertieren einer GenGEd-Layerfunktion (`GGLayerFunction`) in eine Agg-Layerfunktion (`LayerFunction`) besitzt.

Nachdem die Methoden zur Umwandlung der GenGEd- in die Agg-Strukturen besprochen wurden, kommen wir zu den Methoden für die Rückrichtung. Um genau zu sein, ist die Rückkonvertierung nur in dem Fall interessant, wenn eine Algebra in einen Graphen konvertiert und dieser transformiert wurde. Es gilt also nur, die Algebra an den veränderten Graphen anzupassen. Für jeden neuen Knoten muss ein neues Algebraelement angelegt, für einen veränderten Knoten das zugehörige Element angepasst und für einen nicht mehr vorhandenen gelöscht werden. Der letzte Fall benötigt offensichtlich keine Hilfsfunktion, jedoch die anderen beiden. Zu diesem Zweck gibt es die Methoden `createElementForNode()` und `updateElementWithNode()`, die genau dies für gegebene Algebra, Graph, Knoten und Element erledigen.

5.4.2.4 `AGGAlgebraTransformer`

Für seine Arbeit benötigt ein `AGGAlgebraTransformer` (Abb. 5.7) einen `AGGGrammarConverter`. Die `init()`-Methode ruft direkt dessen `init()` auf. `checkMorphism()` und `checkMatch()` sind minimal implementiert, dass heißt, sie berufen sich, wie in Abschnitt 5.4.1.3 erläutert, nur auf `GGMorphism.isValid()`.

`completeMorphism()` und `completeMatch()` konvertieren Morphismus bzw. Match in die entsprechende Agg-Struktur und übertragen, falls Agg die Vervollständigung dieser gelingt, die neu hinzugekommenen Mappings auf das Ausgangsobjekt.

`apply()` arbeitet ganz ähnlich: Als erstes wird der Match konvertiert und ver-

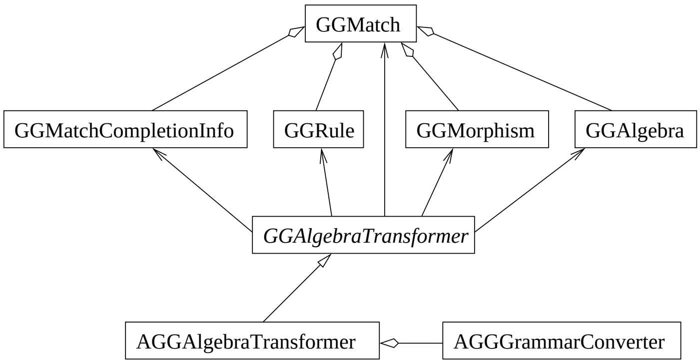

Abbildung 5.7: Struktur der Transformationsklassen.

vollständigt. War dies erfolgreich, so wird AGG für die Transformation bemüht und, so diese ebenfalls glückte, die Änderungen am Graphen auf die Algebra übertragen.

5.5 Parsierung

Die Parsierung ist ebenso wie die Transformation auf die entsprechende Fähigkeit des AGG-Systems zurückgeführt. Im ersten Unterabschnitt wird beschrieben, welche GENGED-Klassen und -Interfaces für die Parsierung vorgesehen sind, der zweite Unterabschnitt erläutert die Implementation der Schnittstelle unter Rückgriff auf AGG. Einen Überblick über die Klassenstruktur gibt Abbildung 5.8.

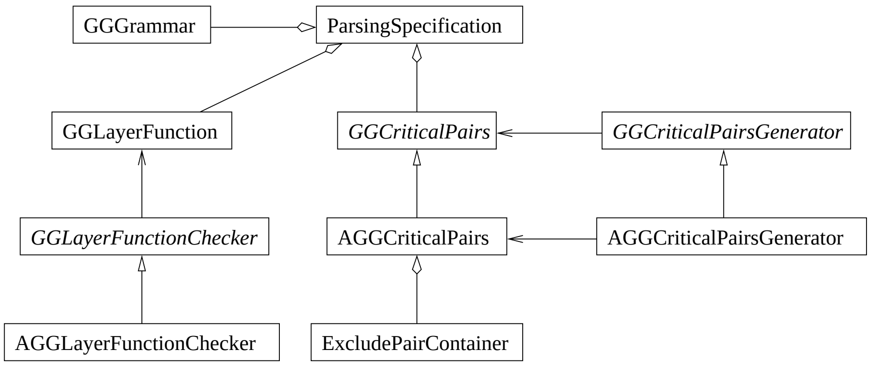

Abbildung 5.8: Struktur der Parsierungsklassen.

5.5.1 Allgemeine Schnittstelle

Die für die Parsierung notwendigen Daten werden entsprechend der Konzepte (Abschnitt 2.3) in einer Parsierungsspezifikation (`ParsingSpecification`) gehalten.

Diese besteht aus der Parsierungsgrammatik und optional einer Layerfunktion und dem Ergebnis einer Analyse der kritischen Paare. Anders als bei der Transformation können die GenGEd-Strukturen für die Parsierung nicht alle notwendigen Daten verwalten. Für die Layerfunktion ist dies zwar möglich – die Klasse `GGLayerFunction` zeichnet sich dafür verantwortlich – nicht jedoch für die kritischen Paare – dafür wird nur ein Interface, `GGCriticalPairs`, bereitgestellt. Der Grund liegt darin, dass die Analyse der kritischen Paare und die Parsierung selbst sehr eng zusammenhängen, da letztlich der Parser mit dem Resultat der Analyse arbeiten muss, und es nur an ihm liegt, welche Daten dazu gespeichert werden müssen.

Neben einem Interface für den Parser (`GGParser`) ist deshalb auch eines für die Analyse der kritischen Paare (`GGCriticalPairsGenerator`) vorgesehen. Ein weiteres, `GGLayerFunctionChecker`, stellt einen Mechanismus zur Verfügung, zu überprüfen, ob eine Layerfunktion den Ansprüchen des Parsers genügt.

5.5.1.1 GGLayerFunction

Ein `GGLayerFunction`-Objekt speichert die Rule-, Creation- und Deletion-Layers in drei `Map`-Objekten. Diese werden initialisiert, das heißt auf 0 gesetzt, durch die mit einer Grammatik parametrisierten `init()`-Methode. Die `Map`-Objekte lassen sich durch `get*Layers()`-Methoden bekommen, ein Layer für eine bestimmte Regel oder Sorte liefern die `get*Layer()`-Methoden und es lässt sich setzen per `set*Layer()`.

5.5.1.2 GGCriticalPairs

Das `GGCriticalPairs`-Interface erweitert `XMLObject` (siehe dazu Abschnitt 5.8), fügt jedoch keine weiteren Methoden hinzu. Dies sollte wenig überraschen, da implementierende Klassen Daten repräsentieren sollen, deren konkreter Inhalt auch nur dort bekannt ist.

5.5.1.3 ParsingSpecification

Eine `ParsingSpecification` besteht aus einer Parsierungsgrammatik, einer Layerfunktion, kritischen Paaren und einer booleschen Variable, die anzeigt, ob die Layerfunktion verwendet werden soll. Für jede dieser Variablen gibt es eine `get*()`- und eine `set*()`-Methode. Es bleibt anzumerken, dass die Daten allerdings nicht unabhängig voneinander verändert werden können: Wird zum Beispiel eine neue Grammatik gesetzt, so werden auch die Layerfunktion neu initialisiert und die kritischen Paare verworfen. Letzteres geschieht auch dann, wenn nur die Layerfunktion oder die boolesche Variable, die deren Verwendung bestimmt, verändert wird.

5.5.1.4 GGLayerFunctionChecker

Das `GGLayerFunctionChecker`-Interface ist denkbar einfach aufgebaut: Es sieht nur eine einzige Methode, `check()` vor, die eine Layerfunktion übergeben bekommt und ein boolesches Resultat liefert.

5.5.1.5 GGCriticalPairsGenerator

Die Analyse der kritischen Paare kann sehr lange dauern, weshalb vorgesehen ist, dass sie asynchron durchgeführt wird. Nach der Initialisierung des Generators per `init()`, welches entweder eine Parsierungsspezifikation oder eine Grammatik und optional eine Layerfunktion übergeben bekommt, ist er bereit und kann mit einem Aufruf von `start()` gestartet werden. Die Methode kehrt sofort zurück, die Arbeit wird in einem separaten Thread verrichtet. Ist die Analyse beendet, kann man das Ergebnis per `getCriticalPairs()` abholen. Für den vorzeitigen Abbruch der Analyse ist die Methode `stop()` vorgesehen.

Um informiert zu werden, wenn das Ergebnis bereitsteht und wie weit die Analyse fortgeschritten ist, muss eine Klasse das `CPGeneratorListener`-Interface implementieren und sich per `addCPGeneratorListener()` beim Generator registrieren. Die Methode `removeCPGeneratorListener()` kann aufgerufen werden, wenn kein Interesse an weiteren Benachrichtigungen besteht. Alle Informationen werden über die `CPGeneratorListener`-Methode `processCPGeneratorEvent()` in Form eines `CPGeneratorEvent`-Objekts übermittelt.

Ein solches unterscheidet die vier Ereignisarten `GENERATION_STARTED`, `GENERATION_STOPPED`, `GENERATION_FINISHED` und `GENERATION_PROGRESS`. Die ersten drei stehen für Beginn, Abbruch und erfolgreiche Beendigung der Analyse, das letzte zeigt den Fortschritt der Berechnung an. Neben einem von `getMessage()` gelieferten `String`, der Auskunft über die derzeitige Tätigkeit des Generators gibt, stellt `CPGeneratorEvent` die Methoden `getStepsToDo()` und `getStepsDone()` zur Verfügung, welche die Zahl der insgesamt auszuführenden bzw. der davon schon abgeschlossenen Teilberechnungen zurückgeben.

Eine partielle Implementation des `GGCriticalPairsGenerator`-Interfaces bietet die Klasse `AbstractCriticalPairsGenerator`. Sie stellt die beiden Methoden zur Verwaltung der Listener zur Verfügung, ebenso wie die Methode `fireCPGeneratorEvent()`, die einen Event an alle Listener propagiert. Des Weiteren besitzt die Klasse eine Variable für die kritischen Paare und `setCriticalPairs()` und `getCriticalPairs()`, um auf sie zuzugreifen. Abgeleitete Klassen müssen also nur noch `init()`, `start()` und `stop()` implementieren.

5.5.1.6 `GGParser`

Der Parser ist im Gegensatz zum `GGCriticalPairsGenerator` nicht asynchron ausgelegt. Zum einen liegt das daran, dass die in GenGEd editierten Diagramme gewöhnlich nicht sehr groß sind, dafür aber relativ viele verschiedene Typen enthalten, was eine schnelle Parsierung ermöglicht. Die Angabe einer Layerfunktion und die Analyse der kritischen Paare beschleunigen den Vorgang zusätzlich. Zum anderen ist die Anwendung der Parsierung auf das Überprüfen von in der Editierung befindlichen Diagramme ausgelegt. Da der Benutzer während der Parsierung eines visuellen Satzes mit GenGEd ohnehin nichts anderes tun wird, scheint der Mehraufwand für die Implementierung einer asynchronen Parsierung nicht gerechtfertigt.

Nach der Initialisierung des Parsers mit einer Parsierungsspezifikation (`init()`) ist er, so die Methode `initCheck()` dies anzeigt, bereit, visuelle Sätze auf Zugehörigkeit zur visuellen Sprache zu prüfen. Der Methode `parse()` wird zu diesem Zweck eine Algebra übergeben und sie liefert bei positivem Bescheid *true*, sonst *false* zurück.

5.5.1.7 `GGFactory`

Die Klasse `GGFactory` stellt ein Factory (Factory-Designpattern) und damit ein Mittel zur Abstraktion von konkreten Implementationen von Interfaces dar. Alle betroffenen Interfaces haben mit Transformation und Parsierung zu tun und alle werden von Klassen implementiert, die eine Anbindung an Agg realisieren. `GGFactory` stellt Methoden zur Verfügung, die diese Klassen instanziieren und die Objekte als Instanzen der entsprechenden Interfaces zurückgeben. Die Anbindung eines neuen Systems zur Transformation und Parsierung erfolgt, abgesehen von der Implementation einer Reihe von Klassen, nur durch die Änderung der `GGFactory`-Methoden.

Die Klasse ist als Singleton (Singleton-Designpattern) konzipiert, die globale Instanz liefert die Methode `getDefault()` zurück. Die Semantik der `create*()`-Methoden wurde allgemein schon erklärt, es genügt also zu erwähnen, Objekte welcher Interfaces kreiert werden: Dies sind `GGAlgebraTransformer`, `GGCriticalPairsGenerator`, `GGLayerFunctionChecker` und `GGParser`. Außerdem liefert `createEmptyCriticalPairs()` ein uninitialisiertes `GGCriticalPairs`-Objekt zurück. Diese Methode wird ausschließlich beim Laden der XML-Dateien benötigt, da dabei (siehe dazu Abschnitt 5.8) schon eine Instanz des zu ladenenden Objekts vorhanden sein muss.

5.5.2 Agg-Anbindung

Für die Anbindung von AGG zur Parsierung, müssen die Interfaces `GGCriticalPairs`, `GGLayerFunctionChecker`, `GGCriticalPairsGenerator` und `GGParser` implementiert werden. Dank der Klasse `AGGGrammarConverter` (Abschnitt 5.4.2.3) stehen alle benötigten Methoden zur Konvertierung der GENGED- in AGG-Strukturen zur Verfügung. Ebenso wie bei der Transformation können auch bei der Parsierung die GENGED-Objekte in AGG-Instanzen umgewandelt werden, um dann AGG die eigentliche Arbeit zu überlassen. Die Aufgabe ist sogar leichter, da in keinem Fall eine Rückkonvertierung vonnöten ist.

5.5.2.1 AGGCriticalPairs

Ein `AGGCriticalPairs`-Objekt enthält nur eine Referenz auf eine Instanz der AGG-Klasse `ExcludePairContainer`. Diese wird dem Konstruktor übergeben und ist danach mittels `getPairContainer()` erreichbar. Auf die Implementation des `XMLObject`-Interfaces wird in Abschnitt 5.8 eingegangen. Ansonsten besitzt die Klasse keine weiteren Methoden.

5.5.2.2 AGGLayerFunctionChecker

`AGGLayerFunctionChecker` implementiert die von `GGLayerFunctionChecker` vorgesehenen Methoden `init()` und `check()`. Erstere kreiert einen `AGGGrammarConverter` und konvertiert die Grammatik. `check()` lässt die Layerfunktion in ein Objekt der AGG-Klasse `LayerFunction` übersetzen, welche mit der Methode `checkLayer()` genau die benötigte Funktionalität zur Verfügung stellt. Deren Rückgabewert wird auch von `check()` zurückgeliefert, wenn nicht schon die Konvertierung fehlschlug; dann ist das Resultat *false*.

5.5.2.3 AGGCriticalPairsGenerator

`AGGCriticalPairsGenerator` ist von `AbstractCriticalPairsGenerator` abgeleitet. In den `init()`-Methoden werden Grammatik und Layerfunktion in die entsprechenden AGG-Strukturen konvertiert, in `start()` muss lediglich die Analyse der kritischen Paare angestoßen werden, die auch in AGG asynchron implementiert ist, und `stop()` stoppt sie.

`AGGCriticalPairsGenerator` implementiert das AGG-Interface `ParserEventListener`, über welches der Benachrichtigungsmechanismus realisiert ist. Zu den eintreffenden `ParserEvent`-Objekten werden entsprechende `CPGeneratorEvent`-Instanzen kreiert und an die Listener des Generators gesendet.

5.5.2.4 `AGGParser`

In der `init()`-Methode der Klasse `AGGParser` werden die in der Parsierungsspezifikation enthaltenen Daten in AGG-Objekte umgewandelt und alle für die Parsierung benötigten Hilfsstrukturen angelegt. `parse()` muss lediglich die übergebene Algebra in einen Graphen konvertieren, einen AGG-Parser (`Parser`) kreieren und das Ergebnis dessen `parse()`-Methode zurückliefern.

5.6 Simulation

Zur Simulation wird AGG nur indirekt, über die Transformation (Abschnitt 5.4), bemüht. Alle benötigten Strukturen sind vollständig in GENGED implementiert. Der Abschnitt ist in zwei Unterabschnitte geteilt: Der erste beschreibt die allgemeineren Klassen, der zweite erläutert die Implementationen der verschiedenen Simulationsausdrücke.

5.6.1 Allgemeine Schnittstelle

Die in Abschnitt 2.4 beschriebenen Konzepte sind fast eins zu eins in Klassen umgesetzt. Eine Simulationsspezifikation wird durch die Klasse `SimulationSpecification` repräsentiert. Sie enthält neben einer Simulationsgrammatik die Simulationsschritte (`SimulationStep`), die ihrerseits jeweils einen Simulationsausdruck (Interface `SimulationExpression`) besitzen.

Zur Ausführung der Simulation selbst werden noch weitere Klassen benötigt. Das sind einerseits `ParameterAssignment` und `ParameterScope` für die Verwaltung der Parameter und andererseits `SEEvaluator` zur Auswertung der Simulationsausdrücke. Einen Überblick über die Klassenstruktur gibt Abbildung 5.9.

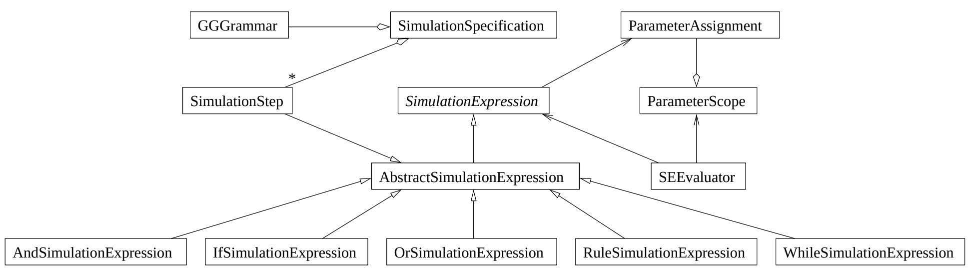

Abbildung 5.9: Struktur der Simulationsklassen.

5.6.1.1 `ParameterAssignment`

Ein `ParameterAssignment` kann Parameter (`GGParameter`) und Werte für diese verwalten. Nach der Konstruktion ist ein solches Objekt gewöhnlich leer, es sei denn, es wird eine `Collection` von `GGParameter`-Objekten übergeben – dann beinhaltet das `ParameterAssignment` diese Parameter, sie haben aber keinen Wert – oder ein anderes `ParameterAssignment`-Objekt – in dem Fall wird dieses kopiert. Der Wert für einen Parameter darf ein beliebiges `Object` sein.

Neue Parameter können mittels `addParameter()` (ein einzelner) oder `addParameters()` (eine `Collection`) hinzugefügt und durch `removeParameter()` (ein einzelner), `removeAllParameters()` (alle Parameter) und `removeUnsetParameters()` (nur die Parameter ohne Wert) entfernt werden. Ob das `ParameterAssignment` einen bestimmten Parameter enthält, prüft `containsParameter()`, `getParameter()` sucht einen Parameter mit einem gegebenen Namen. `getParameters()` liefert eine `Collection` aller, `getSetParameters()` der gesetzten und `getUnsetParameters()` der Parameter ohne Wert.

`setParameterValue()` setzt und `unsetParameterValue()` enfernt den Wert eines Parameters und `unsetAllParameterValues()` löscht gleich alle Parameterwerte. Die Methode `getParameterValue()` bietet Zugriff auf einen Wert.

5.6.1.2 `ParameterScope`

`ParameterScope` ist für die Auswertung von Simulationsausdrücken von Bedeutung. Scopes definieren den Gültigkeitsbereich und den Wert eines Parameters. Sie sind hierarchisch organisiert, ein Scope kann einen übergeordneten Scope besitzen. Als Parameterwerte kommen Objekte beliebigen Typs infrage. Ist ein solches Objekt allerdings wieder ein Parameter, so entspricht der tatsächliche Wert dem Wert des letzteren im übergeordneten Scope, oder er besitzt keinen, wenn kein übergeordneter Scope existiert.

Intern hält ein `ParameterScope` eine Referenz auf den übergeordneten Scope und ein `ParameterAssignment` für die Verwaltung der Werte. Dies sind auch die optionalen Argumente des Konstruktors – übergeordneter Scope und initiale Parameterbelegung. `setParameterValue()` setzt den Wert eines Parameters im `ParameterAssignment` und `unsetParameterValue()` enfernt ihn daraus samt dem Parameter selbst. `getParameterValue()` ermittelt den tatsächlichen Parameterwert: Ist der Parameter nicht im `ParameterAssignment` wird im übergeordneten Scope gesucht, anderenfalls handelt es sich um den tatsächlichen Wert oder um eine Referenz auf einen anderen Parameter, dessen Wert ebenfalls im übergeordneten Scope gesucht wird.

5.6.1.3 `SimulationExpression`

`SimulationExpression` ist das Interface für einen Simulationsausdruck. Es erweitert `Cloneable`, `GGObservable` und `XMLObject`. Jeder Ausdruck hat einen Namen, den `getName()` liefert, für manche Ausdrücke lässt er sich auch setzen (`setName()`).

Simulationsausdrücke bilden Bäume, weshalb einige mehrere Methoden zur Verwaltung ihrer Struktur zu implementieren sind. Dies sind `setParentExpression()` zum Setzen des übergeordneten Ausdrucks und `getParentExpression()` um auf diesen zuzugreifen; `getRootExpression()` liefert die Wurzel des Baums und `getSubExpressions()` eine `Collection` der Unterausdrücke eines Simulationsausdrucks.

Die Methode `getParameterAssignment()` gibt ein optional vorhandenes `ParameterAssignment`-Objekt zurück, das gleichermaßen für die Definition der Parameter des Ausdrucks wie für deren Belegung mit Werten verwendet wird.

Schließlich ist für die Auswertung der Simulationsausdrücke `evaluate()` zu implementieren, welchem neben einem `ParameterScope` ein `SEEvaluator`-Objekt (Abschnitt 5.6.1.7) zu übergeben ist.

5.6.1.4 `AbstractSimulationExpression`

`AbstractSimulationExpression` implementiert `SimulationExpression` partiell, oder um präzise zu sein, alle Methoden mit Außnahme von `evaluate()`. Dazu verwaltet die Klasse einen Namen, Referenzen auf den übergeordneten Ausdruck und auf ein `ParameterAssignment` sowie einen `Vector` der Unterausdrücke. Um letztere hinzufügen, entfernen oder auf sie zugreifen zu können, werden eine Reihe von Methoden zur Verfügung gestellt, die hier nicht aufgezählt werden sollen. Wichtig ist nur, dass abgeleitete Klassen auch konsequent ihre Unterausdrücke mit Hilfe dieser Methoden bekannt machen und auch wieder abmelden, damit etwa `getSubExpressions()` immer das richtige Ergebnis liefert.

5.6.1.5 `SimulationStep`

Um die Flexibilität zu erhöhen und zum Beispiel zuzulassen, dass später Simulationsschritte zu größeren Einheiten zusammengeschaltet werden können, implementiert `SimulationStep` das `SimulationExpression`-Interface. Durch Ableitung von `AbstractSimulationExpression` erübrigt sich die Implementierung aller Methoden des Interfaces bis auf `evaluate()`. Diese wertet, so vorhanden, den Simulationausdruck des Schritts aus und liefert das Ergebnis zurück, andernfalls *true*.

Zusätzlich stellt die Klasse die Methoden `setExpression()` zum Setzen des Ausdrucks und `getExpression()`, um auf ihn zuzugreifen, zur Verfügung. Die Parameter des Simulationsschritts werden direkt im `ParameterAssignment` verwaltet.

5.6.1.6 `SimulationSpecification`

Ein `SimulationSpecification`-Objekt enthält eine Simulationsgrammatik und einen `Vector` von Simulationsschritten. Die Grammatik wird von `setSimulationGrammar()` und `getSimulationGrammar()` gesetzt bzw. zurückgegeben. Zur Verwaltung der Simulationsschritte gibt es zehn Methoden, auf deren Aufzählung jedoch verzichtet werden soll.

5.6.1.7 `SEEvaluator`

Ein `SEEvaluator` wird als Hilfsobjekt bei der Auswertung eines Simulationsausdrucks benötigt. Es kennt den zu simulierenden visuellen Satz sowie einen `GGAlgebraTransformer` und ein `GGMatchCompletionInfo`, die für die Transformationen verwendet werden sollen. Für diese Variablen gibt es entsprechende `set*()`- und `get*()`-Methoden.

Die Auswertung eines Simulationsausdrucks erfolgt durch Aufruf einer der `evaluate()`-Methoden. Der einen wird zusätzlich ein `ParameterScope` übergeben, der verwendet werden soll. Die andere kreiert einen neuen Scope, der den zuvor verwendeten Scope als übergeordneten Scope verwendet und das `ParameterAssignment` des Ausdrucks übernimmt. Mit dem `SEEvaluator`-Objekt und dem Scope wird in beiden Fällen die `evaluate()`-Methode des Simulationsausdrucks aufgerufen. Hat dieser Unterausdrücke auszuwerten, so wird dazu wieder der `SEEvaluator` bemüht. Die Ausdrücke und die Scopes werden in Stacks verwaltet, entsprechende Zugriffsmethoden zur Verfügung gestellt.

5.6.2 Simulationsausdrücke

Von den in Abschnitt 2.4.1 eingeführten Ausdrücken sind **rule**(), **if then else fi**, **while do done**, **AND**, **ANDC**, **OR** und **ORC** implementiert, die anderen können durch diese ausgedrückt werden. Für jeden Ausdruckstyp gibt es eine eigene Klasse, nur **AND** und **ANDC** sowie **OR** und **ORC** teilen sich jeweils eine. Alle Klassen sind direkt von `AbstractSimulationExpression` abgeleitet.

5.6.2.1 `AndSimulationExpression`, `OrSimulationExpression`

Die Klassen `AndSimulationExpression` und `OrSimulationExpression` sind bis auf ihren Namen und die `evaluate()`-Methode fast identisch. Beide besitzen eine boolesche Variable, die angibt, ob die Auswertung konditional (**ANDC**, **ORC**) oder unkonditional (**AND**, **OR**) erfolgen soll. Diese kann mittels `setConditionalEvaluation()` gesetzt und mittels `getConditionalEvaluation()` abgefragt werden.

Die Auswertung erfolgt entsprechend der Konzepte 2.4.1: Die `evaluate()`-Methoden von `AndSimulationExpression` und `OrSimulationExpression` liefern bei einem leeren Ausdruck *true* bzw. *false* zurück, gibt es genau einen Unterausdruck, so wird dieser ausgewertet. Bei konditionaler Auswertung und mehreren Unterausdrücken werden diese einer nach dem anderen ausgewertet, bis ein Zwischenergebnis *false* respektive *true* ist. Im unkonditionalen Fall, werden immer alle Unterausdrücke evaluiert und dann das Gesamtergebnis berechnet.

5.6.2.2 `IfSimulationExpression`

Ein **if then else fi**-Ausdruck besitzt genau drei Unterausdrücke. Für jeden gibt es eine `set*()`- und eine `get*()`-Methode. Ausgewertet wird strikt, das heißt, liefert die Auswertung des ersten Unterausdrucks *true*, so wird der zweite, anderenfalls der dritte Unterausdruck Audruck ausgewertet.

5.6.2.3 `RuleSimulationExpression`

Ein `RuleSimulationExpression`-Objekt hält eine Referenz auf eine Regel (`GGRule`), entsprechende Methoden `setRule()` und `getRule()` sind implementiert. Neben `SimulationStep`, ist diese Klasse die einzige, die ein `ParameterAssignment` verwendet. Dieses enthält die Parameter der Regel und ihre Belegung entweder mit Datentypwerten oder Parametern eines übergeordneten Ausdrucks.

Für die Auswertung müssen alle Parameter belegt sein. `evaluate()` holt sich vom `SEEvaluator` die zu simulierende Algebra, das `GGMatchCompletionInfo` und den `GGAlgebraTransformer`, kreiert einen Match der Regel des Ausdrucks in die Algebra und wendet die Regel an. Konnte die Regel erfolgreich angewandt werden, wird *true* zurückgegeben, sonst *false*.

5.6.2.4 `WhileSimulationExpression`

Ein **while do done**-Ausdruck besitzt genau zwei Unterausdrücke. Für beide gibt es eine `set*()`- und eine `get*()`-Methode. Solange der erste Unterausdruck zu *true* ausgewertet wird, wird auch der zweite Unterausdruck evaluiert. Das Auswertungsergebnis ist *true*, wenn die Schleife mindestens einmal durchlaufen wurde, ansonsten *false*.

5.7 VL-Spezifikation

VL-Spezifikationen werden durch die Klasse `VLSpecification` repräsentiert. Gemäß der Konzepte (Abschnitt 2.5) verwaltet sie eine Syntaxsignatur, eine

Syntaxgrammatik, eine Parsierungspezifikation und beliebig viele Simulationsspezifikationen (Abb. 5.10). Für letztere stehen die üblichen Methoden (`add*()`, `remove*()`, ...) zur Verfügung, für die anderen sind `get*()`- und `set*()`-Methoden vorgesehen. Zu bemerken bleibt, dass das Setzen einer neuen Syntaxsignatur die Syntaxgrammatik und Parsierungs- und Simulationsspezifikationen auf *null* setzt bzw. entfernt.

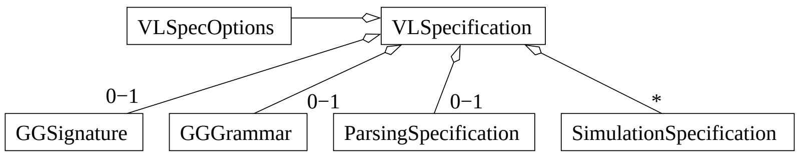

Abbildung 5.10: Klassen der VL-Spezifikation.

Die Hilfsklasse `VLSpecOptions` sammelt die Optionen einer `VLSpecification`. Zurzeit sind dies nur Parameter, die das Speichern der VL-Spezifikation betreffen. Für Syntaxsignatur sowie Syntax-, Parsierungs- und Simulationsgrammatiken kann festgelegt werden, ob sie in die Datei der VL-Spezifikation hineingeschrieben oder nur per Dateinamen referenziert werden sollen. Für die Grammatiken gibt es für den ersten Fall drei Unterfälle: Es ist möglich, alle Informationen zu schreiben oder nur wenige oder gar keine grafische Daten.

5.8 Persistenz

Alle Daten, dass heißt Alphabet, Diagramme, Grammatiken und VL-Spezifikationen, werden in XML-Dateien abgespeichert. Wie solche Dateien genau aussehen müssen, beschreiben sogenannte DTDs (Document Type Definitions). Die ersten Unterabschnitte geben einen Überblick über XML und DTDs im Allgemeinen und erläutern die speziellen Formate, die für GenGEd verwendet werden. Es folgt ein Abschnitt über Xerces, eine Bibliothek, die Funktionalität bereitstellt, um komfortabel mit XML-Dateien umzugehen. Abschließend wird die Implementierung beschrieben.

5.8.1 XML

XML, die Extensible Markup Language ([Con01]), wurde von der unter der Schirmherrschaft des World Wide Web Consortium (W3C) gebildeten XML Working Group entwickelt. Ziel war es, ein Format zu schaffen, mit dessen Hilfe Dokumente einfach zwischen verschiedenen Plattformen austauschbar sind. Insbesondere können XML-Dokumente ob ihres einfachen Aufbaus relativ leicht von Applikationen erzeugt und gelesen werden, zudem sind sie auch menschenlesbar und besitzen eine nachvollziehbare Struktur.

Diese Struktur ist, abgesehen von Details, ein Baum von XML-Elementen. Ein solches Element kann Attribute, Unterelemente und einen Inhalt besitzen. Eingerahmt wird es durch ein Start- und ein Endtag, oder besteht, wenn Unterelemente und Inhalt fehlen, aus einem verkürzten Tag. Dieses oder im ersten Fall das Starttag enthalten die Attribute während sich zwischen Start- und Endtag der Inhalt und die Unterelemente befinden.

Beispiel 5.8.1 *Die Abbildung 5.11 zeigt ein Beispiel eines XML-Dokuments, das einen Graphen beschreibt. Der Name des Wurzeltags ist* `Graph`*, es besitzt ein Attribut* `name` *mit dem Wert* `"Startgraph"` *und jeweils drei Unterelemente* `Node` *und* `Edge`*, welche die verkürzten Tags verwenden.*

```
<Graph name="Startgraph">
  <Node ID="1" />
  <Node ID="2" />
  <Node ID="3" />
  <Edge source="1" target="2" />
  <Edge source="2" target="3" />
  <Edge source="3" target="1" />
</Graph>
```

Abbildung 5.11: Beispiel eines XML-Dokuments.

5.8.2 DTDs

Eine DTD (Document Type Definition) zeichnet eine Teilmenge der wohlgeformten XML-Dokumente mit einer bestimmten Struktur aus. Es wird definiert, welche Typen von Elementen in einem Dokument vorkommen dürfen, welchen Typ das Wurzelelement besitzt, welche Attribute und Unterelemente ein Element haben darf und es wird sogar die Art der Attribute präzisiert.

DTDs werden in XML-Dokumenten angegeben, die verwendete Syntax weicht allerdings von der ansonsten verwendeten ab. Anstatt diese hier einzuführen, seien Interessierte auf [Con01] verwiesen. Die in den folgenden Unterabschnitten erläuterten DTDs werden als Klassendiagramme dargestellt, was zudem auch die Übersichtlichkeit erhöht. Dabei stellen die Klassen die Elementtypen dar, die Attribute der Klassen die Elementattribute, und Kompositionen visualisieren die Unterelementbeziehungen.

5.8.2.1 GTS DTD

Die hier vorgestellte DTD für Graphtransformationssysteme (GTS DTD) wurde an der TU Berlin entworfen ([GXL01]) und wird noch immer weiterentwickelt, bis die internationalen Bemühungen, ein gemeinsames Format zu finden, von Erfolg gekrönt sind. Der Autor selbst nahm einige für GenGEd notwendige Änderungen vor; es sei auf die im GenGEd-Paket enthaltene Datei `genged/doc/gts.dtd` verwiesen, welche die hier beschriebene und in GenGEd implementierte Variante darstellt.

Die GTS DTD trifft keine Festlegungen, was das grafische Layout der Graphen anbelangt – zwar gibt es Elemente, die dafür angedacht sind, doch werden diese nicht ausspezifiziert. Der auf diesen folgende Abschnitt geht näher auf die für GenGEd relevanten Layoutelemente ein.

Verschiedene Elemente dürfen ein Unterelement `TaggedValue` besitzen, auf das hier nicht eingegangen wird, da es ausschließlich für zusätzliche Daten vorgesehen ist.

Der Übersichtlichkeit wegen wird die DTD in vier Abbildungen dargestellt, angefangen mit der groben Struktur (Abb. 5.12):

`GraphTransformationSystem`: Repräsentiert ein Graphtransformationssystem. Neben einem Namen ist eine optionale ID vorgesehen. Unterelemente sind `Types`, in dem die Knoten-, Kanten- und Attributtypen deklariert werden, ein `Graph` (der Startgraph) und eine beliebige Anzahl von `Rule`-Elementen für die Regeln der Graphgrammatik.

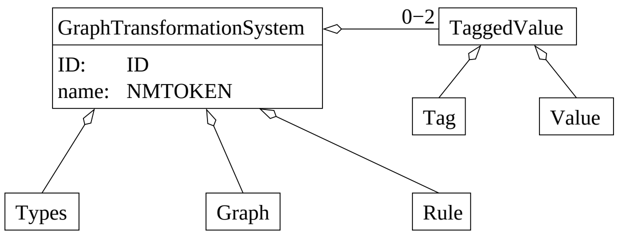

Abbildung 5.12: Die Grobstruktur der GTS DTD.

Abbildung 5.13 verfeinert das `Types`-Element:

`Types`: Enthält Knoten- und Kantentypen in `NodeType`- und `EdgeType`-Elementen.

`NodeType`: Repräsentiert einen Knotentypen. Dieser besitzt eine ID und einen Namen sowie ein optionales Unterelement `NodeTypeLayout` für das Layout des Typs und beliebig viele `AttrType`-Elemente für die Knotenattributtypen.

`EdgeType`: Repräsentiert einen Kantentypen. Dieser besitzt eine ID, einen Namen und Listen von IDs der Quell- und Zielknotentypen (je eine ID bei einfachen Kanten, mehrere für Hyperkanten) sowie ein optionales Unterelement `EdgeTypeLayout` für das Layout des Typs und beliebig viele `AttrType`-

Elemente für die Kantenattributtypen. Zudem sind `AttachName`-Elemente zugelassen.

`AttrType`: Repräsentiert einen Attributtypen. Dieser besitzt eine ID, einen Namen (`attrname`) und einen Namen für den Typ des Datums selbst (`typename`). Auch hier ist ein optionales Layoutelement, `AttrTypeLayout`, vorgesehen.

`AttachName`: Enthält ein Attribut für einen Namen.

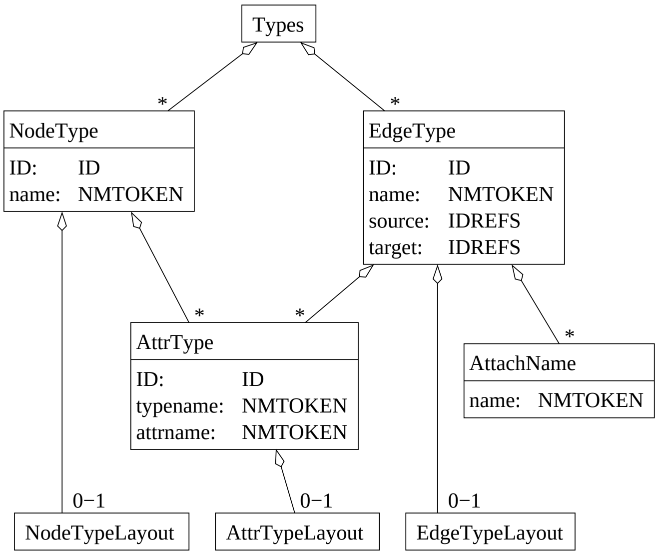

Abbildung 5.13: Der Typ-Anteil der GTS DTD.

Abbildung 5.14 stellt das Element `Graph` und zugehörige Elemente dar:

`Graph`: Repräsentiert einen Graphen. Dieser besitzt eine ID und einen Namen sowie `Node`- und `Edge`-Unterelemente für Knoten und Kanten.

`Node`: Repräsentiert einen Knoten. Dieser besitzt eine ID und einen Typ, der durch dessen ID referenziert wird, sowie ein optionales Unterelement `NodeLayout` für das Layout des Knoten und entsprechend seinem Typ `Attribute`-Elemente für die Attribute.

`Edge`: Repräsentiert eine Kante. Diese besitzt eine ID, einen Typ, der durch dessen ID referenziert wird, und Listen von IDs der Quell- und Zielknoten sowie ein optionales Unterelement `EdgeLayout` für das Layout der Kante und entsprechend ihres Typs `Attribute`-Elemente für die Attribute.

`Attribute`: Repräsentiert ein Attribut. Dieses besitzt einen Namen und eine ID-Referenz auf seinen Typ sowie ein optionales Unterelement `AttrLayout` für das Layout des Attributs und ein Element für den Wert des Attributs.

`AttrVal`: Repräsentiert den Wert eines Attributs. Das Element enthält je nach Art des Werts ein Unterelement `PrimitiveValue` für einen primitiven Typ, `ObjectRef` für eine Referenz auf ein Objekt, `Object` für ein Objekt selbst oder `Expression` für einen Ausdruck.

`PrimitiveValue`: Repräsentiert den Wert eines primitiven Datentyps. Besitzt eine ID sowie Attribute für den Namen des Typs und den Wert.

`ObjectRef`: Repräsentiert den Wert eines komplexen Typs, gegeben durch die ID eines Objekts.

`Object`: Repräsentiert den Wert eines komplexen Typs, gegeben durch ein Objekt selbst. Das Element besitzt eine ID und ein Attribut für den Namen der Klasse des Objekts sowie beliebig viele `Attr`-Unterelemente, die nicht näher spezifiziert sind.

`Expression`: Repräsentiert einen Attributwert gegeben durch einen Ausdruck. Dieser ist im Attribut `value` abgelegt.

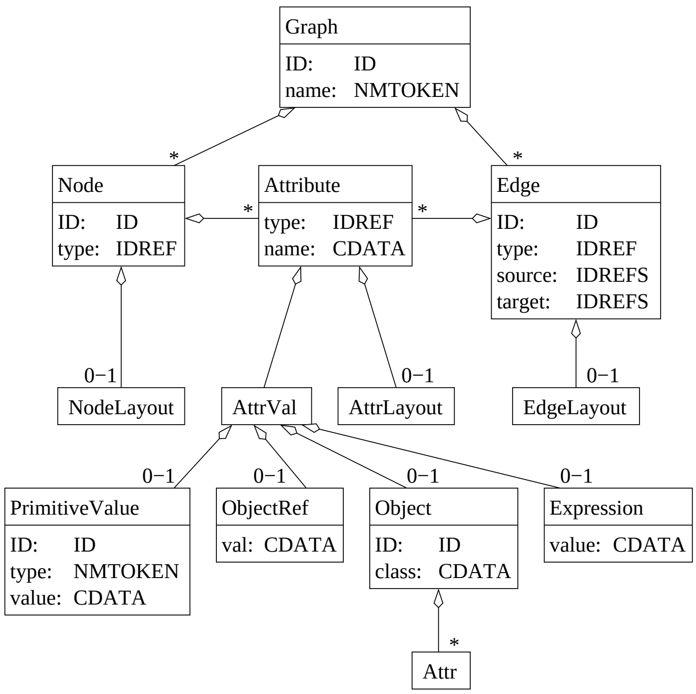

Abbildung 5.14: Der Graph-Anteil der GTS DTD.

Abbildung 5.15 zeigt das Element `Rule` und zugehörige Elemente:

`Rule`: Repräsentiert eine Regel. Diese besitzt eine ID und einen Namen sowie beliebig viele `Parameter`- und `Variable`-Unterelemente für die Parameter und

Variablen der Regel. Für linke und rechte Regelseite gibt es jeweils ein `Graph`-, für den Regelmorphismus ein `Morphism`- und optional ein `ApplCondition`-Element mit den Anwendungsbedingungen.

`Parameter`: Repräsentiert einen Regelparameter. Dieser ist gegeben durch einen Namen, den Namen des Parametertyps und die Art des Parameters – Eingabe- und/oder Ausgabeparameter.

`Variable`: Repräsentiert eine Variable. Diese ist gegeben durch einen Namen und einen Typ.

`Morphism`: Repräsentiert einen Morphismus, jedoch ohne Domain und Codomain. Das Element besitzt ein Attribut für den Morphismusnamen und `Mapping`-Unterelemente für die Mappings.

`Mapping`: Repräsentiert ein Morphismusmapping. Attribute sind die IDs von Original und Bild.

`ApplCondition`: Enthält die Anwendungsbedingungen. Dies können zum einen `NAC`-Elemente für die negativen Anwendungsbedingungen, zum anderen `AttrCondition`-Elemente für Attributbedingungen sein.

`NAC`: Repräsentiert eine NAC. Diese besitzt einen Namen und je ein `Graph`- und `Morphism`-Unterelement für Codomain und Morphismus der NAC.

`AttrCondition`: Repräsentiert eine Attributbedingung, gespeichert im `value`-Attribut.

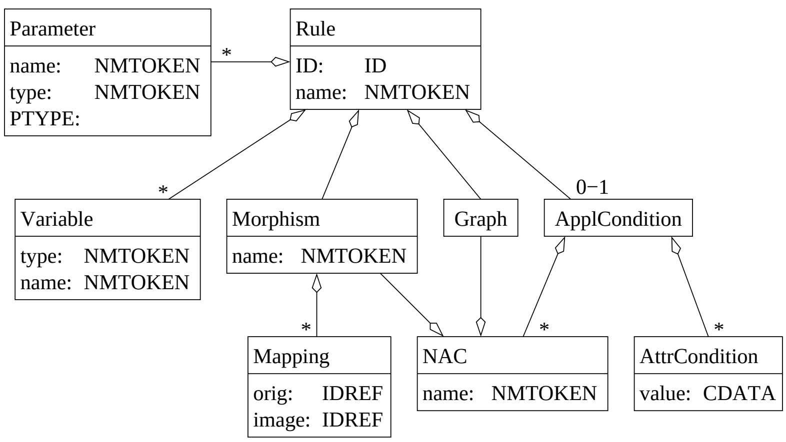

Abbildung 5.15: Der Regel-Anteil der GTS DTD.

5.8.2.2 DTD für das Layout

Die DTD für das Layout ist an jene angelehnt, die für AGG verwendet wird. Es ist jedoch nur ein geringer Grad an Kompatibilität möglich, da AGG nur einige wenige grafische Attribute, etwa Farbe und Form von Knoten und Kanten, bietet, während GENGED ein nahezu beliebiges Aussehen für seine Symbole ermöglicht. Die DTD ist weder endgültig – es gibt Bestrebungen, eine einheitliche DTD für Grafiken zu definieren – noch vollständig. Die im Folgenden erwähnten Elemente `SymbolObject`, `Operation` und `DatatypeObject` werden nicht verfeinert. Sie stehen für Objekte der gleichnamigen GENGED-Klassen und enthalten alle Informationen, um die Objekte wieder vollständig rekonstruieren zu können.

Die DTD ist in Abbildung 5.16 dargestellt:

`NodeTypeLayout`: Enthält ein `SymbolObject`- und ein `Operation`-Element für ein grafisches Symbol und dessen interne Operation.

`NodeLayout`: Enthält ein `SymbolObject`-Element für eine Symbolinstanz.

`EdgeTypeLayout`: Enthält ein `Operation`-Element, das einen Link repräsentiert.

`EdgeLayout`: Wird nicht benötigt.

`AttrTypeLayout`: Enthält ein `DatatypeObject`- und zwei `Operation`-Elemente für den Datentypen und dessen grafische Repräsentation sowie seine interne Operation und den Attributierungslink.

`AttrLayout`: Enthält ein `Datatype`-Objekt für eine Datentypinstanz.

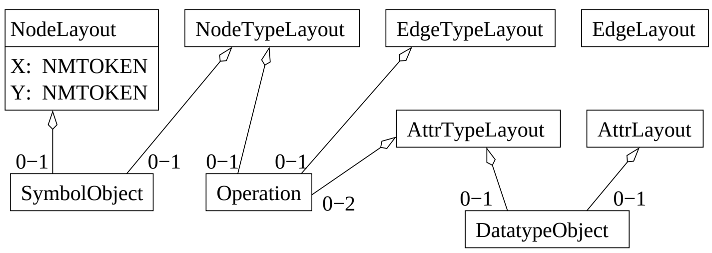

Abbildung 5.16: Die Layout DTD.

5.8.2.3 DTD für VL-Spezifikationen

Die DTD für VL-Spezifikationen wurde vom Autor entwickelt. Sie orientiert sich unmittelbar an der Hierarchie der Klassen für die VL-Spezifikation. Der Übersichtlichkeit halber wird die DTD in drei Abbildungen dargestellt. Einige Elemente sind schon in der GTS DTD definiert (Abschnitt 5.8.2.1) und werden deshalb an dieser Stelle nicht weiter erläutert.

Wir beginnen mit der groben Struktur der DTD (Abb. 5.17):

`VLSpecification`: Repräsentiert eine VL-Spezifikation. Neben einem Namen besitzt das Element ein Unterelement `VLSpecOptions`, welches die Daten der gleichnamigen Klasse enthält sowie optional Elemente `SyntaxAlphabet` für das Alphabet, `SyntaxGrammar` für die Syntaxgrammatik, `ParsingSpecification` für die Parsierungsspezifikation und beliebig viele `SimulationSpecification`-Elemente für die Simulationsspezifikationen.

`VLSpecOptions`: Enthält Attribute, die angeben, ob und wie Alphabet, Syntax-, Parsierungs- und Simulationsgrammatiken in der Datei der VL-Spezifikation enthalten sind.

`SyntaxAlphabet`: Repräsentiert ein Alphabet. Das eigentliche Alphabet ist in einem Unterelement `GraphTransformationSystem` gespeichert oder es ist das Attribute `filename` gegeben, welches eine externe Datei referenziert.

`SyntaxGrammar`: Repräsentiert eine Syntaxgrammatik. Die eigentliche Grammatik ist in einem Unterelement `GraphTransformationSystem` gespeichert oder es ist das Attribute `filename` gegeben, welches eine externe Datei referenziert.

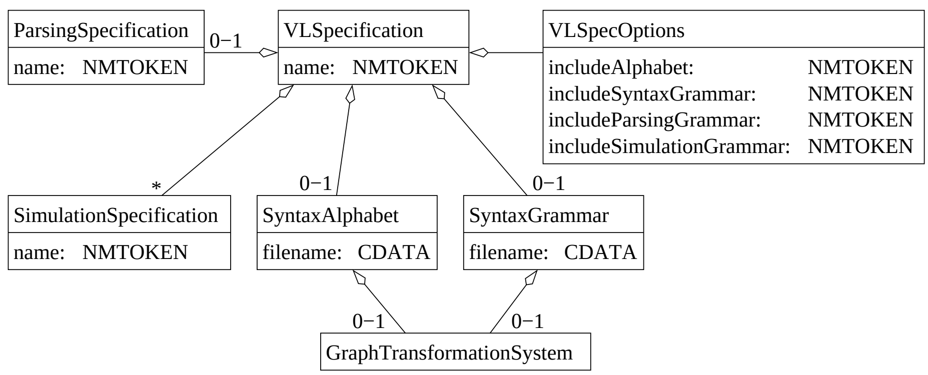

Abbildung 5.17: Die Grobstruktur der DTD für VL-Spezifikationen.

Abbildung 5.18 zeigt den Parsierungsanteil der DTD:

`ParsingSpecification`: Repräsentiert eine Parsierungsspezifikation. Sie besitzt neben einem Attribut, das angibt, ob die Layerfunktion verwendet werden soll, optionale Unterelemente `ParsingGrammar`, `LayerFunction` und `CriticalPairs` für die Parsierungsgrammatik, die Layerfunktion und die kritischen Paare.

`ParsingGrammar`: Repräsentiert eine Parsierungsgrammatik. Die eigentliche Grammatik ist in einem Unterelement `GraphTransformationSystem` gespeichert oder es ist das Attribute `filename` gegeben, welches eine externe Datei referenziert.

`CriticalPairs`: Enthält ein nicht weiter spezifiziertes Unterelement `PairContainer`, das die kritischen Paare beinhaltet.

`LayerFunction`: Repräsentiert eine Layerfunktion. In den Unterelementen `RuleLayers`, `CreationLayers` und `DeletionLayer` sind die Rule-, Creation- und Deletion-Layers gespeichert.

`RuleLayers`: Enthält für jede Regel ein `RuleLayer`-Element.

`CreationLayers`: Enthält für jede Sorte ein `CreationLayer`-Element.

`DeletionLayers`: Enthält für jede Sorte ein `DeletionLayer`-Element.

`RuleLayer`: Definiert das Layer einer Regel. Die Regel wird durch ihre ID referenziert, das Layer ist durch das Attribut `layer` gegeben.

`CreationLayer`: Definiert das Creation-Layer einer Sorte. Die Sorte wird durch die ID ihres korrespondierenden Knotentyps referenziert, das Layer ist durch das Attribut `layer` gegeben.

`DeletionLayer`: Definiert das Deletion-Layer einer Sorte. Die Sorte wird durch die ID ihres korrespondierenden Knotentyps referenziert, das Layer ist durch das Attribut `layer` gegeben.

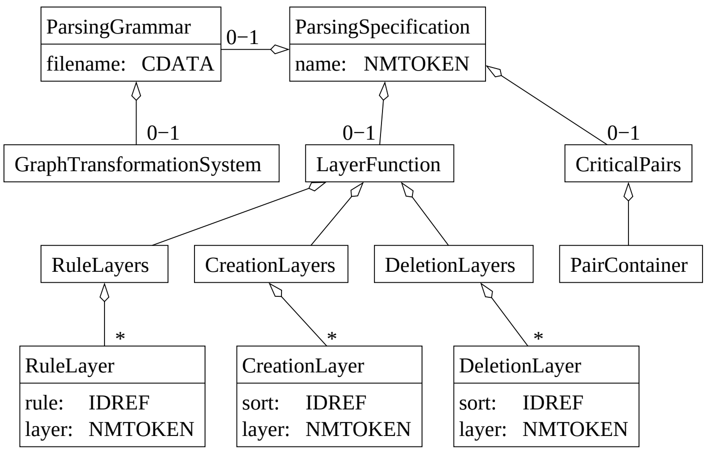

Abbildung 5.18: Der Parsierungsanteil der DTD für VL-Spezifikationen.

Abbildung 5.19 zeigt den Simulationsanteil der DTD:

`SimulationSpecification`: Repräsentiert eine Simulationsspezifikation. Sie besitzt neben einem Namen die Unterelemente `SimulationGrammar` mit der Simulationsgrammatik und `SimulationSteps` mit den Simulationsschritten.

`SimulationGrammar`: Repräsentiert eine Simulationsgrammatik. Die eigentliche Grammatik ist in einem Unterelement `GraphTransformationSystem` gespeichert oder es ist das Attribute `filename` gegeben, welches eine externe Datei referenziert.

`SimulationSteps`: Ist ein Container für `SimulationStep`-Elemente.

`SimulationStep`: Repräsentiert einen Simulationsschritt. Das Element besitzt ein Attribut `name` mit dem Namen des Schritts, ein Unterelement `Parameters` mit seinen Parametern und optional ein Unterelement `SimulationExpression`, welches den Simulationsausdruck des Schritts beinhaltet.

`Parameters`: Ist ein Container für `Parameter`-Elemente.

`SimulationExpression`: Repräsentiert einen Simulationsausdruck. Abhängig von dessen Typ besitzt das Element ein `AndExpression`-, `IfExpression`-, `NotExpression`-, `OrExpression`-, `RuleExpression`- oder `WhileExpression`-Element als Unterelement.

`AndExpression`: Repräsentiert abhängig vom Attribut `conditional` einen **AND**- oder **ANDC**-Ausdruck. Die Unterausdrücke sind durch `SimulationExpression`-Unterelemente gegeben.

`IfExpression`: Repräsentiert einen **if then else fi**-Ausdruck. Die Unterausdrücke sind durch Unterelemente `ConditionExpression`, `ThenExpression` und `ElseExpression` gegeben.

`NotExpression`: Repräsentiert einen **not**-Ausdruck. Der Unterausdruck ist durch ein `SimulationExpression`-Unterelement gegeben.

`OrExpression`: Repräsentiert abhängig vom Attribut `conditional` einen **OR**- oder **ORC**-Ausdruck. Die Unterausdrücke sind durch `SimulationExpression`-Unterelemente gegeben.

`RuleExpression`: Repräsentiert einen **rule**()-Ausdruck. Die Regel wird durch ihre ID referenziert, eine Parameterbelegung ist durch ein `ParameterAssignment`-Element gegeben.

`WhileExpression`: Repräsentiert einen **while do done**-Ausdruck. Die Unterausdrücke sind durch die Unterelemente `ConditionExpression` und `DoExpression` gegeben.

`ConditionExpression`: Enthält einen Simulationsausdruck im `SimulationExpression`-Unterelement.

`ThenExpression`: Enthält einen Simulationsausdruck im `SimulationExpression`-Unterelement.

`ElseExpression`: Enthält einen Simulationsausdruck im `SimulationExpression`-Unterelement.

`DoExpression`: Enthält einen Simulationsausdruck im `SimulationExpression`-Unterelement.

`ParameterAssignment`: Repräsentiert eine Parameterbelegung, die durch `ParameterValue`-Unterelemente gegeben ist.

`ParameterValue`: Das `name`-Attribut des Elements beinhaltet den Namen des Parameters. Ein `ParameterRef`-Unterelement verweist auf einen anderen Parameter, `Object` steht für einen konkreten Wert.

`ParameterRef`: Referenziert einen Parameter durch seinen im `name`-Attribut enthaltenen Namen.

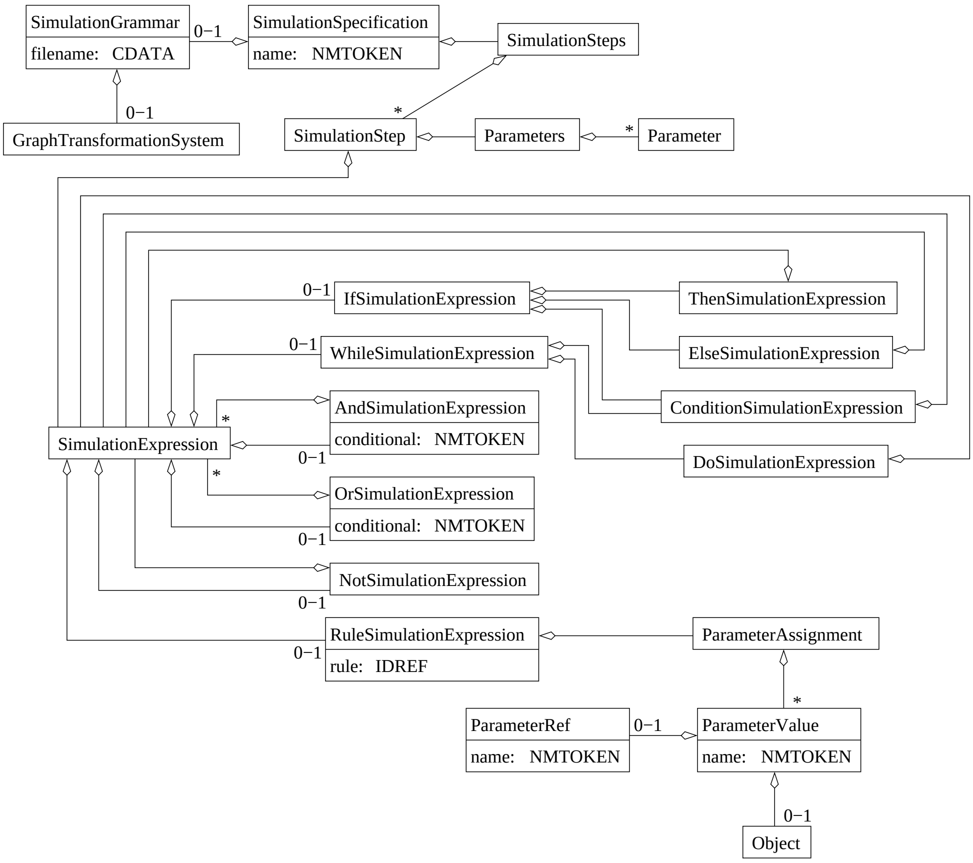

Abbildung 5.19: Der Simulationsanteil der DTD für VL-Spezifikationen.

5.8.3 Implementierung

5.8.3.1 Xerces

Der Xerces Parser ist im Rahmen des *Apache XML Projects* der *Apache Software Foundation* ([Fou01]) entstanden. Es gibt Implementationen in C++ und in Java, wobei letztere für GenGEd von Interesse ist. Xerces ist eigentlich nicht ein Parser, sondern ein Rahmenwerk für den Umgang mit XML-Dokumenten, welches mehrere Parser enthält.

Einer davon ist ein DOM-Parser (Document Object Model). DOM definiert eine Schnittstelle für die Handhabung von XML-Entitäten. Jede Entität wird durch eine eigene Klasse repräsentiert, die einen Zugriff auf deren Daten und auf Unterentitäten ermöglicht. So gibt es zum Beispiel Klassen für Dokumente, Elemente und für Attribute. Der DOM-Parser parsiert eine XML-Datei und generiert dazu das komplette Dokumentobjekt. Dieses kann nun traversiert, verändert und auch wieder in eine Datei geschrieben werden. Die Handhabung von XML-Daten vereinfacht sich auf diese Weise ungemein.

5.8.3.2 `XMLHelper`, `XMLObject`

Xerces stellt ein leicht zu handhabendes XML-Rahmenwerk zur Verfügung, doch bietet es direkt keine Funktionalität an, um das Speichern und Laden von Datenstrukturen zu vereinfachen. Diese Vereinfachung soll durch das Interface `XMLObject`, welches von Klassen, die speicher- und ladbare Objekte besitzen, zu implementieren ist, und die Klasse `XMLHelper` realisiert werden. Ursprünglich wurde dieses Rahmenwerk von Michael Matz für Agg entwickelt, doch im Laufe der Zeit vom Autor dieser Arbeit so stark verändert und erweitert, dass nur noch ein kleiner Teil des ursprünglichen Codes enhalten ist.

Das Interface `XMLObject` deklariert zwei Methoden, `XwriteObject()` und `XreadObject()`, die beide einen `XMLHelper` übernehmen, keinen Rückgabewert haben, aber im Fehlerfall eine `XMLIOException` werfen. Ein Aufruf der ersten Methode lässt ein Objekt mittels des `XMLHelper`-Objekts seine Daten schreiben, die zweite diese lesen. `XMLIOException` ist identisch wie `GGException` (Abschnitt 5.3.2.7) implementiert.

Die eigentliche Arbeit beim Abspeichern oder Laden eine Objekts einer konkreten Klasse wird in `XMLObject`-Methoden verrichtet, `XMLHelper` stellt dafür Hilfsfunktionen zur Verfügung. Im Endeffekt sogar so viele, dass vollständig vom Xerces-Rahmenwerk abstrahiert wird. Ein `XMLHelper`-Objekt bietet Zugriff auf ein XML-Dokument. Direkt nach der Konstruktion ist das Dokument bis auf das Wurzelelement leer. `read_from_xml()` liest ein XML-Dokument aus einer Datei, `write_to_xml()` schreibt das aktuelle Dokument in eine hinein.

Jederzeit ist ein XML-Element aus dem Elementbaum des Dokuments aktiv –

nach der Konstruktion oder dem Laden ist es das Wurzelelement. Methoden, die ein Attribut lesen oder ändern oder ein Unterelement suchen oder erzeugen, wirken immer auf das aktive Element. Zudem wird sich intern zum Zwecke der Traversierung eine Referenz auf ein Unterelement des aktiven Elements gemerkt. Wird ein Unterelement geöffnet (aktiviert) und später wieder geschlossen (`close()`) ist wieder das selbe Element wie zuvor aktiv, das Verhalten ist also Stack-artig.

Die Methode `openSubTag()` kreiiert ein Unterelement im aktiven Element und aktiviert es. `readSubTag()` sucht ein Unterelement mit einem bestimmten Namen im aktiven Element. Dabei beginnt die Suche bei dem Element, auf das die interne Referenz zum Traversieren weist – ist sie *null*, wird von vorn begonnen. Wurde ein Element mit dem gesuchten Namen gefunden, so wird es aktiviert.

Die verschiedenen `addAttr()`-Versionen setzen ein Attribut des aktiven Elements auf einen gegebenen Wert. Sie unterscheiden sich im Typ des Attributwerts – zur angenehmen Handhabung sind alle primitiven Typen berücksichtigt. `hasAttr()` überprüft, ob das aktive Element ein bestimmtes Attribut besitzt und `readAttr()` gibt dessen Wert zurück. Es gibt auch von `readAttr()` verschiedene Varianten, die neben dem Namen des Attributs als zweiten Parameter einen Defaultwert bekommen, der zurückgeliefert wird, falls das Element kein solches Attribut besitzt.

Theoretisch sollten die bisher vorgestellten Methoden schon genügen, doch gibt es noch eine Reihe weiterer. So zum Beispiel die `add*Tag()`-Methoden. Sie kreieren ein Unterelement mit dem übergebenen Namen und legen in diesem den mit dem zweiten Parameter bereitgestellten Wert ab. Handelt es sich um einen primitiven Typ, so wird nur ein `value`-Attribut angelegt, für `Point`, `Color` oder `Font` werden mehrere Attribute benötigt und `addObjectTag()`, welches ein beliebiges Objekt einer Klasse, die das `Serializable`-Interface implementiert, abspeichert, legt gar weitere Unterelemente an. Die `read*Tag()`-Methoden können derart gespeicherte Werte wieder zurückgewinnen.

Die Methode `addSubObject()` bekommt ein `XMLObject`-Objekt übergeben und ruft dessen `XwriteObject()`-Methode auf – wird optional ein Name übergeben, so kreiert die Methode zuvor ein neues Unterelement dieses Namens. `readSubObject()` fungiert dazu als Gegenstück beim Lesen. Statt dieser kann auch `instantiateSubObject()` verwendet werden. Sie bekommt kein entsprechendes Objekt, sondern eine Klasse oder einen Klassennamen und instantiiert sich das Objekt selbst – dazu wird geschaut, ob die Klasse eine statische `instantiate()`-Methode besitzt, wenn nicht wird der parameterlose Konstruktor verwendet.

Die Methode `addEncapsulatedSubObject()` funktioniert ganz ähnlich wie `addSubObject()`, doch schreibt sie zusätzlich ein Attribut `classname` mit dem Klassennamen des Objekts. Die Lesemethode `instantiateEncapsulatedSubObject()` benötigt deshalb nicht einmal einen Klassennamen als Parameter, sondern verwendet den im Attribut abgelegten. Nur erwähnt sei, dass es auch Methoden gibt, die gleich ganze Vektoren oder Aufzählungen von Objekten schreiben und lesen.

Die möglicherweise aufgekommene Frage, warum die bisher genannten mit `XMLObject`-Objekten arbeitenden Methoden alle ein `Sub` im Namen tragen, kann kurz beantwortet werden: Die gleichen Methoden gibt es nämlich nochmal ohne `Sub`. Der Unterschied liegt darin, dass diese zusätzlich mit IDs hantieren. Die Schreibmethoden schreiben ein `ID`-Attribut mit der ID des Objekts, die Lesemethoden lesen das Attribut und ordnen dem Objekt diese zu. Dazu muss gesagt werden, dass `XMLHelper` einen ID-Mechanismus zur Verfügung stellt, der beliebige Objekte mit `String`-IDs assoziieren kann. Die Methode `putObjectForID()` ordnet explizit übergebene ID und Objekt einander zu, `getIDForObject()` gibt die ID eines Objekts zurück – hat dieses noch keine, so wird eine neue generiert und implizit zugeordnet – `getObjectForID()` liefert das Objekt zu einer ID. Der Zweck des ID-Mechanismus wird klar, wenn man einen Blick zum Beispiel auf die GTS DTD wirft; in dieser werden IDs massiv verwendet.

Abschließend sei noch die Klasse `ParameterManager` erwähnt, von der sich `XMLHelper` ableitet. Diese trägt dem Problem Rechnung, dass das `XMLObject`-Interface recht starr ist (wie es jedes Interface wäre) und es nicht möglich ist, den Prozess des Abspeicherns oder Ladens eines Objekts durch Parameter zu beeinflussen. Dies ist aber manchmal vonnöten, etwa wenn wie im Beispiel des Alphabets in einem Fall eine lange Version, die auch alle grafischen Informationen enthält, abgespeichert werden muss, in einem anderen aber eine Kurzversion, die abstrakte Syntax, genügt. Wird die `XwriteObject()`-Methode des `Alphabet`-Objekts aufgerufen, so kann diese zusätzliche Information nur im `XMLHelper`-Objekt gespeichert sein und eben dafür ist die Klasse `ParameterManager` verantwortlich, die im folgenden Abschnitt beschrieben wird.

5.8.3.3 `ParameterManager`

Ein Objekt der Klasse `ParameterManager` verwaltet Werte für Parameter. Ein Parameter wird identifiziert durch ein beliebiges Objekt, im Normalfall wird es sich um einen `String` handeln. Was einen `ParameterManager` von einer einfachen `Map` unterscheidet, sind zunächst einmal eine Reihe von spezialisierten Versionen für Methoden zum Setzen der Werte und den Zugriff auf sie. `getParameter` zum Beispiel gibt es auch mit einem zweiten Argument für einen Defaultwert – und dies auch für primitive Typen – falls der Parameter überhaupt nicht gesetzt ist.

Ein viel wesentlicherer Unterschied zu einer `Map` ist aber ein Stack-artiges Verhalten beim Setzen und Entfernen von Parameterwerten. `pushParameter()` setzt den Wert eines Parameters, doch falls er zuvor einen anderen Wert hatte, geht dieser nicht verloren; nach einem `popParameter()` ist alles wieder beim Alten. `removeParameter()` entfernt einen Parameter gänzlich

5.8.3.4 Alphabet

Zum Speichern eines Alphabets wird die GTS DTD verwendet, genauer deren `Types`-Element. Da es sich bei visuellen Alphabeten um Graphstruktur-Signaturen handelt, ist es kein Problem, sie als Typgraphen abzuspeichern. Symbole werden dabei zu Knoten- (`NodeType`), und Links zu Kanten- (`EdgeType`) und Datentypen zu Attributtypen (`AttrType`). Die einzige Frage, die sich stellt, ist, was mit der konkreten Syntax geschehen soll. Die GTS DTD hat dafür die Elemente `NodeTypeLayout`, `EdgeTypeLayout` und `AttrTypeLayout` vorgesehen, die die in Abschnitt 5.8.2.2 vorgestellte DTD verfeinert hat. Dort wurde auch schon erwähnt, wie verfahren wird: Die Layout-Element besitzten Unterelemente `SymbolObject`, `Operation` und `DatatypeObject` in die einfach alle Daten eines Objekts der gleichnamigen Klassen gespeichert werden. Solange noch kein akzeptables Austauschformat für Grafiken verfügbar ist, stellt dies die mit Abstand einfachste Methode dar.

Die Klassen `Alphabet`, `SymbolObject`, `Operation` und `DatatypeObject` sowie alle Klassen, die Daten für grafische Primitiva enthalten, implementieren `XMLObject`, ebenso wie die `Datatype`-Klassen. Abgesehen von `Alphabet` speichern all diese ihre Daten einfach in Attribute oder Unterelemente; Unterobjekte werden rekursiv hinzugefügt. Etwas mehr Arbeit hat nur `Alphabet`, da es sich an die vorgegebene Struktur des XML-Dokuments halten und so Elemente für Knoten- und Kantentypen anlegen und die Datentypen als Attributtypen der Knotentypen speichern muss.

5.8.3.5 Visueller Satz

Ein visueller Satz wird unter Verwendung der GTS DTD als Startgraph einer Graphgrammatik abgespeichert. Der vorrangegangene Abschnitt beschrieb, wie ein Alphabet als Typgraph in Elemente der DTD kodiert wird, analog geschieht dies auch mit einem visuellen Satz. Die Symbole werden zu Knoten (`Node`), die Links zu Kanten (`Edge`) und die Datentypen zu Attributen (`Attribute`). Auch hier werden die `SymbolObject`- und `Datatype`-Objekte vollständig in den entsprechenden Unterelementen der Layoutelemente gespeichert. Das Speichern der `Operation`-Objekte ist nicht notwendig, da sie keine Informationen enthalten, die nicht aus dem Alphabet rekonstruiert werden können.

Der Dateiname des Alphabets des visuellen Satzes wird in einem `TaggedValue`-Element gespeichert. Nichtsdestotrotz muss eine verkürzte Version im `Types`-Element niedergelegt werden, um eine korrekte Zuordnung der IDs der Typen zu ermöglichen. Die IDs sind nämlich nur innerhalb eines Dokuments konsistent (siehe dazu Abschnitt 5.8.3.2) – wird ein Alphabet mehrmals abgespeichert, müssen die Typen nicht in allen Dateien die gleichen IDs tragen. Verkürzt ist die beim visuellen Satz abgespeicherte Version des Alphabets insofern, dass keinerlei Layoutelemente enthalten sind.

Das Speichern der visuellen Sätze übernimmt nicht die Klasse `DiagramAlgebra`, sondern `GGAlgebra`. Diese sowie die Klassen `GGElement` und `GGAttribute` implementieren `XMLObject`. Die Vorgehensweise ist recht einfach: Die Algebra iteriert über die Elemente und lässt sie sich als `Node` abspeichern, die Elemente wiederum sorgen dafür, dass die Attribute geschrieben werden. Anschließend werden die `GGOperationMapping`-Objekte in `Edge`-Elementen kodiert. Das Lesen erfolgt dual. Wenn nicht gegeben, wird das Alphabet aus der Datei, deren Name im `TaggedValue`-Element zu finden ist, geladen und in eine Signatur konvertiert. Anschließend wird das `Types`-Element analysiert, das heißt, dessen Inhalt mit der Signatur verglichen und die Sorten und Operationssymbole den IDs zugeordnet. Mit der bekannten Signatur und der Zuordnung von IDs und Typen kann die Algebra aus dem gespeicherten Graphen rekonstruiert werden.

5.8.3.6 Grammatik

Da die Struktur einer visuellen Grammatik im Wesentlichen der einer Graphgrammatik entspricht und, wie in den vorangegegangenen Abschnitten erläutert, Alphabete als Typgraphen und Algebren als Graphen kodiert werden können, ist das Speichern und Laden einer visuellen Grammatik kanonisch: `GGGrammar`, `GGRule`, `GGMorphism` und `GGParameter` implementieren das `XMLObject`-Interface und sind in der Lage, ihre Daten in die entsprechenden Elemente der GTS DTD zu schreiben und daraus auch wieder zu rekonstruieren. Analog zum Vorgehen bei Diagrammen wird auch bei einer Grammatik der Name der Alphabetdatei gespeichert und eine Kurzform des Alphabets in das `Types`-Element geschrieben.

5.8.3.7 VL-Spezifikation

VL-Spezifikationen werden entsprechend der für sie entworfenen DTD (Abschnitt 5.8.2.3) gespeichert. Alle beteiligten Klassen implementieren `XMLObject` und schreiben ihre Daten in ein zumeist sogar gleichnamiges Element. Alphabete und Grammatiken werden in `GraphTransformationSystem`-Elemente entsprechend der GTS DTD gespeichert, die Klasse `AGGCriticalPairs` verwendet für ihre `ExcludePairContainer`-Instanz den in AGG implementierten Mechanismus – der `XMLHelper` stellt dafür spezielle Methoden (`addAGGSubObject()` und `readAGGSubObject()`) zur Verfügung.

5.9 Editoren

Zur Realisierung der in den Abschnitten 4.1, 4.2, 4.5 und 4.6 beschriebenen Anforderungen an die grafische Benutzerschnittstelle wurde das GUI des in GenGEd-Editor umgetauften Alphabeteditors erweitert. Grammatik- und VL-Editor aus GramEd erfuhren eine vollständige Reimplementation, ein VL-

Spezifikationseditor, eine Simulationsumgebung und eine VL-Umgebung wurden neu entworfen. Allen ist gemein, dass sie in Klassen implementiert sind, die von `AbstractEditor` ableiten und somit mit Klassen zusammenarbeiten, die das `EditorManager`-Interface zur Verfügung stellen; dies ist neben `GenGEdEditor` auch `VLEnvironment`. In letzterer sind Simulationsumgebung sowie Diagramm- und VL-Editor integriert.

Jedem der Editoren ist ein Unterabschnitt gewidmet, in dem auf seine Strukturansicht, Arbeitsfläche, Benutzeraktionen und spezifische Details eingegangen wird. Die Strukturansicht besteht immer aus einem `JTree`-Objekt (vergleiche Abschnitt 3.2.2.4); relevant für dessen Inhalt ist ein Modell (Model-View-Controller-Designpattern), eine das `TreeModel`-Interface implementierende Klasse. In allen Fällen leitet sie sich von `GenGEdDefaultTreeModel` ab. Zur Kontrolle der Selektion wird ebenfalls ein Modell benötigt, welches von einer Klasse, die das `TreeSelectionModel`-Interface bereitstellt, und bei allen Editoren `DefaultTreeSelectionModel` beerbt, repräsentiert. Das Selektionsmodell stellt jeweils eine Reihe von Hilfsmethoden zur Verfügung, die einen komfortableren Zugriff auf die Selektion bieten; es wird, da es aus dem Gesichtspunkt des Entwurfs nicht besonders interessant ist, nicht näher erläutert.

Alle Editoren verwenden konsequent den aus dem Alphabeteditor stammenden Action-Mechanismus (3.2.2.4). Alle Benutzeraktionen, die durch Menüpunkte, Buttons der Werkzeugleiste oder ähnliches initiiert werden können, sind durch eine `EditorAction`-Instanz repräsentiert. Bei ihrer Auslösung wird eine sogenannte Hook- oder Eintrittsmethode aufgerufen. Die Unterabschnitte „Benutzeraktionen“ listen die Eintrittsmethoden der Aktionen des jeweiligen Editors vollständig auf.

5.9.1 Grammatikeditor

Die Klasse `NewGrammarEditor` implementiert das GUI für den Grammatikeditor. Abbildung 5.20 zeigt seine Beziehungen zu den hier relevanten Klassen.

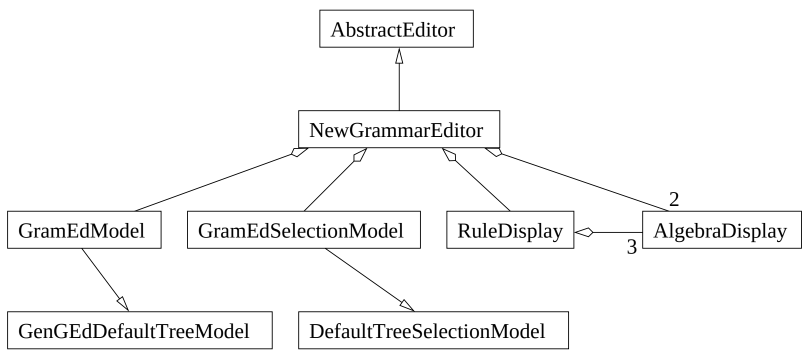

Abbildung 5.20: Die Klassen des Grammatikeditors.

5.9.1.1 Strukturansicht

Das Datenmodell für die Strukturansicht wird von `GramEdModel` implementiert. Es stellt eine Grammatik sowie beliebig viele Diagramme als Baum dar. Die Regeln der Grammatik werden unter einem Punkt „Regeln", das Startdiagramm und alle weiteren Diagramme unter „Diagrammen" aufgelistet. Hat eine Regel NACs, so besitzt sie entsprechende Unterpunkte. Dies ist der Standardmodus des Modells, es gibt jedoch die Möglichkeit, die Ansicht noch zu verfeinern. Die Methode `setShowSymbols()` dient dazu, zwischen dem Standardmodus und einer Ansicht, bei der jede Regel Unterpunkte für die Diagramme in der linken und rechten Regelseite besitzt und jedes Diagramm all seine Symbole auflistet, umzuschalten.

`setGrammar()`, `addDiagram()`, `removeDiagram()` und `removeAllDiagrams()` dienen dazu festzulegen, welche Grammatik und welche Diagramme das Modell darstellen soll. Alle weiteren Methoden sind entweder privat und helfen, auf die Nodes zuzugreifen oder neue zu kreieren, oder es handelt sich um Listener-Methoden, die aufgerufen werden, wenn für die Darstellung relevante Änderungen an den Daten vorgenommen wurden.

5.9.1.2 Arbeitsfläche

Die Arbeitsfläche des Grammatikeditors besteht aus einem `RuleDisplay` zur Anzeige von Regeln mit NAC und zwei `AlgebraDisplay`-Objekten für zu editierende Diagramme, Regeln und NACs. Diese Klassen sind adaptierte Versionen von `RuleDisplay` und `DiagramDisplay` aus GRAMED.

Ein `AlgebraDisplay` zeigt eine `GGAlgebra` an. Es verwendet dazu eine `DiagramDrawArea` (aus GRAMED, jedoch modifiziert), die ein Objekt der Klasse `DiagramAlgebra` visualisieren kann. `RuleDisplay` besteht aus drei `AlgebraDisplay`-Objekten für LHS und RHS der Regel und ein NAC-Diagramm.

Zur Visualisierung von Morphismen werden Zahlen rechts über den Symbolen angezeigt. Tragen in verschiedenen Displays Symbole die gleiche Zahl, so existiert ein Mapping zwischen diesen. Die Displays haben damit nicht viel zu tun, denn die Symbole selbst, genauer die Klasse `GraphicObject`, zeichnen zusätzlich an eine der vier Ecken ihrer Bounding-Box einen beliebigen Text, den man mittels `setText()` setzen kann. Festgelegt werden die Zahlen durch ein Objekt der Klasse `GGMorphismSet`, die alle auf der Arbeitsfläche dargestellten Morphismen kennt und somit vermeiden kann, dass Kollisionen auftreten.

`GGMorphismSet` repräsentiert keine, wie der Name suggerieren mag, strukturlose Menge von Morphismen, sondern vielmehr Morphismen eines Morphismenschemas. Bevor überhaupt ein Morphismus in das `GGMorphsimSet`-Objekt gelangen kann, muss erst ein solches Schema definiert werden. Dies geschieht mit Hilfe der Methoden `addAlgebra()` und `addMorphism()`. Erstere bekommt eine Zahl, die einen Platzhalter für eine Algebra darstellt, und – jedoch nur weil sich Probleme beim automatischen Aktualisieren der Zeichenfläche aufgetan haben – ein `AlgebraDisplay`, in dem die entsprechende Algebra visualisiert wird. `addMorphism()` hat als Parameter eine Zahl, die Platzhalter für einen Morphismus ist, und zwei Algebranummern für Domain und Codomain des Morphismus. Das so definierte Schema kann als Graph aufgefasst werden, der Algebranummern als Knoten und Morphismennummern als Kanten besitzt.

Erst nachdem das Schema definiert ist, kommen tatsächliche Algebren und Morphismen ins Spiel: `setAlgebra()` ordnet einer Algebranummer eine Algebra und `setMorphism()` einer Morphismennummer einen Morphismus zu. Übergibt man *null*, so wird die Algebra respektive der Morphismus wieder entfernt. `unsetAllMorphisms()` entfernt alle Morphismen, `unsetAll()` zusätzlich auch noch die Algebren. Immer wenn ein Morphismus gesetzt oder entfernt wurde oder sich seine Mappings verändert haben, berechnet `GGMorphismSet` eine neue Zuordnung der Zahlen zu den Symbolen.

5.9.1.3 Benutzeraktionen

Die Tabellen 5.1 und 5.2 enthalten alle Eintrittsfunktionen der Benutzeraktionen samt einer kurzen Beschreibung ihrer Funktionalität.

Eintrittsfunktion	Funktionalität
`NewFile()`	erstellt eine neue Grammatik
`OpenFile()`	öffnet eine Grammatikdatei
`SaveFile()`	speichert die Grammatik in eine Datei
`Undo()`	macht den letzten Schritt rückgängig
`Redo()`	wiederholt den zuletzt rückgängig gemachten Schritt
`WorkOnRule()`	beginnt die Arbeit an einer Regel
`Duplicate()`	dupliziert die gewählte Regel oder NAC
`Remove()`	entfernt die gewählte Regel oder NAC
`Export()`	exportiert Regeln der Grammatik
`AdjustToAlphabet()`	passt die Grammatik an ein geändertes Alphabet an
`NewRule()`	fügt zur Grammatik eine neue leere Regel hinzu
`NewNAC()`	fügt zur selektierten Regel eine neue leere NAC hinzu
`DisplaySymbols()`	schaltet die Strukturansicht zwischen Standardmodus und erweiterter Anzeige um

Tabelle 5.1: Eintrittsmethoden der Klasse `NewGrammarEditor`, Teil 1.

Eintrittsfunktion	Funktionalität
`AddMapping()`	fügt zum Match ein Mapping zwischen den selektierten Symbolen hinzu
`RemoveMapping()`	entfernt vom Match ein Mapping eines selektierten Symbols
`RemoveAllMappings()`	entfernt alle Mappings des Matches
`Complete()`	vervollständigt den Match
`ApplyRule()`	wendet die gewählte Regel an
`AddRuleMapping()`	fügt zum Regel- oder NAC-Morphismus ein Mapping zwischen den selektierten Symbolen hinzu
`RemoveRuleMapping()`	entfernt vom Regel- oder NAC-Morphismus ein Mapping eines selektierten Symbols
`RemoveAllRuleMappings()`	entfernt alle Mappings des Regel- oder NAC-Morphismus
`CompleteRule()`	vervollständigt den Regel- oder NAC-Morphismus
`IdenticCopy()`	macht den Regel- oder NAC-Morphismus zum identischen Morphismis auf dem Domain
`SetRuleParameters()`	öffnet einen Dialog zum Editieren der Regelparameter
`EditRuleAttributeConditions()`	öffnet einen Dialog zum Editieren der Attributbedingungen
`DefineRule()`	beendet die Arbeit an einer Regel

Tabelle 5.2: Eintrittsmethoden der Klasse `NewGrammarEditor`, Teil 2.

5.9.1.4 Transformation

Im Grammatikeditor werden Startdiagramm und Regeln der Grammatik durch Regelanwendungen editiert. Die dazu notwendigen Match- und Morphismenvervollständigungen und Transformationen erledigt ein `GGAlgebraTransformer`. Dieser wird während der Initialisierung des Grammatikeditors von der `GGFactory`-Methode `createAlgebraTransformer()` kreiiert und von da an in der Variable `transformer` gehalten. Immer wenn eine neue Grammatik gesetzt wird, wird seine `init()`-Methode mit der Signatur zur Grammatik aufgerufen, um ihn auf die folgenden Aufgaben vorzubereiten.

Die Aktualisierung nach einer Transformation ist unproblematisch: Die Displays implementieren einen Listener-Mechanismus und werden sofort auf den neuesten Stand gebracht, und bei Veränderungen an Morphismen wird das zuständige

`GGMorphismSet`-Objekt über den `GGObservable`-Mechanismus benachrichtigt und kann entsprechend reagieren.

5.9.2 VL-Editor

Die Klasse `NewVLEditor` implementiert das GUI für den VL-Editor. Abbildung 5.21 zeigt seine Beziehungen zu den hier relevanten Klassen. Die Ähnlichkeit zu `NewGrammarEditor` sollte wenig überraschen, denn von der Funktionalität ist der VL-Editor nur ein eingeschränkter, um eine Verwaltung der Diagramme erweiterter Grammatikeditor.

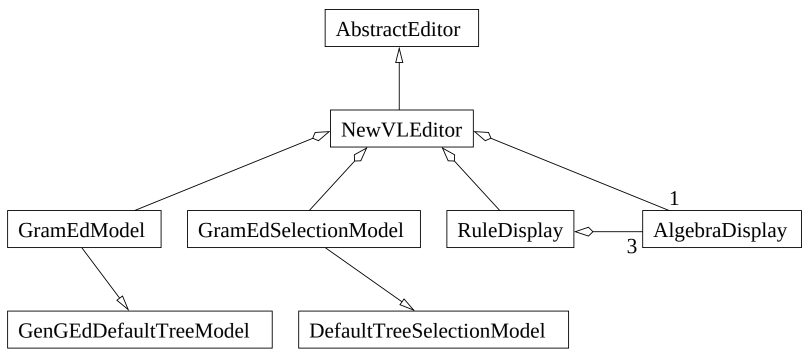

Abbildung 5.21: Die Klassen des VL-Editors.

5.9.2.1 Strukturansicht

Es werden die selben Modellklassen, `GramEdModel` und `GramEdSelectionModel`, wie beim Grammatikeditor verwendet (Abschnitt 5.9.1.1). Genauer: Dass neben der Grammatik optional weitere Diagramme zum Modell gehören ist eine Erweiterung der ursprünglichen, für den Grammatikeditor benötigten Funktionalität, um die Klassen für den VL-Editor wiederverwenden zu können.

5.9.2.2 Arbeitsfläche

Die Arbeitsfläche des VL-Editors ähnelt der des Grammatikeditors (Abschnitt 5.9.1.2), besitzt jedoch nur ein `AlgebraDisplay`, da nur einfache Diagramme, keine Regeln oder NACs, editiert werden können.

5.9.2.3 Benutzeraktionen

Tabelle 5.3 enthält alle Eintrittsfunktionen der Benutzeraktionen samt einer kurzen Beschreibung ihrer Funktionalität.

Eintrittsfunktion	Funktionalität
`OpenGrammar()`	öffnet eine Grammatikdatei
`OpenDiagram()`	öffnet eine Diagrammdatei
`SaveDiagram()`	speichert das gewählte Diagramm in eine Datei
`Undo()`	macht den letzten Schritt rückgängig
`Redo()`	wiederholt den zuletzt rückgängig gemachten Schritt
`Remove()`	entfernt die selektierten Diagramme
`NewDiagram()`	fügt eine Kopie des Startdiagramm hinzu
`AddMapping()`	fügt zum Match ein Mapping zwischen den selektierten Symbolen hinzu
`RemoveMapping()`	entfernt vom Match ein Mapping eines selektierten Symbols
`RemoveAllMappings()`	entfernt alle Mappings des Matches
`Complete()`	vervollständigt den Match
`ApplyRule()`	wendet die gewählte Regel an

Tabelle 5.3: Eintrittsmethoden der Klasse `NewVLEditor`.

5.9.3 VL-Spezifikationseditor

Das GUI für den VL-Spezifikationseditor ist in der Klasse `VLSpecificationEditor` implementiert. Ihre Beziehungen zu anderen Klassen zeigt Abbildung 5.22.

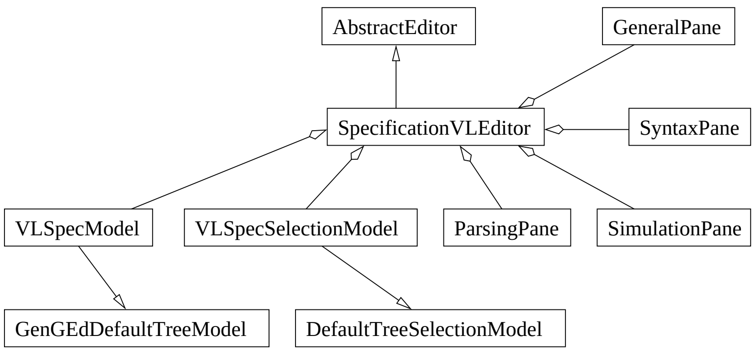

Abbildung 5.22: Die Klassen des VL-Spezifikationseditors.

5.9.3.1 Strukturansicht

Die Strukturansicht des VL-Spezifikationseditors zeigt nur im weiteren Sinne die Struktur der editierten VL-Spezifikation. Es werden immer die vier Punkte „Allgemeines", „Syntax", „Parsierung" und „Simulation" präsentiert, von denen nur der letzte Unterpunkte, nämlich für die Simulationsspezifikationen, besitzen kann.

Die für Daten- und Selektionsmodell zuständigen Klassen `VLSpecModel` und `VLSpecSelectionModel` sind dementsprechend einfach. `VLSpecModel` besitzt eine Methode `setSpecification()` zum Setzen der zu visualisierenden VL-Spezifikation, alle anderen Methoden sind Hilfsmethoden.

5.9.3.2 Arbeitsfläche

Je nachdem welcher der vier Aspekte in der Strukturansicht gewählt wurde, wird eine andere Komponente auf der Arbeitsfläche angezeigt. Dabei handelt es sich jeweils um ein Objekt der Klassen `GeneralPane`, `SyntaxPane`, `ParsingPane` und `SimulationPane`.

`GeneralPane` präsentiert ein GUI zum Editieren der `VLSpecOptions` der VL-Spezifikation und `SyntaxPane` ermöglicht das Festlegen von Alphabet und Syntaxgrammatik. `ParsingPane` besitzt GUI-Elemente, mit denen sich die in Parsierungsspezifikation verändern lässt. Es kann die Parsierunggrammatik angegeben werden, die Layerfunktion und man kann die Analyse der kritischen Paare durchführen lassen. Am komplexesten ist `SimulationPane`, das heißt nicht die Klasse selbst, denn mit ihr lassen sich nur die Simulationsgrammatik einer Simulationsspezifikation festlegen und Simulationsschritte hinzufügen, entfernen und deren Namen ändern, sondern der Dialog zum Editieren eines Simulationsschritts. Dieser wird im Abschnitt 5.9.3.4 beschrieben.

5.9.3.3 Benutzeraktionen

Tabelle 5.4 enthält alle Eintrittsfunktionen der Benutzeraktionen samt einer kurzen Beschreibung ihrer Funktionalität.

5.9.3.4 Simulationsschrittdialog

Die Klasse des Simulationsschrittdialogs, `SimStepDialog` und alle zugehörigen Klassen residieren im eigenen Package `genged.gui.simulation`. Es soll hier nur dessen grobe Struktur (Abb. 5.23) erläutert werden.

Der Dialog präsentiert eine Strukturansicht des editierten Simulationsschritts, eine Arbeitsfläche, in der in ersterer ausgewählte Objekte editiert werden können sowie eine Spalte mit Buttons zum Einfügen, Verschieben und Löschen der Objekte. Für die Strukturansicht wird ein `JTree` verwendet, dessen Modell die Klasse `SimStepModel` implementiert. Es wird eine Liste der Parameter des Simulationsschritts und der Baum seines Simulationsausdrucks angezeigt. Ist ein Parameter gewählt, so präsentiert die Arbeitsfläche eine `ParameterPane` zum Editieren des

Eintrittsfunktion	Funktionalität
`NewFile()`	erstellt eine neue leere VL-Spezifikation
`OpenFile()`	öffnet eine VL-Spezifikationsdatei
`SaveFile()`	speichert die VL-Spezifikation in eine Datei
`Undo()`	macht den letzten Schritt rückgängig
`Redo()`	wiederholt den zuletzt rückgängig gemachten Schritt
`NewSimulationSpec()`	fügt zur VL-Spezifikation eine neue leere Simulationsspezifikation hinzu
`Remove()`	entfernt die selektierten Simulationsspezifikationen

Tabelle 5.4: Eintrittsmethoden der Klasse `VLSpecificationEditor`.

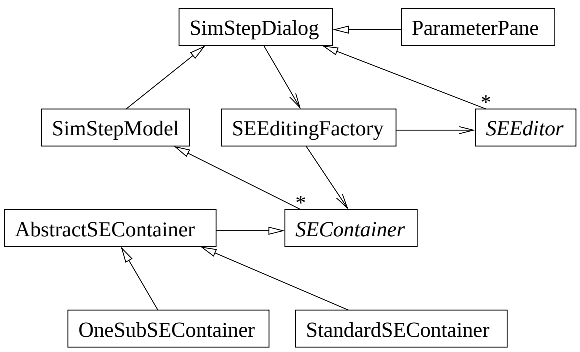

Abbildung 5.23: Die Klassen des Simulationsschrittdialogs.

Parameters. Alle Teile, die mit Simulationsausdrücken zu tun haben, sind generisch implementiert oder präziser, verwenden eine Schicht zur Abstraktion.

Die Strukturansicht visualisiert aus diesem Grund alle Simulationsausdrücke auf die gleiche Weise, durch eine Node, die den Namen des Ausdrucks trägt, sowie Unternodes, die als Container für die Unterausdrücke fungieren. Ein solcher grafischer Container hat ein nicht grafisches Gegenstück im Interface `SEContainer`. Dieses stellt die Methode `getName()`, die den Namen des Containers zurückgibt, sowie alle zur Verwaltung von Unterausdrücken notwendigen Methoden zur Verfügung, sogar ein `getMaximalSubExpressionCount()`, das die Kapazität des Containers liefert. Die Anwendung ist wie folgt gedacht: Gegeben einen Simulationsausdruck kreiert die `SEEditingFactory`-Methode `createContainersFor()` einen Vector von `SEContainer`-Objekten. Im Beispiel eines If-Ausdrucks würden drei Container mit Kapazität eins für die Bedingung, den Then- und den Else-Ausdruck kreiert, für einen And-Ausdruck wäre es nur ein Container mit unbeschränkter Kapzität. Auf diese Weise kann `SimStepModel` die Struktur eines Ausdrucks visualisieren, ohne dessen konkrete Klasse zu kennen.

Doch nicht nur die Visualisierung der Struktur, auch deren Editieren ist problemlos möglich: Ist ein Container gewählt (oder ein Ausdruck in einem Container), dessen Kapazität noch nicht erschöpft ist, so kann ein Ausdruck eingefügt werden, das Löschen funktioniert sowieso ohne Schwierigkeiten. Nun bleibt nur noch zu klären, wie etwaige Parameter eines Ausdrucks editiert werden.

Zu diesem Zweck stellt `SEEditingFactory` die Methode `createEditors()` bereit, die ein Array von `SEEditor`-Objekten zurückgibt, und zwar für jeden Ausdruckstyp eines. Das `SEEditor`-Interface stellt Methoden zur Verfügung, die den Namen des Ausdruckstyp und seine Klasse zurückgeben ebenso wie eine neu kreierte Instanz, doch am Wichtigsten eine GUI-Komponente (`getEditingComponent()`), die es ermöglicht, einen Ausdruck dieses Typs zu editieren.

Die wesentlichen Schritte, um einen neuen Ausdruckstypen im GUI verfügbar zu machen, bestehen also darin, Containerklassen für die Unterausdrücke und eine `SEEditor` implementierende Klasse zu schreiben und diese in die `SEEditingFactory` aufzunehmen. Bei den Containerklassen ist es nicht nötig, von null anzufangen. Es kann zumindest auf `AbstractSEContainer` zurückgegriffen werden, in der der kanonische Teil, das heißt Attribute und Zugriffsmethoden für Namen und Kapazität und Methoden, die sich auf andere zurückführen lassen, implementiert ist. Noch spezieller sind `OneSubSEContainer` für Container mit Kapazität eins und `StandardSEContainer` für solche, die direkt auf die Unterausdrucksverwaltung von `SimulationExpression` aufsetzen.

5.9.4 Simulationsumgebung

Die Simulationsumgebung ist keine unabhängig verwendbare Komponente, wie alle anderen (Unter-)Editoren. Sie wurde explizit als Teilsystem der VL-Umgebung konzipiert. Ihr GUI ist in `SimulationEnvironment` implementiert, die Beziehungen zu anderen Klassen visualisiert Abbildung 5.24.

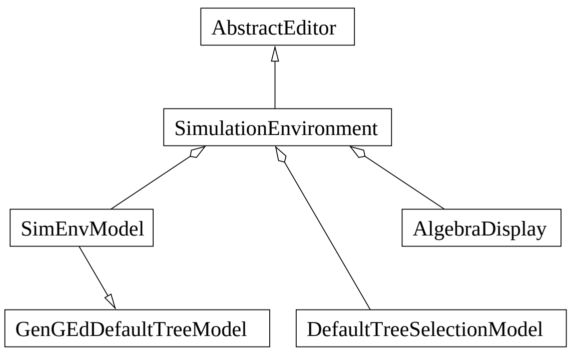

Abbildung 5.24: Die Klassen der Simulationsumgebung.

5.9.4.1 Strukturansicht

Als Datenmodell für die Strukturansicht ist die Klasse `SimEnvModel` vorgesehen. Sie kreiert Nodes für die Simulationsspezifikationen einer VL-Spezifikation und Unternodes für deren Simulationsschritte. Da die VL-Spezifikation in der Simulationsumgebung nicht verändert wird, muss `SimEnvModel` nicht einmal auf Änderungen reagieren können. Außer `setSpecification()` zum Setzen der VL-Spezifikation ist keine weitere Methode erwähnenswert.

Ob der geringen Anforderungen an das Selektionsmodell für die Simulationsumgebung wird direkt `DefaultTreeSelectionModel` verwendet, ohne davon abzuleiten.

5.9.4.2 Arbeitsfläche

Die Arbeitsfläche besteht nur aus einem `AlgebraDisplay`, in dem das gerade simulierte Diagramm angezeigt wird.

5.9.4.3 Benutzeraktionen

Tabelle 5.5 enthält alle Eintrittsfunktionen der Benutzeraktionen samt einer kurzen Beschreibung ihrer Funktionalität.

Eintrittsfunktion	Funktionalität
`Undo()`	macht den letzten Schritt rückgängig
`Redo()`	wiederholt den zuletzt rückgängig gemachten Schritt
`Step()`	wendet den selektierten Simulationsschritt an
`ResetSimulation()`	setzt die Simulation auf den Ausgangszustand zurück
`StopSimulation()`	stoppt die Simulation

Tabelle 5.5: Eintrittsmethoden der Klasse `SimulationEnvironment`.

5.9.4.4 Simulation

Die Simulationsumgebung implementiert eine Schrittsimulation, das heißt, ein ausgewählter Simulationschritt kann auf das aktuelle Diagramm angewendet werden. Dies erfolgt durch Auswertung des Schritts mittels eines `SEEvaluator`-Objekts. Ein `GGAlgebraTransformer`, der bei der Auswertung benötigt wird, wird bei der Initialisierung der Umgebung kreiert und bei jeder Änderung der VL-Spezifikation durch Aufruf seiner `init()`-Methode auf die entsprechende Signatur eingestellt.

Da die Auswertung eines Simulationsschritts längere Zeit dauern, ja sogar nicht terminieren kann, muss sie asynchron durchgeführt werden, damit dem Benutzer die Möglichkeit des Abbruchs der Simulation eingeräumt werden kann. Zu diesem Zweck wurde die Klasse `StepEvaluator` entworfen, die das `Runnable`-Interface implementiert und nach dem Aufruf der `start()`-Methode einen separaten Thread kreiiert und diesem die Auswertung überlässt. Wenn die Ausführung eines Simulationsschritts ansteht, kreiert die Simulationsumgebung ein `StepEvaluator`-Objekt und initiert den Vorgang. Eine Referenz auf dieses Objekt wird in der Variable `evaluator` gespeichert. Solange diese nicht auf *null* zurückgesetzt wurde, ist die Auswertung noch im Gange. Bei deren fehlerfreier Termination, Abbruch oder im Fehlerfall wird eine der `SimulationEnvironment`-Methoden `simulationFinished`, `simulationStopped` oder `simulationFailed` aufgerufen. Diese sind `synchronized`, ebenso wie `stopSimulation()`, die den `StepEvaluator` auffordert, die Auswertung abzubrechen, sodass „Race Conditions“ vermieden werden.

5.9.5 VL-Umgebung

Die GUI-Klasse der VL-Umgebung, `VLEnvironment` implementiert das `EditorManager`-Interface, um drei Subeditoren verwalten zu können (Abb. 5.25). Dies sind `DiagramEditor` und `VLEditor` zum Editieren von VL-Diagrammen und `SimulationEnvironment` für die Simulation.

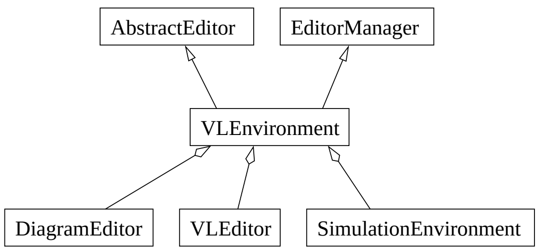

Abbildung 5.25: Die Klassen der VL-Umgebung.

5.9.5.1 Strukturansicht

Die VL-Umgebung besitzt keine eigene Strukturansicht. Es wird die Strukturansicht des aktiven Untereditors präsentiert. Dies geschieht dadurch, dass die Methode `getTree()` von `AbstractEditor` `getTree()` des aktiven Subeditors aufruft und dessen Rückgabewert zurückliefert. Ist kein Editor aktiv, was nur dann der Fall ist, wenn keine VL-Spezifikation geladen ist, wird ein `JTree` mit einem leeren Datenmodell bereitgestellt.

5.9.5.2 Arbeitsfläche

Die Arbeitsfläche der VL-Umgebung ist ein einfaches `JPanel` mit einem `CardLayout`, das die Arbeitsflächen der Untereditoren verwaltet. Je nachdem welcher von diesen aktiv ist, wird dessen Arbeitsfläche präsentiert. Im Falle, dass kein Editor aktiv ist, weil keine VL-Spezifikation geladen wurde, wird eine leere Fläche angezeigt.

5.9.5.3 Benutzeraktionen

Für die Untereditoren der VL-Umgebung sind eigene Benutzeraktionen definiert. Da die VL-Umgebung eine eigene Menü- und Werkzeugleiste benötigt und nicht jene des aktiven Untereditors verwenden kann – einige Punkte für das Laden einer VL-Spezifikation und die Diagrammverwaltung müssen hinzukommen, während etwa das Laden einer Grammatik im VL-Editor nicht möglich sein darf – können die Aktionen der Untereditoren nicht direkt verwendet werden. Dies liegt daran, dass die `EditorAction`-Objekte sowie Menü- und Werkzeugleiste im `AbstractEditor` aus mehreren festgelegten Resourcedateien in einem Verzeichnis, dessen Name dem Konstruktor übergeben wird, kreiert werden. Andere Dateien können nicht referenziert werden und andere Objekte als Ziel für die Eintrittsmethoden sind ebenfalls nicht möglich.

Somit mussten nicht nur die Aktionen, die Menü- und die Werkzeugleiste für die VL-Umgebung in den Resourcedateien vollständig definiert werden, sondern auch alle Aktionen der Untereditoren. Um nicht für jede Subeditoraktion auch eine Eintrittsmethode in der VL-Umgebung zu implementieren, wurde deren Name mit einem Präfix versehen, der die Zugehörigkeit zum Untereditor anzeigt und es konnte sich auf drei Eintrittsmethoden, `FreeAction()`, `SyntaxAction()` und `SimulationAction()`, beschränkt werden, in denen die entsprechende Aktion im Untereditor ausgelöst wird. Es gibt also für jede Benutzeraktion eines Subeditors zwei `EditorAction`-Objekt, ein „Stellvertreterobjekt", das eine der drei genannten Methoden aufruft und das vom Untereditor definierte Objekt, das die dortige Eintrittsmethode ausführt.

Da die Untereditoren nur dafür sorgen, dass die Verfügbarkeit (die Auslösbarkeit durch den Benutzer) ihrer `EditorAction` immer korrekt aktualisiert wird, jedoch das Stellvertreterobjekt davon nichts mitbekommt, musste ein zusätzlicher Listener-Mechanismus implementiert werden: Die Aktionen eines jeden Editors werden von einem Objekt der Klasse `ActionManager` verwaltet, welche auch Methoden bereitstellt, um sie zu modifizieren (`setEnabled()`, `setSelected()`). Diese Methoden informieren nun sofort alle `ActionManagerListener`, die sich zuvor beim `ActionManager` registriert haben. `VLEnvironment` implementiert dieses Interface und registriert sich zum Zeitpunkt der Initialisierung bei den `ActionManager`-Objekten der Subeditoren. Somit wird es bei jeder Änderung informiert und kann diese an die entsprechende Stellvertreteraktion weitergeben.

Tabelle 5.6 enthält alle Eintrittsfunktionen der Benutzeraktionen samt einer kurzen Beschreibung ihrer Funktionalität.

Eintrittsfunktion	Funktionalität
`OpenSpecification()`	öffnet eine VL-Spezifikationsdatei
`OpenDiagram()`	öffnet eine Diagrammdatei
`SaveDiagram()`	speichert das aktuelle Diagramm in eine Datei
`Undo()`	macht den letzten Schritt rückgängig
`Redo()`	wiederholt den zuletzt rückgängig gemachten Schritt
`NewDiagram()`	fügt ein neues Diagramm hinzu
`DuplicateDiagram()`	dupliziert das aktuelle Diagramm
`EditDiagramName()`	öffnet einen Dialog zum Editieren des Namens des aktuellen Diagramms
`RemoveDiagram()`	entfernt das aktuelle Diagramm
`ParseDiagram()`	parsiert das aktuelle Diagramm
`EditingMode()`	schaltet in den Editiermodus
`SimulationMode()`	schaltet in den Simulationsmodus
`FreeAction()`	führt eine Aktion des Diagrammeditors aus
`SyntaxAction()`	führt eine Aktion des VL-Editors aus
`SimulationAction()`	führt eine Aktion der Simulationsumgebung aus

Tabelle 5.6: Eintrittsmethoden der Klasse `VLEnvironment`.

5.9.5.4 Parsierung

Die Parsierung ist neben dem Laden einer VL-Spezifikation und der Verwaltung der VL-Diagramme die einzige im GUI sichtbare Funktionalität der VL-Umgebung. Enthält die geladene VL-Spezifikation eine brauchbare Parsierungsspezifikation, so wird der bei der Initialisierung der Umgebung von `GGFactory.createParser` kreierte Parser damit konfiguriert. Die Parsierung ist synchron implementiert – das aktuelle Diagramm wird also einfach an die `parse()`-Methode des Parsers übergeben.

Kapitel 6

Spezifikation hierarchischer Statecharts

Statecharts, als visueller Formalismus 1987 im Journal „Science of Computer Programming“ vorgestellt ([Har87]), erweitern das einfache Konzept der Zustandsautomaten um Mittel zur Hierarchisierung, Parallelisierung und Kommunikation. Ihre einfache und übersichtliche grafische Repräsentation, die Unterstützung durch verschiedene Applikationen und nicht zuletzt die Aufnahme der objektorientierten Statecharts in die UML ([UML01]) machten sie zu einer der beliebtesten Modellierungssprachen zur Beschreibung reaktiver Systeme. Zahlreiche Erweiterungen von verschiedenen Parteien führten dazu, dass es mittlerweile eine große Zahl verschiedener Statechart-Varianten gibt.

Wir beziehen uns auf eine reduzierte Variante, die zur Untersuchung der kompositionalen Semantik formal in [Joh99] eingeführt wurde, beschränken uns jedoch auf eine informelle Einführung. Mit Abschnitt 6.2 beginnt die Spezifikation einer VL-Umgebung für diese visuelle Sprache. Es werden das visuelle Alphabet und eine Syntaxgrammatik vorgestellt. Da sich mit letzterer jedoch auch Statecharts definieren lassen, die wir als inkorrekt ansehen wollen, wird im darauf folgenden Abschnitt eine Parsierungsspezifikation angegeben, welche diese erkennt. Schließlich definiert der Abschnitt 6.4 eine Simulationsspezifikation. Sie wird zwar nur mit einem Teil der korrekten Statecharts umgehen können, doch wollen wir uns, um den Rahmen dieser Arbeit nicht zu sprengen, damit begnügen.

6.1 Einführung

Ein Statechart besteht aus einer Menge von Zuständen, die baumartig strukturiert sind, das heißt es gibt einen Wurzelzustand und jeder andere Zustand besitzt genau einen übergeordneten Zustand. Wir unterscheiden zwei Zustandsarten, Hierarchie- und Parallelzustände. Zwischen beliebigen Zuständen darf eine Transition, also ein

Zustandsübergang, bestehen, die mir einem Event beschriftet ist.

Am in Abbildung 6.1 dargestellten Beispiel einer Funkuhr ist dies leicht nachzuvollziehen.

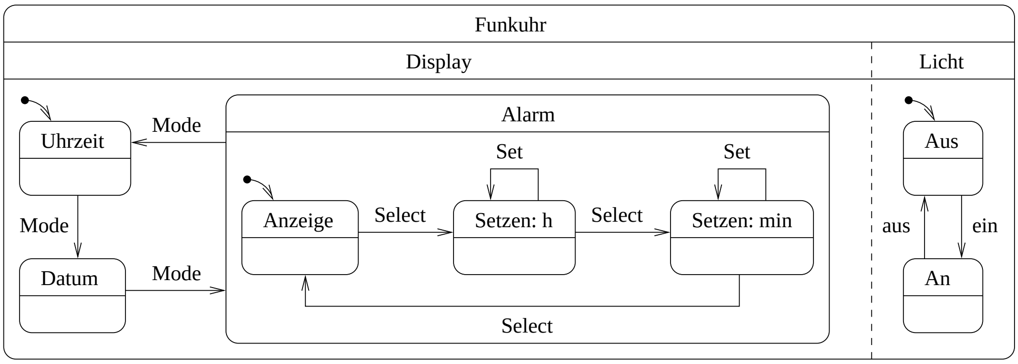

Abbildung 6.1: Beispiel eines Statecharts.

Der Wurzelzustand ist „Funkuhr". Dieser ist ein Parallelzustand mit zwei Unterzuständen, „Display" und „Licht". Alle anderen Zustände sind Hierarchiezustände. Das Licht kann also unabhängig vom Display der Uhr ein und ausgeschaltet werden. Auf dem Display kann zwischen den Zuständen „Uhrzeit", „Datum" und „Alarm" mit Hilfe des „Mode"-Knopfs umgeschaltet werden. Im Zustand „Alarm" ist es mit dem „Select"-Knopf möglich, von der Anzeige zum Setzen der Stunden und Minuten und wieder zurück umzuschalten. Ein kleiner Pfeil kennzeichnet „Uhrzeit", „Anzeige" und „Aus" als initiale Zustände. In jedem Hierarchiezustand muss genau ein Unterzustand als initial markiert sein. Die Semantik ist die, dass, wenn in einen Hierarchiezustand umgeschaltet wird, auch sein initialer Unterzustand aktiviert wird. Ist ein Parallelzustand aktiv, sind es auch all seine Unterzustände.

Die Bedeutung der Transitionen ist intuitiv: Ist der Quellzustand einer Transition aktiv, wenn der Event, mit dem sie beschriftet ist, auftritt, so wird der Quellzustand deaktiviert und der Zielzustand aktiviert. Was genau für Fälle auftreten können, wenn mehrere Transitionen zugleich schalten könnten, soll hier nicht weiter untersucht werden. [Joh99] betrachtet dazu Mengen von Transitionsscharen, aus denen letztlich auch wieder eine Schar gewählt werden muss, wenn mehrere verbleiben. Relevant ist diese Untersuchung nur für die Simulation und für diese werden wir naiv versuchen, einfach alle möglichen Transitionen schalten zu lassen. Das mag zwar für manche Statecharts nicht funktionieren, doch nehmen wir das, in Anbetracht dessen, dass die Komplexität sonst sehr viel größer wäre, in Kauf.

Eine zusätzliche Bedingung soll noch an korrekte Statecharts gestellt werden: Quell- und Zielzustand einer Transition dürfen nicht (mittelbar oder unmittelbar) in verschiedenen Unterzuständen eines Parallelzustands liegen. In unserem Beispiel sind also Transitionen von „Display“ nach „Licht“, von „Uhrzeit“ nach „Aus“, von „An“ nach „Anzeige“ oder ähnliche nicht erlaubt.

6.2 Alphabet und Syntaxgrammatik

Da das visuelle Alphabet in GenGEd nicht zyklisch sein darf, können wir die Baumstruktur der Zustände nicht direkt, also mit einem Symbol für einen Zustand und einem Link für die Baumrelation, modellieren, sondern benötigen ein oder mehrere Hilfsymbole. Tatsächlich werden dafür vier Symbole, `S`, `SC`, `H`, `P`, verwendet (die vollständigen Symbolnamen sind in Tabelle 6.1 zu finden). Ein Zustand setzt sich aus den zwei Symbolen `S` und `SC` zusammen. Das erste repräsentiert den Zustand in seiner Eigenschaft als Zustand selbst, das heißt als Start- und Endpunkt für Transitionen, kennzeichenbar als Wurzel- oder initialer Zustand, `SC` hingegen dient als Container für Unterzustände. `H` und `P` legen fest, ob ein Unterzustand zu einem Hierarchie- oder Parallelzustand gehört und ordnet ihm das entsprechende `SC`-Symbol zu. Abbildung 6.2 zeigt die abstrakte Syntax des vollständigen Alphabets.

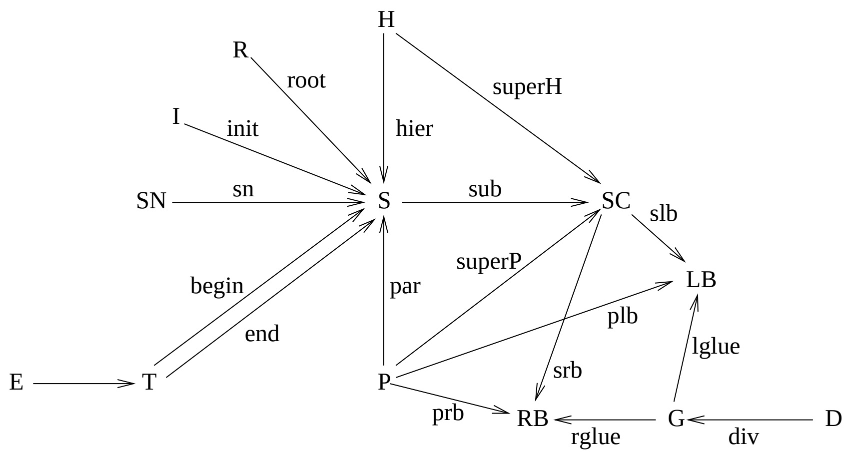

Abbildung 6.2: Abstrakte Syntax des Statechart-Alphabets.

Symbol-kürzel	Name	Symbol-kürzel	Name
`S`	State, Zustand		
`SC`	Statechart	`H`	Hierarchie
`P`	parallel	`R`	Root, Wurzel
`I`	Initial	`SN`	State Name, Zustandsname
`T`	Transition	`E`	Event
`LB`	Left Border, linker Rand	`RB`	Right Border, rechter Rand
`G`	Glue, Kleber	`D`	Divider, Trennlinie

Tabelle 6.1: Namen der Statechart-Symbole.

Die Symbole `R` und `I` kennzeichnen den Wurzelzustand bzw. initiale Zustände, `SN` ist ein Datentyp für den Zustandsnamen, `E` für einen Event. Eine Transition wird durch das Symbol `T` repräsentiert. `LB`, `RB`, `G` und `D` sind Hilfsymbole für das Layout der Parallelzustände. `LB` und `RB` markieren den linken und rechten Rand eines `P`- oder den rechten und linken Rand (aus Implementierungsgründen umgekehrt) eines `SC`-Symbols. `G` wird dazu benutzt, um den rechten Rand eines mit dem linken Rand eines anderen Objekts zu verkleben und `D` ist eine sichtbare Trennlinie für eine solche Verklebungsstelle. Die Grafiken aller Symbole zeigt Tabelle 6.2. Die Datentypen `SN` und `E` wurden ausgelassen, es handelt sich um einfache Strings. Gepunktete Linien oder Rechtecke stellen unsichtbare Grafiken dar.

Symbol	Grafik	Symbol	Grafik	Symbol	Grafik
`S`		`SC`		`H`	
`P`		`R`		`I`	
`T`		`LB`		`RB`	
`G`		`D`			

Tabelle 6.2: Grafiken der Statechart-Symbole.

Wie Symbole und Links zusammenwirken, zeigen die Abbildungen 6.3 bis 6.6. Es sind jeweils abstrakte und konkrete Syntax dargestellt, jedoch mussten, obwohl zu beiden Statecharts je zwei Abbildungen gegeben sind, in denen unterschiedliche Beziehungen visualisiert wurden, aus Platzgründen einige Details ausgelassen werden.

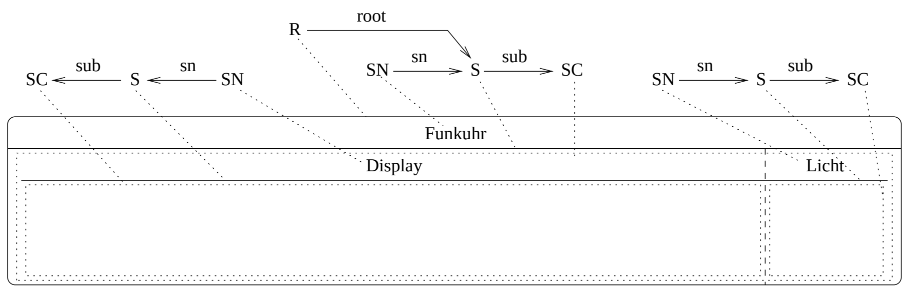

Abbildung 6.3: Links des Statechart-Alphabets, Parallelzustand, Teil 1.

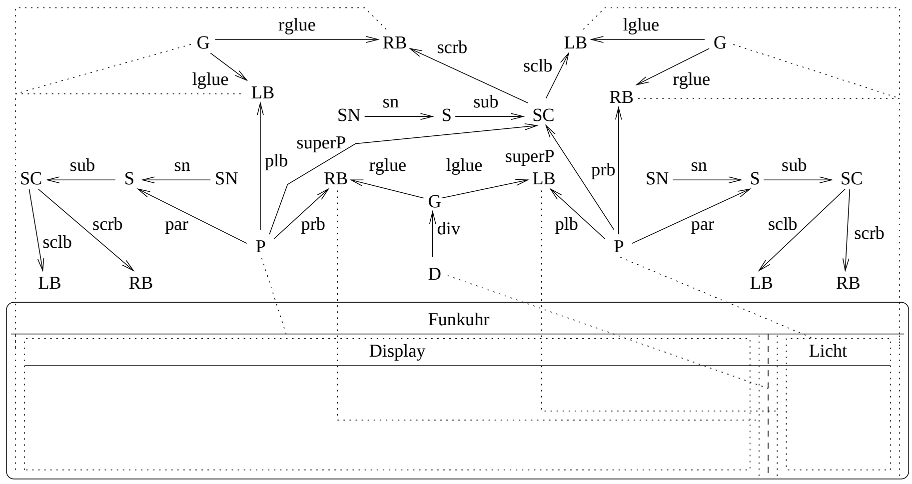

Abbildung 6.4: Links des Statechart-Alphabets, Parallelzustand, Teil 2.

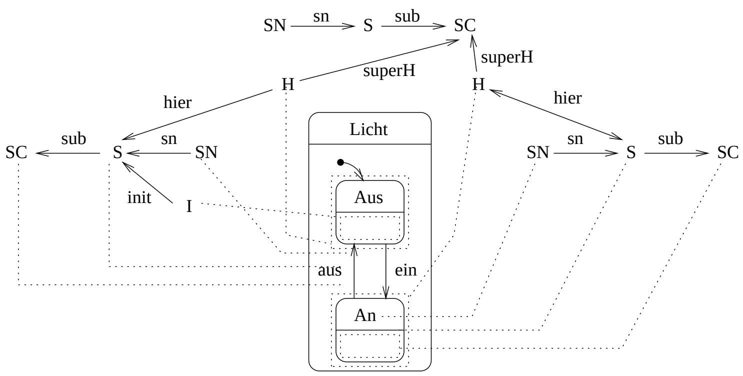

Abbildung 6.5: Links des Statechart-Alphabets, Hierarchiezustand, Teil 1.

Kommen wir nun zur Definition der Syntaxgrammatik. Die Abbildungen 6.7 und 6.8 zeigen die abstrakte Syntax aller sechs Regeln. Die Datentypen wurden nicht mit angegeben, aber deren Verwendung sollte klar sein. Das Startdiagramm der Grammatik ist leer, die einzige Regel die auf diese angewendet werden kann,

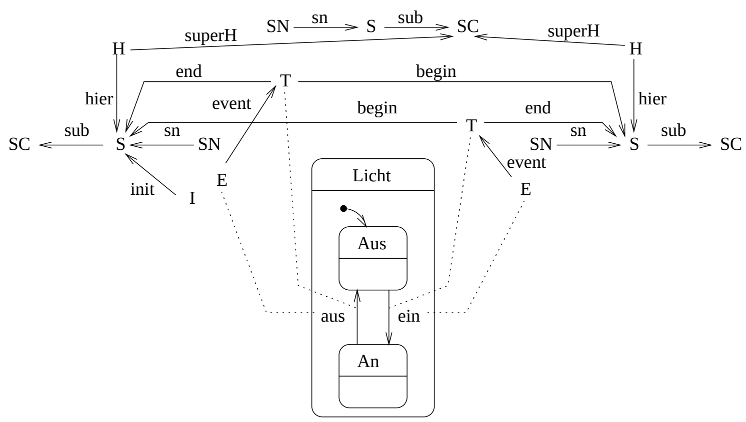

Abbildung 6.6: Links des Statechart-Alphabets, Hierarchiezustand, Teil 2.

ist „InsertRoot“. Aufgrund ihrer NAC ist die Anwendung tatsächlich auch nur auf das Startdiagramm möglich. Der einzige Grund, warum sie nicht weggelassen und dafür ihre rechte Seite als Startdiagramm verwendet wurde, ist die Tatsache, dass keine Regel zum Ändern eines Zustandsnamens vorgesehen ist, sodass es unmöglich wäre, dem Wurzelzustand einen frei gewählten Namen zu geben.

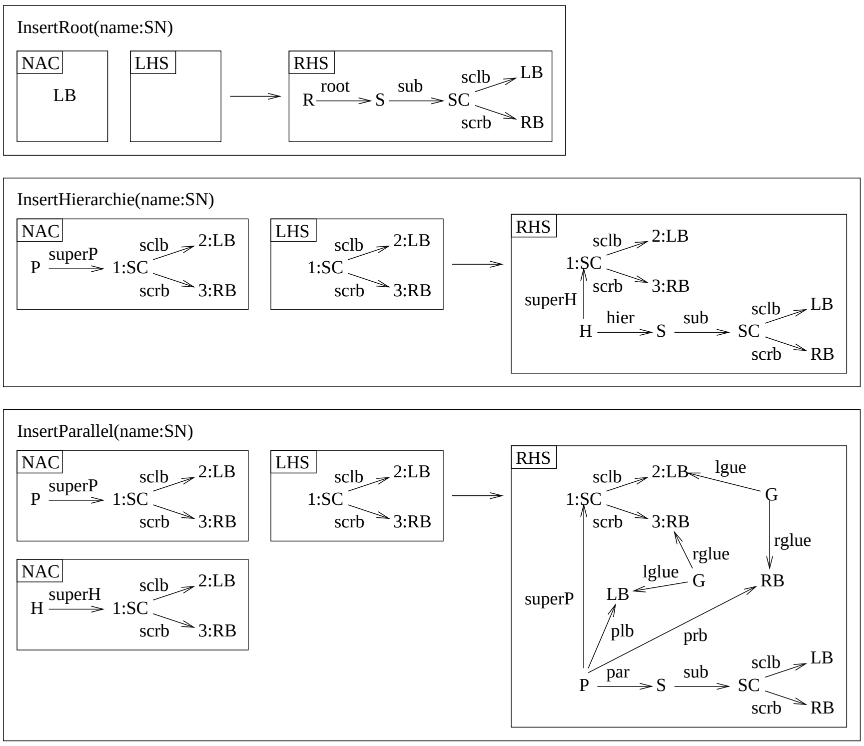

Abbildung 6.7: Statechart-Syntaxgrammatik, Teil 1.

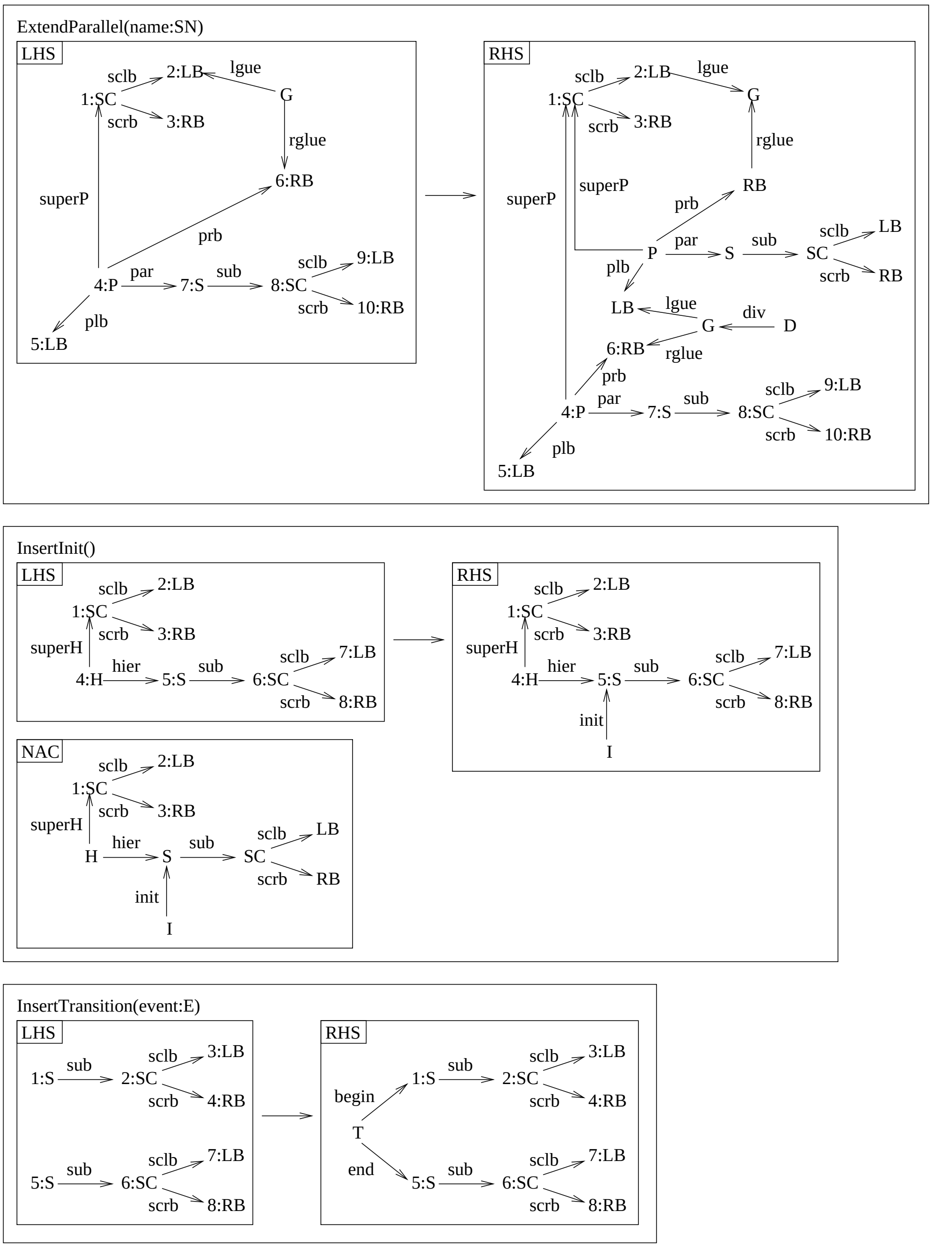

Abbildung 6.8: Statechart-Syntaxgrammatik, Teil 2.

Solange ein Zustand noch keine Unterzustände besitzt, ist sein Typ noch nicht festgelegt, erst mit dem Einfügen eines solchen, wird auch ein `H`- oder `P`-Symbol an sein `SC`-Symbol angehängt, was den Typ entsprechend determiniert. Aus diesem Grund besitzen die Regeln „InsertHierarchie" und „InsertParallel" auch eine NAC, die verhindert, dass die Regel auf einen Zustand des anderen Typs angewendet werden kann. Während „InsertHierarchie" beliebig oft für ein und den selben übergeordneten Zustand angewendet werden kann, um Unterzustände einzufügen, ist „InsertParallel“ nur für das Einfügen des ersten Unterzustands eines Parallelzustands vorgesehen. Dies liegt daran, dass der linke Rand des einzufügenden Unterzustands mit dem linken Rand des `SC`-Symbols des übergeordneten Zustands „verklebt“ werden muss, während die nachfolgenden Unterzustände jeweils an die rechte Seite des Vorgängers gehaftet werden. Dazu dient die Regel „ExtendParallel“. In beiden Fällen wird der rechte Rand des neuen Zustands mit der rechten Seite des übergeordneten `SC`-Symbols verklebt.

Die Regel „InsertInit“ markiert einen Unterzustand eines Hierarchiezustands als initial. Die NAC stellt, sicher, dass jeweils maximal ein initialer Zustand existieren kann. Die verbleibende Regel „InsertTransition“ fügt eine Transition zwischen zwei beliebigen Zuständen ein. Die Bedingung, dass Transitionen keine verschiedenen Unterzustände eines Parallelzustands (oder direkte oder indirekte Unterzustände) als Quelle und Ziel haben dürfen wird also nicht berücksichtigt. Ebenso wenig ist sichergestellt, dass jeder Hierarchiezustand auch genau einen initialen Unterzustand besitzt. Die Überprüfung dieser Bedingungen bleibt Aufgabe der Parsierung.

6.3 Parsierung

Die Parsierung soll als Ergänzung zum syntaxgesteuerten Editieren per Syntaxgrammatik dienen, um zwei Bedingungen sicherzustellen. Zum einen muss jeder Hierarchiezustand einen initialen Unterzustand besitzen und zum anderen darf keine Transition zwischen parallelen Zuständen existieren. Es muss also dafür gesorgt werden, dass in solchen Fällen das Diagramm nicht zum Stoppdiagramm der Parsierungsgrammatik abgebaut werden kann. Für beide Bedingungen wird das Diagramm deshalb erst analysiert, das heißt systematisch erweitert, bevor versucht wird, es mittels der restlichen Parsierungsregeln abzubauen.

Um sicherzustellen, dass es in jedem Hierarchiezustand einen initialen Unterzustand gibt, werden alle Unterzustände in einem Hierarchiezustand, wo das der Fall ist, markiert. Später werden dann nur die Zustände gelöscht, die markiert sind, sodass nur solche Statecharts vollständig abgebaut werden können, die die Bedingung erfüllen.

Die zweite Forderung ist etwas aufwendiger zu prüfen. Das liegt daran, dass die betroffenen Zustände entlang der Baumrelation beliebig weit vom Parallelzustand entfernt sein können, der die Ursache für die Verletzung der Bedingung darstellt. Die Lösung liegt darin, für jede Transition Transitionen zwischen allen Zuständen einzufügen, die oberhalb des Quell- und des Zielzustands der Transition liegen. Insbesondere würde, im Falle der Verletzung der Bedingung auch zwischen direkten Unterzuständen eines Parallelzustands eine Transition eingefügt. Es muss somit beim Abbau des Diagramms nur noch per NAC verboten werden eine solche Transition zu löschen, damit die Parsierung fehlschlägt.

Die Abbildungen 6.9 bis 6.12 zeigen alle Regeln der Parsierungsgrammatik. „CompleteInit“ nimmt die Markierung der Zustände für die erste Forderung vor, „CompleteTransSourceH“, „CompleteTransSourceP“, „CompleteTransTargetH“, und „CompleteTransTargetP“ erledigen das Einfügen der zusätzlichen Transitionen. Es werden vier Versionen benötigt, da zum einen zwischen Quell- und Zielzustand unterschieden werden muss und zum anderen Hierarchisierung und Parallelität nicht mit einer Regel abgehandelt werden können.

Ist die Erweiterung des Diagramms abgeschlossen, werden zuerst die Transitionen („DeleteTransition“) entfernt – natürlich nur solche, die nicht gegen die NAC, also die Forderung verstoßen. Dann folgen die Zustände („DeleteRoot“, „DeleteParallel“, „DeleteHierarchie“) und schließlich werden alle Hilfssymbole gelöscht („DeleteSC“, „DeleteD“, „DeleteG“, „DeleteB“).

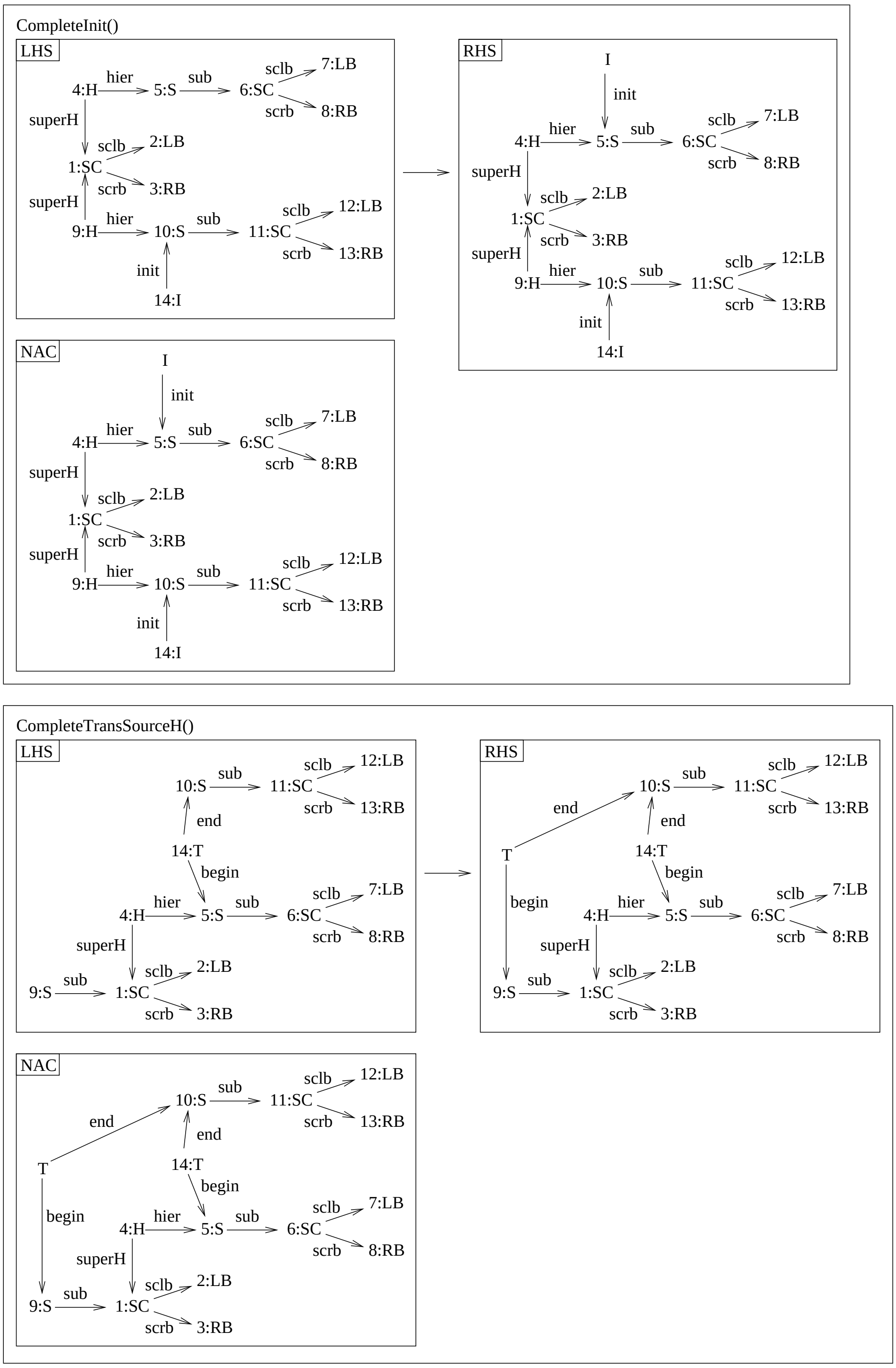

Abbildung 6.9: Statechart-Parsierungsgrammatik, Teil 1.

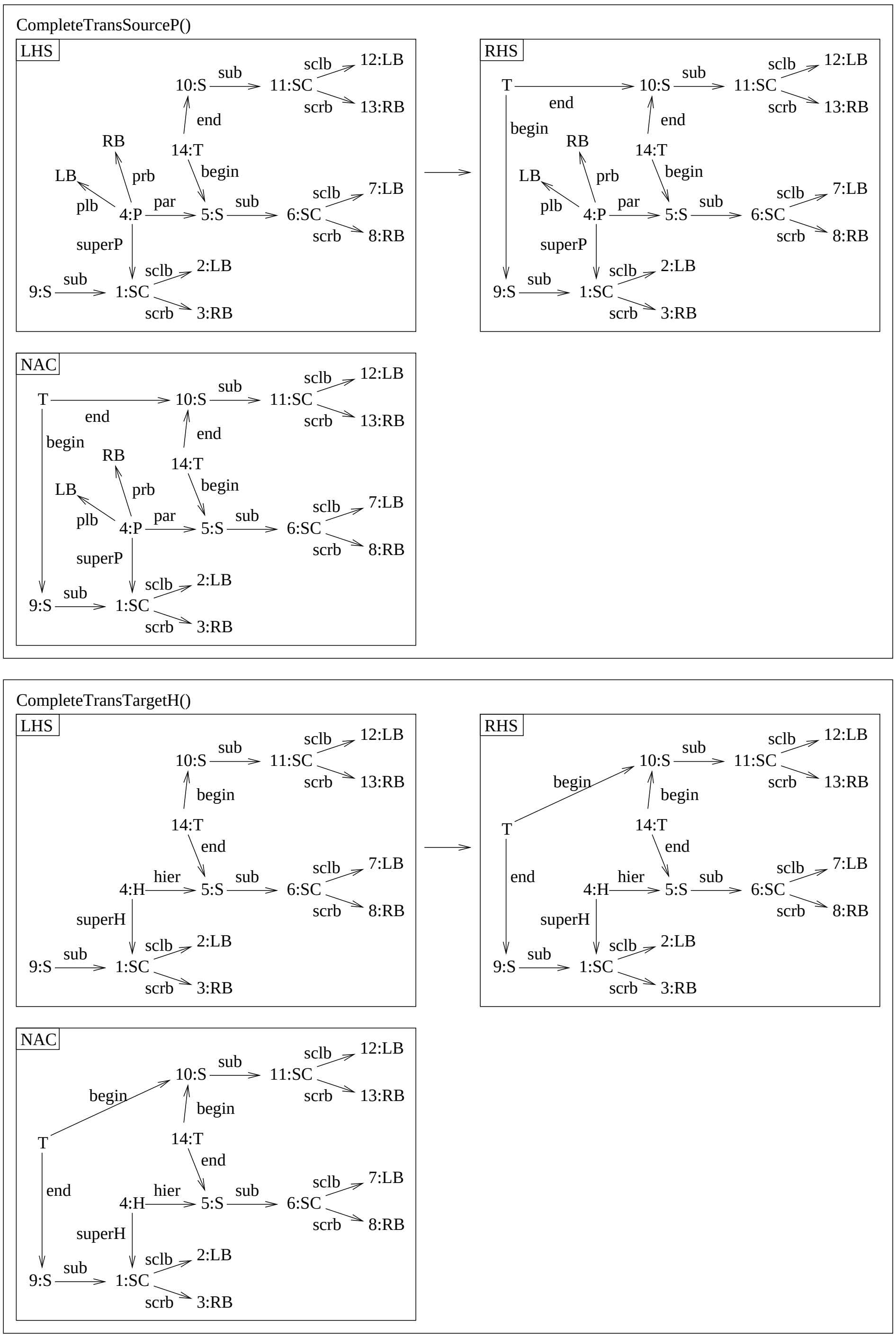

Abbildung 6.10: Statechart-Parsierungsgrammatik, Teil 2.

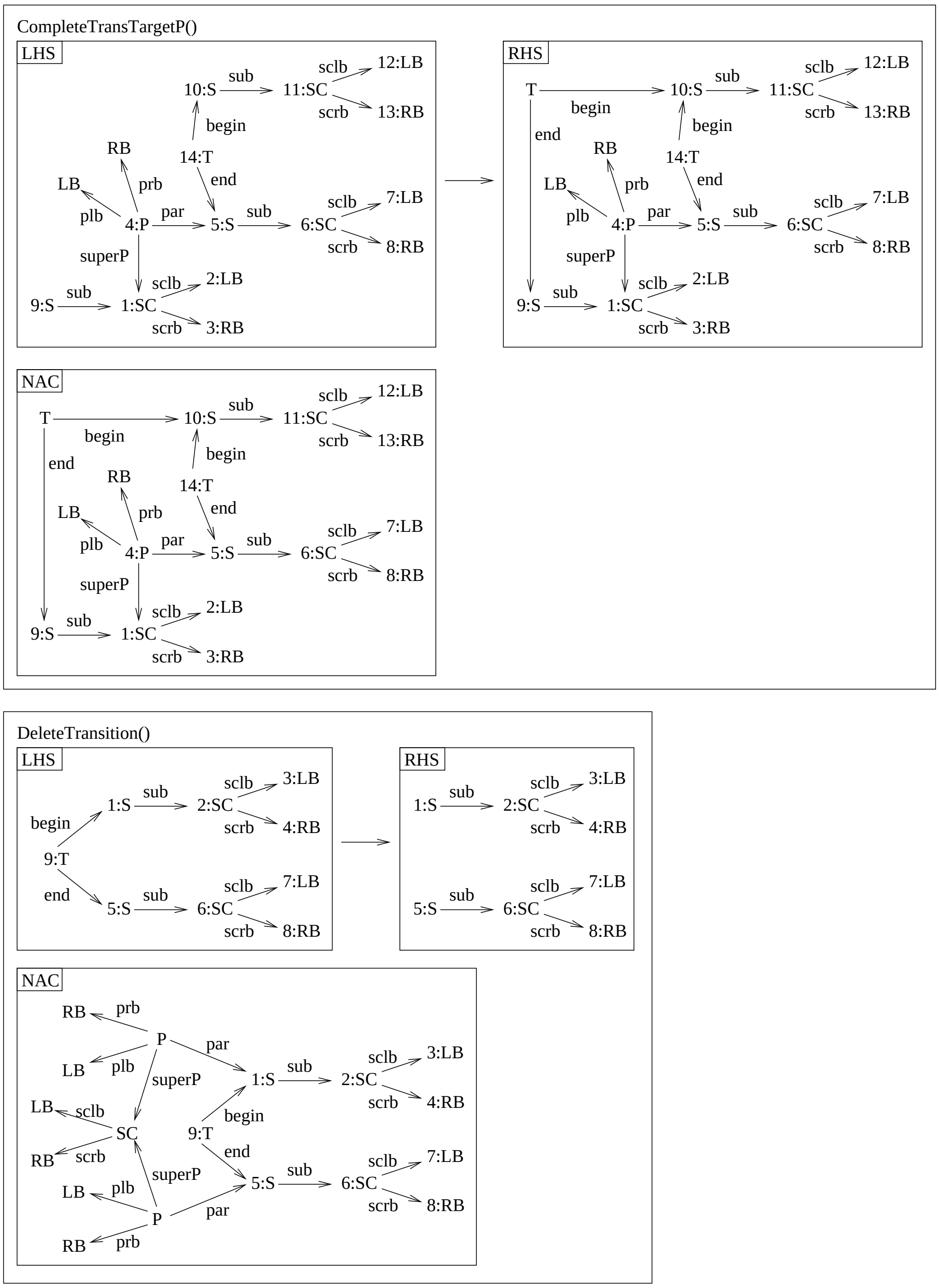

Abbildung 6.11: Statechart-Parsierungsgrammatik, Teil 3.

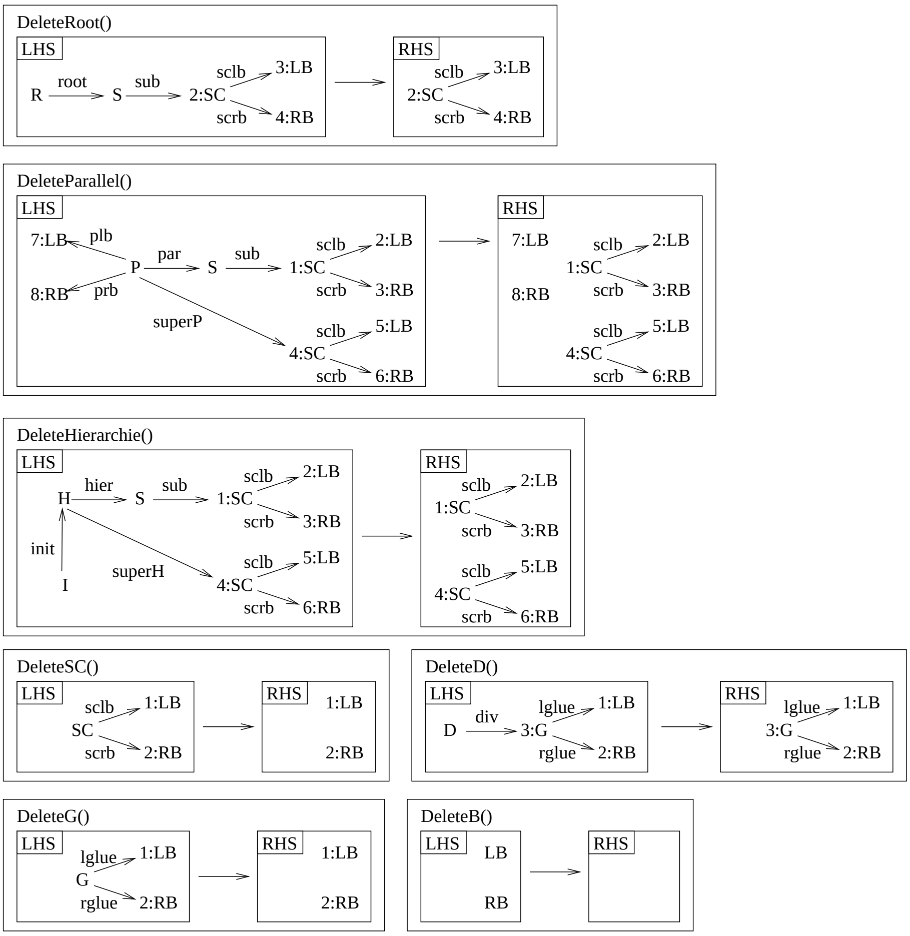

Abbildung 6.12: Statechart-Parsierungsgrammatik, Teil 4.

Damit überhaupt in dieser Reihenfolge vorgegangen wird, ist eine Layerfunktion vonnöten. Die Rule-Layer sind in Tabelle 6.3 aufgelistet.

Regel	Layer	Regel	Layer
CompleteInit	0		
CompleteTransSourceH	1	CompleteTransSourceP	1
CompleteTransTargetH	2	CompleteTransTargetP	2
DeleteTransition	3	DeleteRoot	4
DeleteParallel	5	DeleteHierarchie	5
DeleteSC	6	DeleteD	7
DeleteG	8	DeleteB	9

Tabelle 6.3: Layer der Regeln der Statechart-Parsierungsgrammatik.

6.4 Simulation

Um die definierten Statecharts simulieren zu können, müssen wir das ursprüngliche Alphabet um ein Symbol erweitern, das visualisiert, in welchen Zuständen sich ein Statechart befindet. Der Übersichtlichkeit halber werden wir in diesem Abschnitt die Hilfssymbole weglassen, die ausschließlich zur korrekten Positionierung der Grafiken verwendet werden – dies sind `LB`, `RB`, `G` und `D` – ebenso wie die Zustandsnamen. Das für die Simulation benutzte Alphabet ist in Abbildung 6.13 zu sehen.

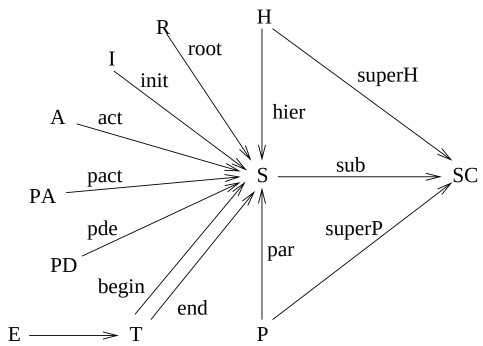

Abbildung 6.13: Statechart-Alphabet für die Simulation.

Neben dem Symbol `A`, das an einem Zustand hängt, wenn er aktiv ist, verwenden wir die Hilfssymbole `PA` und `PD`. Diese tauchen in einem Statechart nur auf, während ein Simulationsschritt im Gange ist. Sie dienen dazu, Zustände zu markieren, die aktiviert oder deaktiviert werden sollen. Ihre genaue Bedeutung wird bei der Erklärung der Simulationsschritte klarer. Die Simulationsschritte sind in Abbildung 6.14 angegeben, die Regeln über denen sie definiert sind, finden sich in den Abbildungen 6.15 bis 6.18.

Es werden zwei Simulationsschritte definiert: *init* fügt zu einem neu erstellten Statechart alle Informationen hinzu, die für die Simulation benötigt werden, und *step*, welches einen Event als Parameter bekommt, schaltet alle Transitionen, die mit dem Event beschriftet sind und einen aktiven Quellzustand besitzen.

Gehen wir etwas genauer auf den Simulationsschritt *init* ein: Zuerst versucht er, durch Anwendung der Regel „ActivateRoot“ an den Wurzelzustand ein `A`-Symbol zu hängen, welches ihn als aktiv markiert. Dies kann nur dann scheitern, wenn schon einmal *init* auf das Statechart angewendet wurde. Anschließend werden in einer Schleife sooft wie möglich die Regeln „InitHierarchy“ und „InitParallel“ angewendet. Erstere aktiviert den initialen Unterzustand eines aktiven Hierarchiezustands, die zweite aktiviert alle Unterzustände eines Parallelzustands. Die Menge der nach der Abarbeitung der Schleife aktiven Zustände heißt nach [Joh99] initiale Konfiguration.

Mit Hilfe des Simulationsschritts *step* wird jeweils eine Folgekonfiguration be-

$$
\begin{array}{lcl}
\underline{init} & = & (\emptyset, initExpr) \\
initExpr & = & \mathbf{rule}(ActivateRoot, \emptyset) \\
 & & \mathbf{and\ while} \\
 & & \quad \mathbf{rule}(InitHierarchy, \emptyset) \\
 & & \quad \mathbf{or\ rule}(InitParallel, \emptyset) \\
 & & \mathbf{do\ true\ done} \\
\underline{step} & = & (e, stepExpr) \\
stepExpr & = & \mathbf{while} \\
 & & \quad \mathbf{rule}(TriggerTransition, \{e \mapsto e\}) \\
 & & \mathbf{do\ true\ done} \\
 & & \mathbf{or\ while} \\
 & & \quad \mathbf{rule}(DeactivateTarget, \emptyset) \\
 & & \mathbf{do\ true\ done} \\
 & & \mathbf{or\ while} \\
 & & \quad \mathbf{rule}(PrepareDeactivationH, \emptyset) \\
 & & \quad \mathbf{or\ rule}(PrepareDeactivationP, \emptyset) \\
 & & \mathbf{do\ true\ done} \\
 & & \mathbf{or\ while} \\
 & & \quad \mathbf{rule}(Deactivate, \emptyset) \\
 & & \mathbf{do\ true\ done} \\
 & & \mathbf{or\ while} \\
 & & \quad \mathbf{rule}(CleanupPD, \emptyset) \\
 & & \mathbf{do\ true\ done} \\
 & & \mathbf{or\ while} \\
 & & \quad \mathbf{rule}(Activate, \emptyset) \\
 & & \mathbf{do\ true\ done} \\
 & & \mathbf{or\ while} \\
 & & \quad \mathbf{rule}(CleanupPA, \emptyset) \\
 & & \mathbf{do\ true\ done} \\
 & & \mathbf{or\ while} \\
 & & \quad \mathbf{rule}(ActivateParentH, \emptyset) \\
 & & \quad \mathbf{or\ rule}(ActivateParentP, \emptyset) \\
 & & \mathbf{do\ true\ done} \\
 & & \mathbf{or\ while} \\
 & & \quad \mathbf{rule}(InitHierarchy, \emptyset) \\
 & & \quad \mathbf{or\ rule}(InitParallel, \emptyset) \\
 & & \mathbf{do\ true\ done}
\end{array}
$$

Abbildung 6.14: Simulationsschritte für Statecharts.

rechnet. Diese Berechnung ist, wie an der Größe des Simulationsausdrucks des Schritts zu erkennen ist, sehr aufwendig. Es sei darauf hingewiesen, dass der Aufwand noch größer gewesen wäre, wenn die in [Joh99] definierte Schrittsemantik implementiert worden wäre. Stattdessen kann nun für manche Statecharts die Simulation fehlschlagen. Doch werfen wir einen genaueren Blick auf den Simulationsausdruck. Er besteht aus durch **or** verknüpften **while do done**-Schleifen, in denen jeweils eine oder zwei Regeln solange angewendet werden, wie dies möglich ist.

Begonnen wird mit der Anwendung der Regel „TriggerTransition“, die Quell- und Zielzustand jeder zu schaltenden Transition mit einem `PD`- bzw. `PA`-Symbol versieht. Die Aktiviertheit von Zuständen ändert sich zu diesem Zeitpunkt noch nicht. Die Quellzustände werden als zu deaktivieren und die Zielzustände als zu aktivieren vorgemerkt. Die folgende Anwendung von „DeactivateTarget“ markiert die Zielzustände zusätzlich auch noch als zu deaktivieren. Auch wenn dies möglicherweise merkwürdig erscheinen mag, hat es seinen Sinn, denn das erste zu erreichende Etappenziel ist die Deaktivierung sämtlicher Zustände unterhalb des Scopes der Transitionen (siehe dazu [Joh99]). Erst danach werden die noch immer als zu aktivieren gekennzeichneten Zielzustände, samt ihrer übergeordneten Zustände, tatsächlich aktiviert.

Doch schreiten wir nicht zu schnell voran, denn zuerst müssen rekursiv die Unterzustände von zu deaktivierenden Zuständen entsprechend markiert werden. Dies erledigen die Regeln „PrepareDeactivationH“ und „PrepareDeactivationP“ für Hierarchie- und Parallelzustände. Anschließend werden mittels „Deactivate“ die so markierten Zustände deaktiviert. Es folgt „CleanupPD“, welches alle noch übrig gebliebenen `PD`-Symbole entfernt.

Nachdem die Zustandsdeaktivierung damit vollständig abgeschlossen ist, wird „Activate“ angewendet, um die als zu aktivieren markierten Zustände zu aktivieren. „CleanupPA“ verfährt analog zu „CleanupPD“, indem es die verbliebenen `PA`-Symbole entfernt. Die Regeln „ActivateParentH“ und „ActivateParentP“ aktivieren die übergeordneten Hierarchie- bzw. Parallelzustände von aktivierten Zuständen. Als letztes werden mittels der schon erwähnten Regeln „InitHierarchy“ und „InitParallel“ die initialen Zustände aktivierter Hierarchiezustände und alle Unterzustände von aktivierten Parallelzuständen aktiviert.

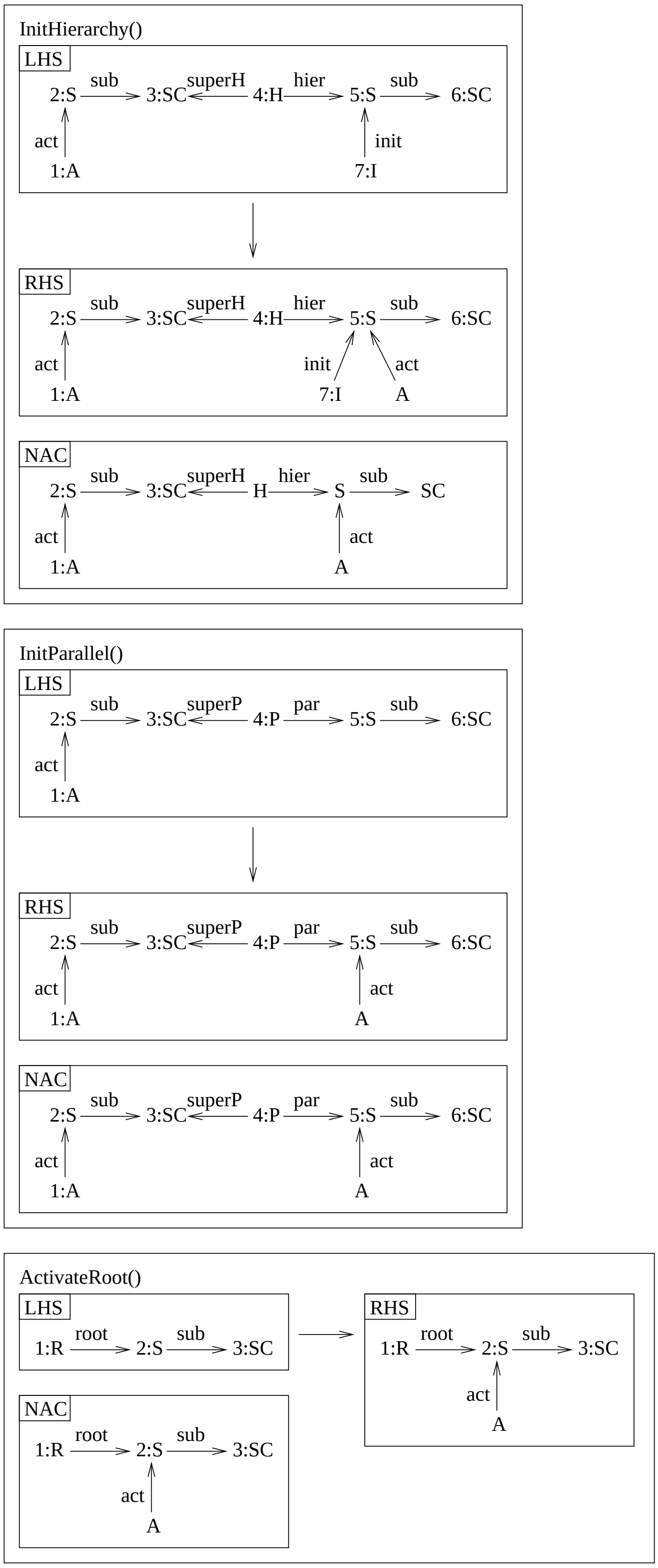

Abbildung 6.15: Statechart-Simulationsgrammatik, Teil 1.

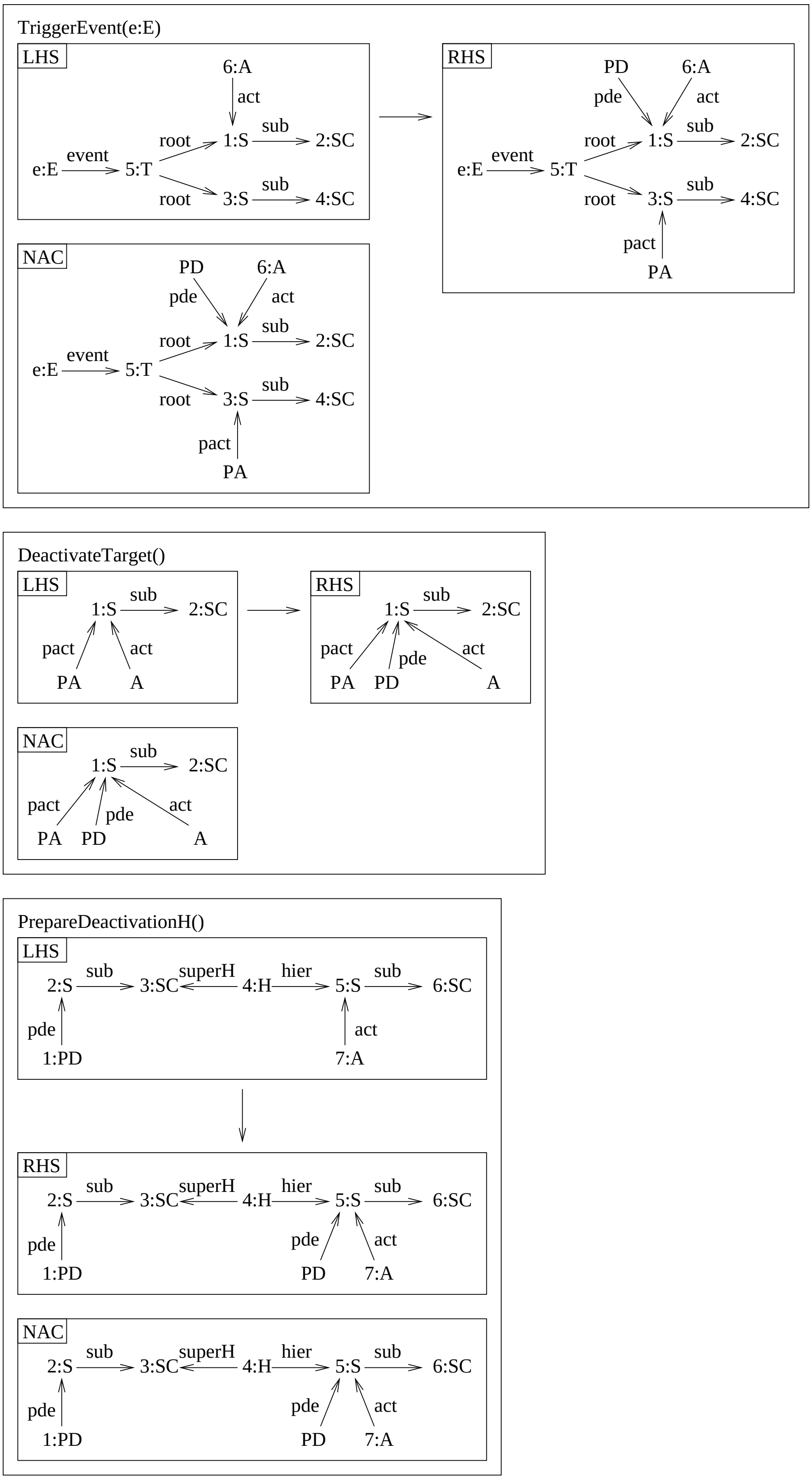

Abbildung 6.16: Statechart-Simulationsgrammatik, Teil 2.

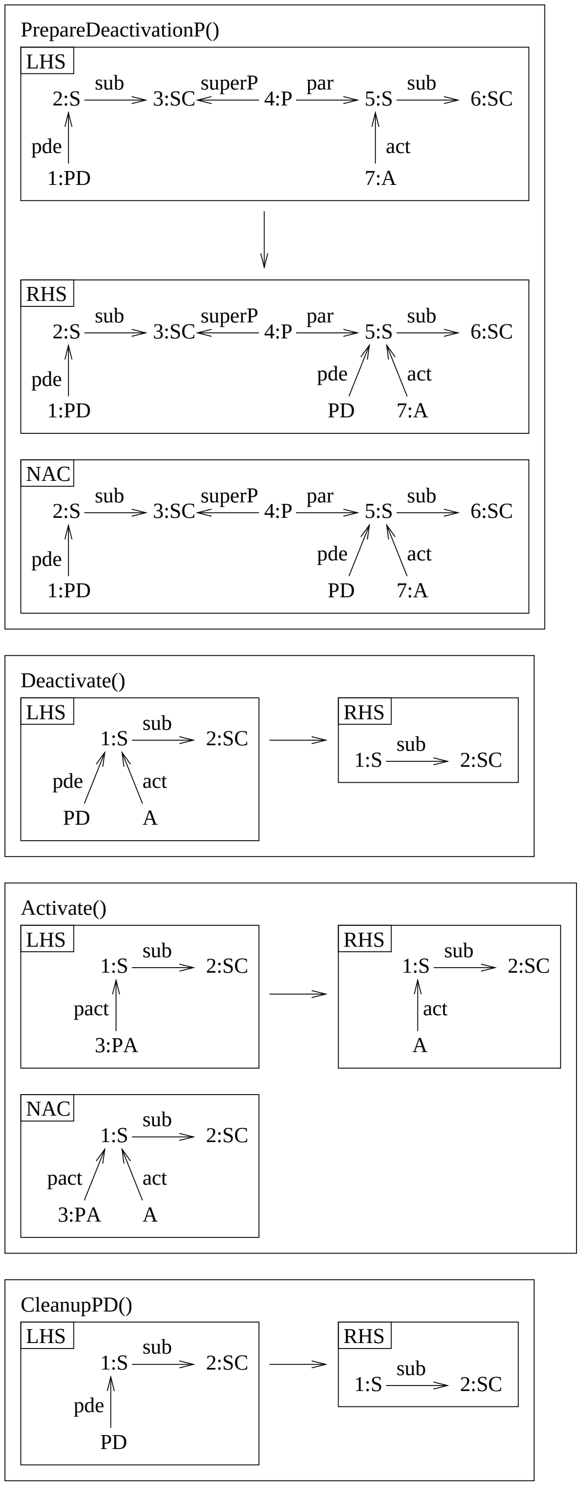

Abbildung 6.17: Statechart-Simulationsgrammatik, Teil 3.

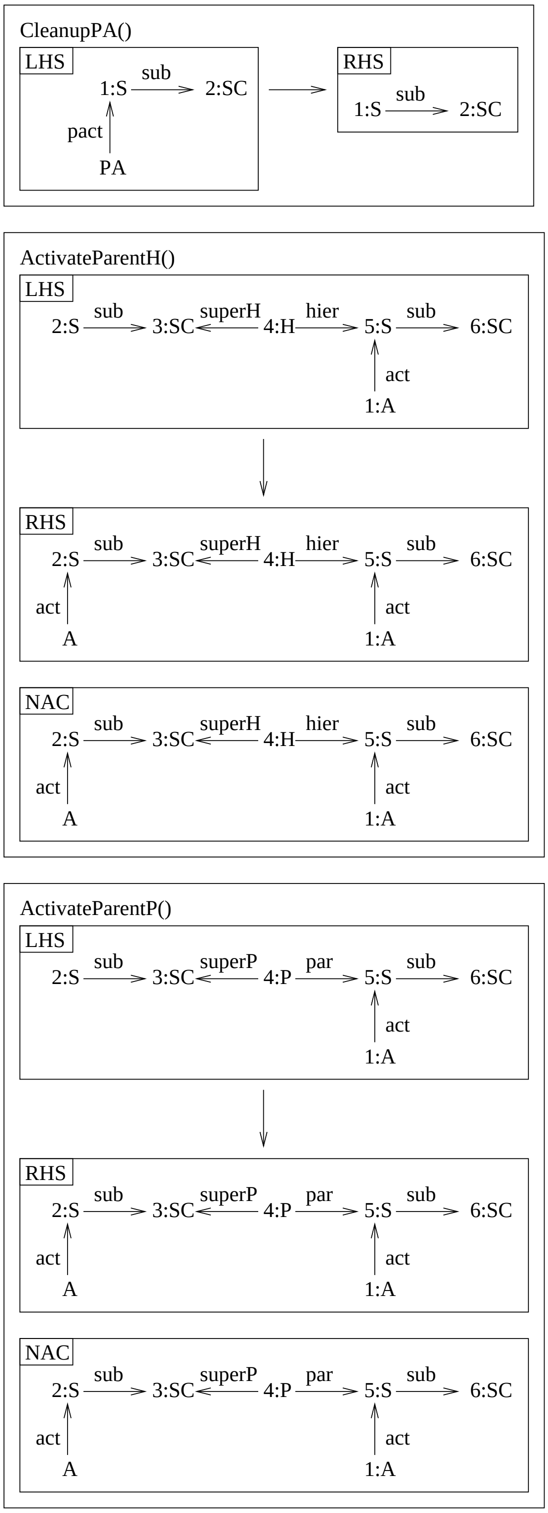

Abbildung 6.18: Statechart-Simulationsgrammatik, Teil 4.

Kapitel 7

Ergebnisse und Ausblick

Dieses Kapitel fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche Verbesserungen und Erweiterungen. Es werden dabei neben kleineren Änderungen auch solche vorgeschlagen, die größerer Umstrukturierungen bedürfen oder gar visionären Charakter besitzen.

7.1 Ergebnisse

Die Arbeit begann mit einer Einführung in die schon ausgearbeiteten Konzepte von GenGEd ([Bar00]). Als visuelle Sprachen wurden Mengen von visuellen Sätzen über einem visuellen Alphabet definiert. Ein Alphabet besteht aus Symbolen, Datentypen und Links, letztlich basierend auf grafischen Primitiva verbunden durch grafische Constraints. Formalisiert kann ein Alphabet als algebraische Signatur verstanden werden, eine Algebra über dieser Signatur als visueller Satz. Visuelle Grammatiken, angelehnt an algebraische Graphgrammatiken, sind ein Mittel, um mit Hilfe eines endlichen Regelsystems eine visuelle Sprache zu beschreiben. Einerseits konstruktiv durch eine Syntax-, andererseits deskriptiv durch eine Parsierungsgrammatik.

Die Konzepte zur Parsierung wurden adaptiert aus einer Diplomarbeit über attributierte Graphparsierung und deren Implementation im Agg-System ([Sch01]), die Simulationskonzepte wurden völlig neu entwickelt. Basis einer Simulationsspezifikation sind Simulationsschritte, deren Kern wiederum die Simulationsausdrücke bilden. Der wichtigste Simulationsausdruck ist der **rule**()-Ausdruck, parametrisiert mit einer Regel und einer Parameterbelegung für diese Regel. Die weiteren Ausdrücke sind logischer Natur (**and**, **andc**, **or**, **orc**) oder defineren programmatische Konstrukte (**if then else fi**, **while do done**). Die eigentliche Simulation erfolgt als eine Art Seiteneffekt der Auswertung der Simulationsausdrücke – der **rule**()-Ausdruck wendet eine Regel an, alle anderen Ausdrücke steuern die Auswertung.

Nachdem die Konzepte vorgestellt waren, erfolgte eine Analyse des Ist-Zustands des GenGEd-Systems. Die Funktion und die für diese Arbeit relevanten Teile der Implementation wurden näher beleuchtet und Probleme und Schwächen herausgestellt. Zu den verwendeten Bibliotheken und Applikationen, wie Java, ParCon, ParCon-Daemon und Agg, fand ebenfalls eine Bestandsaufnahme statt, inklusive Hinweise auf die Änderungen, die seit 1999 durchgeführt wurden.

Nach einer Auflistung der implementatorischen Anforderungen erfolgte die Beschreibung der Realisierung. Es wurden neue Datenstrukturen für die Grammatiken entwickelt, ein Transformationsrahmenwerk, das von der konkreten Implementation der Transformation (in diesem Fall mittels Agg) abstrahiert, ein Rahmenwerk für die Parsierung und Strukturen für Simulation und VL-Spezifikationen. Zur Ermöglichung eines Austauschs der abgespeicherten Alphabete, Diagramme und Grammatiken ist das Abspeichern und Laden von XML-Dateien unter Verwendung der GTS DTD ([GXL01]) implementiert worden; für VL-Spezifikationen wird eine eigens entwickelte DTD verwendet. Den Erklärungen zu den grundlegenden Strukturen und der Realisierung ihrer Persistenz folgte schließliche eine Beschreibung der Implementation der grafischen Benutzerschnittstelle.

Quasi als „proof of principle“ der Eignung der GenGEd-Konzepte zur Spezifikation einer VL-Umgebung für eine in der Praxis verwendete visuelle Sprache, folgt ein Kapitel, in dem für eine Statechart-Variante Alphabet, Syntaxgrammatik und Parsierungs- und Simulationsspezifikation definiert werden.

Ein Benutzerhandbuch und eine API-Dokumentation runden die Arbeit ab.

7.2 Ausblick

Es sollen nun einige Punkte zur Verbesserung und zum Ausbau von GenGEd genannt werden. Die Liste beginnt mit notwendigen Veränderungungen und solchen Erweiterungen, deren Implementation weniger aufwendig sein sollte, und endet mit Ideen, die umfangreich genug sind, um als Themen weiterer Diplomareiten zu dienen.

- Ein möglicherweise nicht leicht zu lösendes Problem, welches aber von enormer Bedeutung für die praktische Benutzbarkeit von GenGEd ist, betrifft die CSPs. Sobald ein Diagramm eine gewisse Größe, das heißt Anzahl von Symbolen und Links, insbesondere aber mit diesen assoziierten Constraints, erreicht hat, ist der Constraintsolver ParCon nicht mehr in der Lage, das zugehörige CSP auch nur annähernd in Echtzeit zu lösen. Das liegt nicht daran, dass ParCon ein langsamer Constraintsolver ist, sondern an der Art, wie die Constraints in GenGEd verwendet werden. Die Anzahl der Constraints wächst linear mit der Anzahl der Symbole und Links, doch, zumindest wenn Oder-Constraints verwendet werden, die Größe des Suchraums für das CSP exponentiell.

Einen Lösungsansatz kann der Autor leider nicht anbieten. Eine Richtung, in die die Überlegungen gehen könnten, wäre, Constraints für häufig auftretende Anwendungen, wie das Ankoppeln von Pfeilen oder Linien an andere Objekte, aus den Constraint Networks herauszunehmen und durch einen eigenen Algorithmus zu bearbeiten. Ebenfalls hilfreich könnte es sein, darüber nachzudenken, ob es möglich ist das CSP für ein Diagramm in kleinere CSPs zu zerlegen. Denkt man etwa an die hierarchischen Statecharts, so ist es durchaus vorstellbar, erst die CSPs der inneren States zu lösen und sich dann entlang der Hierarchie stufenweise nach außen vorzuarbeiten und dabei die Werte der Variablen der schon gelösten CSPs als Konstanten zu betrachten.

- Gewöhnlich soll in einem Diagramm vermieden werden, dass sich Symbole der gleichen Sorte überlappen. Mit GenGEd ist es derzeit nicht möglich, eine solche Forderung zu realisieren. Dies liegt nicht unbedingt daran, dass keine zyklischen Alphabete und damit Links zwischen Symbolen einer Sorte möglich sind, denn mit Hilfe von Diagrammconstraints lässt es sich auch bewerkstelligen. Ein Diagrammconstraint gehört nicht zu einem Symbol oder Link eines Diagramms sondern zum Diagramm selbst und ist deshalb nicht beschränkt auf das eine oder die zwei beteiligten Symbole, sondern kann beliebig viele Objekte referenzieren. Um Diagrammconstraints einzuführen, müssen die Regeln entsprechend erweitert werden.

- Im Abschnitt 3.2.2.7 wurde Kritik an der Handhabung der Datentypen geübt. Die eigentlichen Daten sollen von deren Visualisierung getrennt werden, damit es möglich ist, zum Beispiel auch Java-Klassen zu verwenden. Zur grafischen Darstellung und zum Editieren sind separate Klassen vorzusehen.

- Abschnitt 5.8.2 erwähnt, dass für das Speichern des grafischen Layouts der Symbole keine spezielle DTD verwendet wird. Sollte ein Austauschformat für Grafiken vorliegen und dieses für die von GenGEd verwendeten grafischen Daten adäquat sein, so ist es sinnvoll dies zu implementieren.

- Die schon rudimentär implementierte Fähigkeit, Grammatiken an veränderte Alphabete anzupassen (Abschnitt 5.3.3.5) kann stark erweitert werden, insbesondere durch Unterstützung von Benutzerinteraktionen.

- Im VL-Editor (syntaxgesteuertes Editieren in der VL-Umgebung) nimmt die Anzeige der Regel erheblichen Platz weg und verkleinert somit die Sichtfläche dessen, was eigentlich von Interesse ist, das editierte Diagramm. Bisher ist dies aber nötig, da es die einzige Möglichkeit darstellt, gezielt einen Match zu definieren, denn die automatische Matchvervollständigung liefert bei größeren Diagrammen selten das Gewünschte. Erstrebenswert wäre es also, eine Benutzerführung für die Matchdefinition ohne Ansicht der Regel zu entwickeln.

- Die Fähigkeiten der Simulationskomponente sind derzeit noch recht beschränkt – sie unterstützt nur vom Benutzer angestoßene Einzelschritte. Wünschenswert ist jedoch auch die Möglichkeit eine Simulation über mehrere Schritte automatisch durchführen zu können und eventuell einen Baum (oder Graphen) möglicher Simulationzustände angezeigt zu bekommen. Weitere Werkzeuge zur Analyse sind denkbar.

- Java Beans definieren ein Modell für Javakomponenten. Es ist vorstellbar, generierte VL-Umgebungen als Java Beans zu realisieren und dadurch die Einbettung in andere Applikationen zu ermöglichen.

- Häufig werden Softwaremodelle mit Hilfe verschiedener visueller Sprachen modelliert. Üblich ist zum Beispiel eine Trennung der statischen und der dynamischen Aspekte des Systems. Da letztlich immer das gleiche System, eben nur aus verschiedenen Sichten, beschrieben wird, ist es sinnvoll eine integrierte Umgebung anzubieten, in der alle Aspekte bearbeitet werden können. Eine Änderung in einer Sicht hat dann möglicherweise Auswirkungen auf andere Sichten, die entsprechend angepasst werden müssen. Die Ausarbeitung der Konzepte und die Erweiterung von GenGEd zu einem solchen System ist sicher erstrebenswert, wenn zugegebenermaßen auch recht komplex.

Literaturverzeichnis

[Bar00] R. Bardohl. GenGEd – *Visual Definition of Visual Languages based on Algebraic Graph Transformation.* Verlag Dr. Kovac, 2000. PhD thesis, Technical University of Berlin, Dept. of Computer Science, 1999.

[BST01] R. Bardohl, T. Schultzke, and G. Taentzer. Visual Language Parsing in GenGEd. *Electronic Notes of Theoretical Computer Science*, 50(3), June 12–13 2001.

[Con01] W3C World Wide Web Consortium. *Extensible Markup Language (XML)*, 2001. Available at `http://www.w3.org/XML/`.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter 4, pages 247–312. World Scientific, 1997.

[ERT99] C. Ermel, M. Rudolf, and G. Taentzer. The AGG-approach: Language and tool environment. In H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, pages 551–604. World Scientific, 1999.

[Fou01] The Apache Software Foundation. *The Apache XML Project*, 2001. Available at `http://xml.apache.org`.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[Gmb86] F.A. Brockhaus GmbH. *Brockhaus-Enzyklopädie: in vierundzwanzig Bänden – Neunzehnte, völlig neu bearbeitete Auflage.* F.A. Brockhaus GmbH, 1986. Mannheim, 1986.

[Gri96] P. Griebel. *ParCon - Paralleles Lösen von grafischen Constraints.* PhD thesis, University of Paderborn, Germany, 1996.

[GXL01] *XML–based Exchange Formats for Graphs and Graph Transformation Systems*, 2001. Available at `http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html`.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[Joh99] S. John. Zur kompositionalen Semantik von objekt–orientierten Statecharts. Master's thesis, TU Berlin, 1999.

[LB93] M. Löwe and M. Beyer. AGG — An Implementation of Algebraic Graph Rewriting. In *Proc. Fifth Int. Conf. Rewriting Techniques and Applications*, number 690 in LNCS, pages 451–456. Springer, 1993.

[LKW93] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[Mel98] B. Melamed. Design and Realisation of an Attribute Manager for Conditional and Distributed Graph Transformation. Master's thesis, TU Berlin, 1998.

[Nie99] M. Niemann. Konzeption und Implementierung eines generischen Grammatikeditors für visuelle Sprachen. Master's thesis, TU Berlin, 1999.

[Rud97] Michael Rudolf. Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation. Master's thesis, TU Berlin, 1997.

[Sch99] M. Schwarze. Konzeption und Implementierung eines generischen Alphabeteditors für visuelle Sprachen. Master's thesis, TU Berlin, 1999.

[Sch01] T. Schultzke. Konzeption und Implementierung eines Parsers für visuelle Sprachen. Master's thesis, TU Berlin, 2001.

[UML01] *Rational Software Cooperation. Unified Modeling Language – Version 1.3*, 2001. Available at `http://www.rational.com/uml/`.

Anhang A

Benutzerhandbuch

Dieses GenGEd-Benutzerhandbuch ist eine Ergänzung der Handbücher für Alphabet- ([Sch99]), Grammatik- und VL-Editor ([Nie99]). Es wiederholt nicht die Beschreibungen dieser Editoren, sondern erläutert nur die Modifikationen und Erweiterungen, die an ihnen vorgenommen wurden. Alle neuen Komponenten werden vollständig beschrieben.

Als Basis für die grafische Benutzerschnittstelle wurde das GUI des Alphabeteditors herangezogen. Der Constrainteditor und das Hilfesystem wurden aus der Liste der Untereditoren entfernt, Grammatikeditor, VL-Spezifikationseditor und VL-Umgebung sind hinzugekommen. Wird GenGEd in einem besonderen Modus gestartet, ist nur die VL-Umgebung verfügbar.

In jedem der Untereditoren gibt es ein *Extras*-Menü zum Ändern globaler Einstellungen, wie Sprache und ob die Einstellungen gespeichert werden sollen, und ein nicht funktionales *Hilfe*-Menü. Beide sind aus dem Alphabeteditor bekannt und werden nicht weiter erläutert.

Bevor mit der Beschreibung der grafischen Komponenten begonnen wird, erklärt der erste Abschnitt, wie das GenGEd-System zu installieren und zu starten ist.

A.1 Installation und Start

Es wird im Folgenden davon ausgegangen, das der Quellcode des GenGEd-Systems vorliegt. Dieser ist jedoch nur für eine kleine Gruppe zugänglich, weshalb für andere die ersten zwei Unterabschnitte nicht von Belang sind. Ihnen liegt vermutlich eine Binärdistribution vor – bisher wurde noch keine Version veröffentlicht – in der, falls überhaupt notwendig, weitere Instruktionen enthalten sind.

GenGEd wurde unter den Betriebssystemen Linux und Solaris (SunOS) getestet und funktioniert auch nur unter diesen. Java selbst ist zwar plattformunabhängig, doch der verwendete Constraintsolver ParCon liegt uns nur in einer

Linux- und einer Solaris-Version vor.

Es muss eine Java-Version ab 1.3 verwendet werden, sonst kann GenGEd nicht übersetzt werden oder es können während der Arbeit Fehler auftreten. Mindestvoraussetzungen für die Hardware müssen eigentlich nicht genannt werden – auf einem Rechner, der nicht älter als zwei Jahre ist, sollte GenGEd in akzeptabler Geschwindigkeit laufen. Der Autor arbeitete auf einem Notebook mit 128 MB Speicher und einem 600 MHz Pentium III Prozessor und war zumindest nicht unzufrieden. Ein Mehr an Speicher und Prozessorleistung könnte das Arbeiten jedoch flüssiger machen.

A.1.1 `configure`

Bevor der GenGEd-Quellcode übersetzt werden kann, muss das Shell-Skript `genged/configure` aufgerufen werden. Dieses stellt fest, unter welchem Betriebssystem es sich befindet und generiert aus der Datei `genged/src/Makefile.in` die Datei `genged/src/Makefile` (das globale Makefile für GenGEd); aus `genged/bin/GenGEd.in` wird die Datei `genged/bin/GenGEd` (das Startskript für GenGEd) generiert.

`configure` kann verschiedene optionale Parameter bekommen, von denen keiner benötigt wird, wenn die Java-Werkzeuge einer Version, die aktuell genug ist, mit `java`, `javac`, `javadoc` bzw. `jar` ohne Angabe eines Pfades aufgerufen werden können.

`--javapath` ***path***: *path* legt den Pfad der Java-Installation fest, die verwendet werden soll. Im Unterverzeichnis `bin` werden die Werkzeuge gesucht.

`--java` ***command***: *command* legt das Kommando zum Starten des Java-Interpreters fest.

`--javaflags` ***flags***: *flags* sind zusätzliche dem Java-Interpreter zu übergebende Parameter.

`--javac` ***command***: *command* legt das zum Kompilieren der Java-Klassen des GenGEd-System zu verwendende Kommando fest.

`--javacflags` ***flags***: *flags* sind zusätzliche dem Java-Compiler zu übergebende Parameter.

`--javadoc` ***command***: *command* legt das Kommando zum Erzeugen der API-Dokumentation fest.

`--javadocflags` ***flags***: *flags* sind zusätzliche dem Kommando, das die API-Dokumentation erzeugt, zu übergebende Parameter.

`--jar` ***command***: *command* legt das Kommando zum Archivieren von Java-Klassen in JAR-Dateien fest.

`--parcon` ***command***: *command* legt das Kommando zum Starten von ParCon fest.

`--parcondport` ***port***: *port* legt den Port fest, auf dem der ParCon-Daemon kommunizieren soll.

A.1.2 Kompilation

Zum Generieren der zu erzeugenden Dateien wird ein Makefile verwendet, genauer ein Makefile für GNU-Make (GNU-Make wird im Folgenden mit `make` bezeichnet). Dieses Makefile definiert mehre Ziele, von denen `genged` voreingestellt ist:

`world`: Erzeugt sämtliche zum GenGEd-System gehörenden Dateien.

`genged`: Übersetzt die Java-Klassen. Die kompilierten Dateien werden in `genged/bin` abgelegt.

`parcond`: Übersetzt den ParCon-Daemon. Die ausführbare Datei wird in `genged/bin` erzeugt.

`doc`: Übersetzt die im Quellcode vorliegenden Dokumentationen (z. B. diese Arbeit). Die Dateien werden in `genged/doc` abgelegt.

`clean`: Löscht sämtliche von „`make world`" erzeugte Dateien.

`clean_genged`: Löscht sämtliche von „`make genged`" erzeugte Dateien.

`clean_parcond`: Löscht sämtliche von „`make parcond`" erzeugte Dateien.

`distclean`: Wie `clean`, doch werden zusätzlich das Makefile und des Startskript gelöscht, sodass die Quellcodedistribution wieder im Ausgangszustand ist.

A.1.3 Programmstart

Der Start des GenGEd-Systems erfolgt durch Aufruf des in `genged/bin` befindlichen Startskripts `GenGEd`. Dieses darf nicht von dort fortbewegt werden, einen symbolischen Link darauf anzulegen verursacht jedoch keine Probleme.

Das Skript prüft zuerst, ob schon ein ParCon-Daemon im System läuft, und startet einen, wenn dies noch nicht der Fall ist. Danach wird der Java-Interpreter aufgerufen. Sämtliche Argumente, die an das Startskript übergeben wurden, werden an GenGEd weitergereicht. Der einzige bisher berücksichtigte Parameter ist `--vlenvironment`, bei dem nur die VL-Umgebung bereitgestellt wird.

A.2 Alphabeteditor

Das GUI des Alphabeteditors wurde nur geringfügig verändert. Abgesehen von einer Ansicht des Alphabets als Signatur, die mittels *Signatur anzeigen ...* im *Ansicht*-Menü aufzurufen ist, sind nur einige kleinere Korrekturen vorgenommen worden, etwa wo die Internationalisierung nicht konsequent implementiert war.

Die größte Änderung betrifft das Speichern und Laden von Alphabeten und Diagrammen: Gespeichert werden können diese Daten nur noch im XML-Format. Das Laden von Alphabeten, die mit dem alten Alphabeteditor abgespeichert wurden, ist weiterhin möglich.

A.3 Grammatikeditor

Der Grammatikeditor wurde völlig neu implementiert. Dennoch wurde sein GUI dem alten Editor nachempfunden, sodass es auch hier nur nötig ist, die Unterschiede zu beschreiben. Die Möglichkeit auswählen zu können, welche Morphismen visualisiert werden sollen, ist nicht mehr vorhanden, ansonsten wurden keine Fähigkeiten entfernt, sondern alte erweitert und neue hinzugefügt.

Da es nun drei verschiedene Grammatikarten gibt und für eine Grammatik zudem festgelegt werden muss, ob sie die Dangling-Condition verwendet oder nicht, wurden folgende zwei Punkte erweitert:

Datei:Neu ... **:** Während sich früher sofort ein Dateidialog zum Öffnen eines Alphabets auftat, wird nun zuerst ein Dialog angezeigt (Abb. A.1), in dem ein Name für die Grammatik eingegeben, der Typ dieser festgelegt und sich für oder gegen die Verwendung der Dangling-Condition entschieden werden kann.

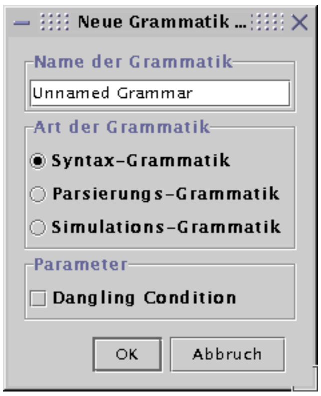

Abbildung A.1: Grammatikeditor: *Datei:Neu*

Datei:Export ... **:** Auch hier öffnet sich jetzt ein Dialog (Abb. A.2), in dem der Name der exportierten Grammatik und ihr Typ gewählt werden können. Zudem müssen natürlich die Regeln ausgewählt werden, die in der Grammatik enthalten sein sollen. Dies geschieht durch Selektion der entsprechenden Regeln in der Liste. Die Option, minimale Layoutdaten zu speichern, bewirkt, dass nur die Position der Symbole gespeichert wird. Informationen wie die Größe der Symbole oder die Anordnung ihrer grafischen Primitiva (so nicht durch Constraints erzwungen), geht verloren. Dies sorgt für wesentlich kleinere Grammatikdateien und ist zum Beispiel für eine Parsierungsgrammatik völlig ausreichend.

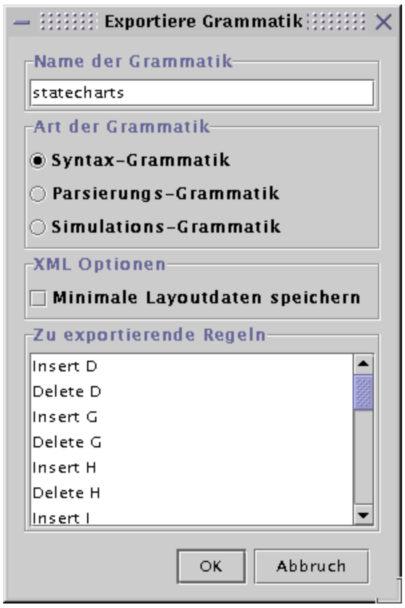

Abbildung A.2: Grammatikeditor: *Datei:Export*

Verbessert wurde der Undo-Mechanismus. Es kann nun mehr als nur der letzte Schritt rückgängig gemacht werden; ebenfalls können rückgängig gemachte Schritte erneut ausgeführt werden. Die Regelwerkzeugleiste wurde um einen Button erweitert, der einen Dialog zum Editieren der Attributbedingungen der bearbeiteten Regel (Abb. A.3) öffnet. Folgende Menüpunkte sind hinzugekommen:

Datei:An Alphabet anpassen ... **:** Hat sich während der Arbeit mit der Grammatik das ihr Alphabet geändert, so wird versucht, die Grammatik an das neue Alphabet anzupassen, genauer, es werden Alphabetregeln für neue Symbole und Datentypen hinzugefügt. Gelingt die Anpassung nicht, wird eine Fehlermeldung angezeigt.

***Bearbeiten:Duplizieren*:** Dupliziert die ausgewählte Regel oder NAC.

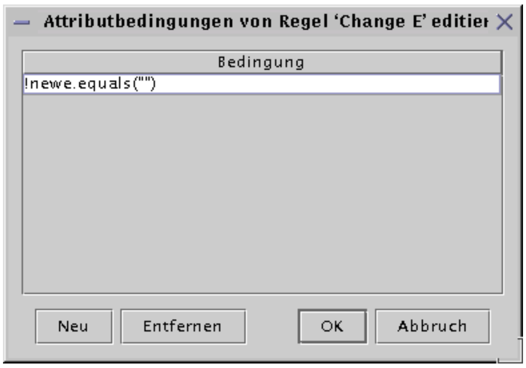

Abbildung A.3: Grammatikeditor: Regelattributbedingungen

Wie schon beim Alphabeteditor ist auch im Grammatikeditor das Speichern nur noch im XML-Format möglich, die alten Grammatikdateien können allerdings nicht mehr geladen werden.

A.4 VL-Spezifikationseditor

Mit dem VL-Spezifikationseditor lassen sich VL-Spezifikationen erstellen und editieren. Die Struktur der VL-Spezifikation wird in der Strukturansicht angezeigt. Gewöhnlich sind dort die vier Punkte „Syntax", „Parsierung", „Simulation" und „Allgemeines" zu sehen, nur „Simulation" kann Unterpunkte, nämlich die Simulationsspezifikationen, haben. Wird einer der vier Punkte angewählt, so werden auf der Arbeitsfläche die grafischen Komponenten präsentiert, um den entsprechenden Aspekt der VL-Spezifikation zu editieren.

Jedem dieser Aspekte ist ein eigener Unterabschnitt gewidmet. Hinzu kommt ein weiterer, der die Menüleiste beschreibt. Auf die Buttons in der Werkzeugleiste wird nicht eingegangen, denn ihre Funktionalität ist auch über die Menüleiste verfügbar.

A.4.1 Menüleiste

Die Menüleiste besitzt die Menüs *Datei* und *Bearbeiten* mit jeweils nur wenigen Menüpunkten:

***Datei:Neu*:** Verwirft die aktuelle VL-Spezifikation und erstellt eine leere neue.

***Datei:VL-Spezifikation öffnen* ... :** Lädt eine VL-Spezifikation aus einer Datei.

***Datei:VL-Spezifikation speichern*:** Speichert die aktuelle VL-Spezifikation in die Datei, in die sie zuletzt gespeichert oder aus der sie geladen wurde. Ist die VL-Spezifikation neu, so wird ein Dateidialog zur Auswahl eines Dateinamens präsentiert.

Datei:VL-Spezifikation speichern unter … **:** Öffnet einen Dateidialog zur Auswahl eines Dateinamens und speichert die aktuelle VL-Spezifikation in diese Datei.

***Datei:Programm beenden*:** Beendet GenGEd. Zuvor wird jedoch überprüft ob in einem der Editoren noch nicht gespeicherte Daten vorhanden sind.

***Bearbeiten:Rückgängig*:** Macht den letzten Editierschritt des Benutzers rückgängig.

***Bearbeiten:Wiederherstellen*:** Führt den zuletzt rückgängig gemachten Editierschritt erneut aus.

***Bearbeiten:Neue Simulationsspezifikation*:** Fügt eine neue leere Simulationsspezifikation zur VL-Spezifikation hinzu.

***Bearbeiten:Löschen*:** Entfernt die selektierte Simulationsspezifikation aus der VL-Spezifikation.

A.4.2 Syntax

Ist in der Strukturansicht der Punkt „Syntax“ gewählt, so wird auf der Arbeitsfläche das GUI zum Editieren des Syntaxaspekts (Abb. A.4) angezeigt. Derzeit kann nur das Alphabet und die Syntaxgrammatik für die VL-Spezifikation festgelegt werden. Ein Alphabet sollte die VL-Spezifikation immer besitzen, denn ohne dieses kann sie weder dazu genutzt werden, Diagramme zu erstellen, noch diese zu parsieren oder zu simulieren. Ist ein Alphabet, jedoch keine Syntaxgrammatik gegeben, so wird später in der VL-Umgebung ein Freihandeditor zur Verfügung gestellt, mit Syntaxgrammatik ein syntaxgesteuerter Editor.

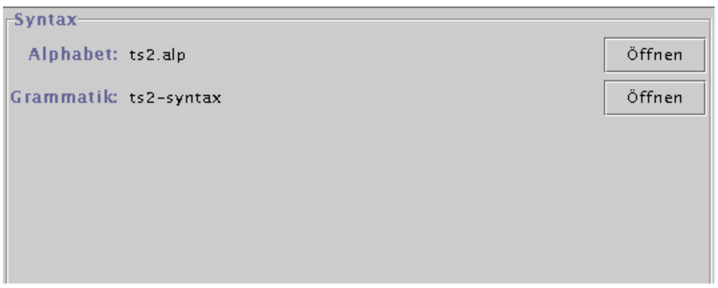

Abbildung A.4: VL-Spezifikationseditor: Syntax

Ist kein Alphabet gesetzt und möchte man ohnehin eine Syntaxgrammatik angeben, so kann man letzteres auch sofort tun, für das Alphabet wird das Alphabet der

Grammatik verwendet. Ist schon ein Alphabet gesetzt, muss die Syntaxgrammatik (und alle weiteren Grammatiken) mit diesem kompatibel sein. Verändert man das Alphabet, so werden die bisherige Syntaxgrammatik sowie die Parsierungs- und Simulationsspezifikationen verworfen.

A.4.3 Parsierung

Ist in der Strukturansicht der Punkt „Parsierung“ gewählt, so wird auf der Arbeitsfläche das GUI zum Editieren des Parsierungsaspekts (Abb. A.5) angezeigt. Es kann zunächst entschieden werden, ob überhaupt eine Parsierungsspezifikation verwendet werden soll. Entscheidet man sich dafür, so kann eine Parsierungsgrammatik festgelegt werden – diese muss zum zuvor angegebenen Alphabet der VL-Spezifikation kompatibel sein – und es bleibt zu überlegen, ob eine Layerfunktion verwendet und ob kritische Paare berechnet werden sollen.

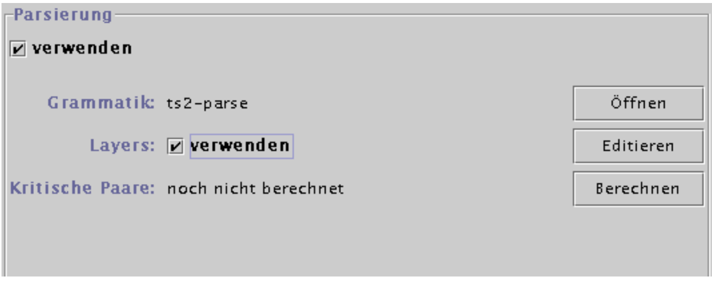

Abbildung A.5: VL-Spezifikationseditor: Parsierung

Die Layerfunktion wird in einem Dialog (A.6) editiert. Es ist für jede Regel ein Rule-Layer und für jede Sorte ein Creation- und ein Deletion-Layer zu definieren. Der Status der Layerfunktion wird angezeigt und immer aktualisiert, wenn ein Eintrag geändert wurde (nach Betätigung der Return-Taste oder Verlassen des Felds). Ist er „valid“, so handelt es sich um eine gültige Layerfunktion, bei „invalid“ wird sie später nicht verwendet. Die Gültigkeit einer Layerfunktion sagt selbstverständlich nichts darüber aus, ob sie auch sinnvoll ist.

Die Analyse der kritischen Paare wird durch das Betätigen des „Berechnen“-Buttons initiiert. Ein Fenster (Abb. A.7) mit einer Fortschrittsanzeige öffnet sich. Da die Berechnung sehr lange dauern kann, ist ein „Abbruch“-Button vorgesehen. Wurden die kritischen Paare einmal berechnet, sollte tunlichst vermieden werden, die Layerfunktion zu verändern, da dann die kritischen Paare wieder verworfen werden. Es ist also zu empfehlen, vor dem Start der Analyse noch einmal einen Blick auf die Grammatik und die Layerfunktion zu werfen.

Sowohl die Verwendung der kritischen Paare, als auch die einer Layerfunktion sind optional. Sie entscheidet nur darüber, welcher Parsierungsalgorithmus später verwendet wird. Ist keines von beiden angegeben, so wird auf einen einfachen

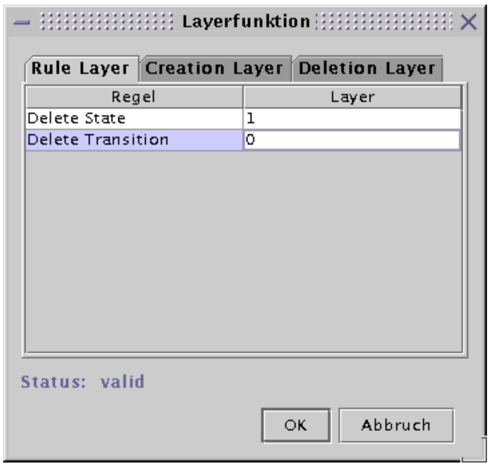

Abbildung A.6: VL-Spezifikationseditor: Definition der Layerfunktion

Abbildung A.7: VL-Spezifikationseditor: Berechnung der kritischen Paare

Backtracking-Algorithmus zurückgegriffen, mit kritischen Paaren auf eine optimierte Variante. Die Layerfunktion kann in beiden Fällen dazu benutzt werden, die Möglichkeiten, in welcher Reihenfolge die Regelanwendungen erlaubt sind, einzuschränken, was den Suchraum verkleinert, letztlich also die Parsierung beschleunigt.

A.4.4 Simulation

Ist in der Strukturansicht eine Simulationsspezifikation (nicht der Punkt „Simulation"!) gewählt, so wird auf der Arbeitsfläche das GUI zum Editieren dieser (Abb. A.8) angezeigt.

Ist ein Name und die Simulationsgrammatik, welche mit dem Alphabet der VL-Spezifikation kompatibel sein muss, festgelegt, kann mit dem Definieren der Simulationsschritte begonnen werden. Auf der Arbeitsfläche selbst können mit Hilfe der Buttons „Neu" und „Entfernen" neue leere Simulationsschritte hinzugefügt oder vorhandene entfernt werden. Die Namen der Schritte sind direkt in der Liste zu ändern. Die eigentliche Definition eines Simulationsschritts erlaubt ein Dialog, der sich öffnet, wenn der „Bearbeiten"-Button gedrückt wurde.

Wie in Abbildung A.9 zu sehen ist der Dialog dreigeteilt: Links wird eine Struk-

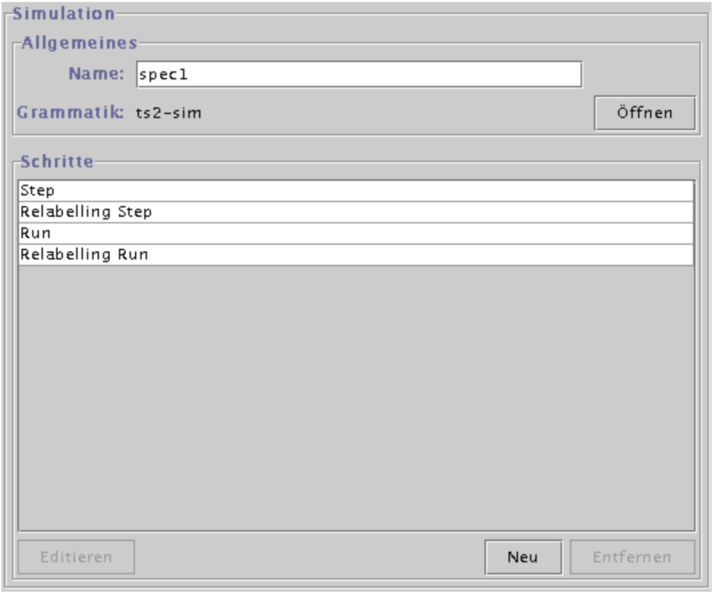

Abbildung A.8: VL-Spezifikationseditor: Simulation

turansicht angezeigt, in der Mitte eine Spalte von Buttons und rechts eine Arbeitsfläche. Die Strukturansicht stellt ein Liste der Parameter und den Ausdrucksbaum des Simulationsschritts dar und die Buttonleiste bietet alle Funktionen, um diese Struktur zu verändern. Ist in der Strukturansicht ein Parameter oder ein Ausdruck selektiert, so erscheinen auf der Arbeitsfläche die zur Manipulation des gewählten Objekts nötigen grafischen Elemente.

Die Buttonleiste ist in drei Gruppen von Buttons unterteilt. Mit den ersten lassen sich Objekte in die Struktur einfügen, die zweite dient zum Umsortieren und die dritte besteht nur aus einem Button zum Löschen von Objekt aus der Struktur.

„Parameter“: Fügt einen Parameter an das Ende der Parameterliste ein.

„AND“: Fügt einen And-Ausdruck an die selektierte Stelle in die Struktur ein.

„OR“: Fügt einen Or-Ausdruck an die selektierte Stelle in die Struktur ein.

„IF“: Fügt einen If-Ausdruck an die selektierte Stelle in die Struktur ein.

„WHILE“: Fügt einen While-Ausdruck an die selektierte Stelle in die Struktur ein.

„Rule“: Fügt einen Rule-Ausdruck an die selektierte Stelle in die Struktur ein.

„Hoch“: Ist ein Ausdruck aus einer Liste von Ausdrücken selektiert, so wird er in der Liste ein Position höher geschoben.

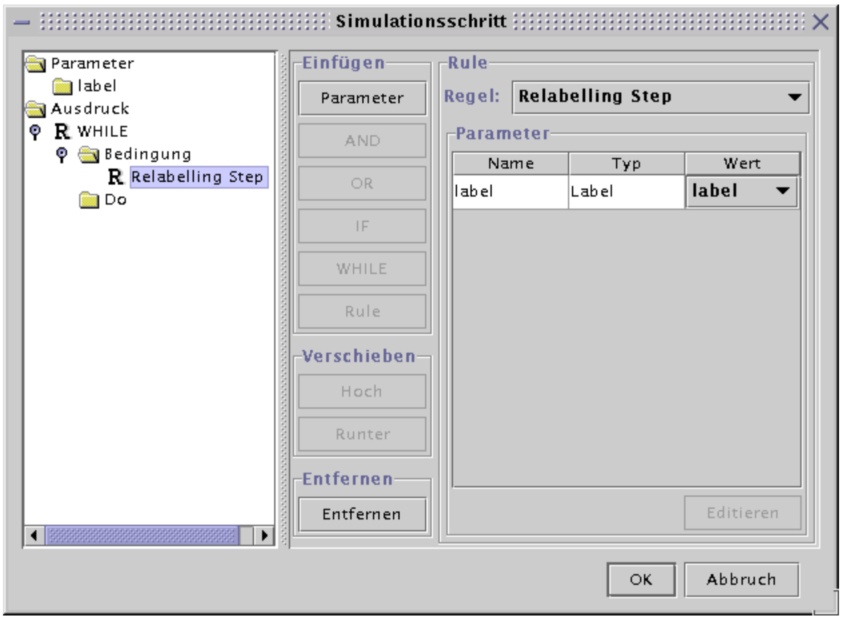

Abbildung A.9: VL-Spezifikationseditor: Definition eines Simulationsschritts

„Runter“: Ist ein Ausdruck aus einer Liste von Ausdrücken selektiert, so wird er in der Liste ein Position tiefer geschoben.

„Entfernen“: Entfernt den selektieren Parameter oder Ausdruck.

Ist ein Parameter selektiert, so werden auf der Arbeitsfläche ein Textfeld, in dem sich der Parametername ändern lässt, und ein Element zur Auswahl des Parametertyps angezeigt. Für Und- bzw. Oder-Ausdrücke lässt sich nur einstellen, ob sie konditional oder unkonditional ausgewertet werden sollen, If- und While-Ausdrücke haben überhaupt keine veränderbaren Parameter.

Ist ein Rule-Ausdruck selektiert, präsentiert sich die Arbeitsfläche wie in Abbildung A.9. Es gilt als Erstes, die Regel für den Ausdruck auszuwählen. Anschließend müssen allen Parametern der Regel entweder ein Datentypwert oder ein Parameter des Simulationsschritts zugeordnet werden. Ein Datentypwert kann nach Betätigen des „Editieren“-Buttons in einem separaten Dialog verändert werden.

A.4.5 Allgemeines

Ist in der Strukturansicht der Punkt „Allgemeines“ gewählt, so wird auf der Arbeitsfläche das GUI zum Editieren der allgemeinen Einstellungen für die VL-Spezifikation (Abb. A.5) angezeigt. Um genau zu sein, sind derzeit nur Optionen vorgesehen, die das Speichern des Alphabets und der Grammatiken in die Datei der VL-Spezifikation betreffen. Für die Grammatiken sind folgende vier Einstellungen möglich, davon die ersten beiden auch für das Alphabet:

„nur den Dateinamen“: Nur der Dateiname des Alphabets bzw. der Grammatik wird abgespeichert.

„vollständig“: Das Alphabet bzw. die Grammatik wird vollständig mit in die Datei der VL-Spezifikation abgespeichert.

„mit minimalem Layout“: Die Grammatik wird zwar mit in die Datei der VL-Spezifikation abgespeichert, jedoch mit minimalen Layoutdaten (siehe dazu Abschnitt A.3, *Datei:Export ...*).

„ohne Layout“: Die Grammatik wird zwar mit in die Datei der VL-Spezifikation abgespeichert, jedoch gänzlich ohne Layoutdaten. Für die Parsierung etwa werden diese nicht benötigt.

Der Autor empfiehlt, dass, während noch an der VL-Spezifikation gearbeitet wird, nur die Dateinamen gespeichert werden. Erst wenn die Endversion feststeht und sie den Endanwendern zur Verfügung gestellt werden soll, bietet es sich an, das Alphabet und die Grammatiken selbst mit aufzunehmen, um alle Informationen in einer Datei bereitstellen zu können. Für das Alphabet sollte dann „vollständig“ eingestellt werden, für die Grammatiken genügt „ohne Layout“.

A.5 VL-Umgebung

Die VL-Umgebung kann als eine Komponente neben den Editoren oder auch „stand alone“ betrieben werden (siehe Abschnitt A.1.3). In jedem Fall ist ihre Funktionalität exakt die selbe. Für die eigentliche Arbeit muss sie erst durch eine VL-Spezifikation konfiguriert werden. Enthält diese eine Syntaxgrammatik, so wird ein syntaxgesteuerter Editor bereitgestellt, ist nur ein Alphabet enthalten, ein Freihandeditor. Unabhängig davon sind Parsierung und Simulation. Besitzt die VL-Spezifikation eine Parsierungsspezifikation, kann das jeweils aktuelle Diagramm jederzeit parsiert werden. Ist mindestens eine Simulationsspezifikation vorhanden, so kann zwischen Editor und Simulationskomponente hin- und hergeschaltet werden.

A.5.1 Menüleiste

Die Menüleisten der Editier- und der Simulationskomponente unterscheiden sich. Hier soll nur auf den gemeinsamen Anteil eingegangen werden:

***Datei:VL-Spezifikation öffnen ...* :** Lädt eine VL-Spezifikation aus einer Datei.

***Datei:Diagramm öffnen* ... :** Lädt ein Diagramm aus einer Datei. Das Diagramm muss mit dem Alphabet der aktuellen VL-Spezifikation kompatibel sein.

***Datei:Diagramm speichern*:** Speichert das aktuelle Diagramm in die Datei, in die es zuletzt gespeichert oder aus der es geladen wurde. Ist das Diagramm neu, so wird ein Dateidialog zur Auswahl eines Dateinamens präsentiert.

***Datei:Diagramm speichern unter* ... :** Öffnet einen Dateidialog zur Auswahl eines Dateinamens und speichert das aktuelle Diagramm in diese Datei.

***Datei:Programm beenden*:** Beendet GenGEd. Zuvor wird jedoch überprüft ob in einem der Editoren noch nicht gespeicherte Daten vorhanden sind.

***Diagramm:Neues Diagramm*:** Erzeugt ein neues Diagramm. Im Freihandmodus ist dieses leer, beim syntaxgesteuerten Editieren wird eine Kopie des Startdiagramms der Syntaxgrammatik erzeugt.

***Diagramm:Diagramm duplizieren*:** Dupliziert das aktuelle Diagramm.

***Diagramm:Diagramm entfernen*:** Entfernt das aktuelle Diagramm.

***Diagramm:Diagrammnamen editieren*:** Öffnet einen Dialog zum Editieren des Namens des aktuellen Diagramms.

***Bearbeiten:Rückgängig*:** Macht den letzten Editierschritt des Benutzers rückgängig.

***Bearbeiten:Wiederherstellen*:** Führt den zuletzt rückgängig gemachten Editierschritt erneut aus.

Unter den aufgelisteten Punkten des *Diagramm*-Menüs werden die Namen aller geladenen und neu erstellten Diagramme angezeigt. Der Name des aktuellen Diagramms ist gekennzeichet. Zu einem anderen Diagramm kann durch Auswahl des entsprechenden Menüeintrags umgeschaltet werden.

A.5.2 Editor

Als Freihandeditor wird der Diagrammeditor verwendet, als syntaxgesteuerter Editor der VL-Editor. Ersterer stammt aus dem Alphabeteditor und ist nahezu unverändert übernommen worden. Jene Aufgaben, die die VL-Umgebung als Ganzes wahrnimmt, wie das Erzeugen, Laden und Speichern von Diagramm, tauchen in den Editormenüs natürlich kein zweites Mal auf.

Ähnliches gilt auch für den syntaxgesteuerten Editor. Der VL-Editor ist zwar neu implementiert worden, doch das GUI ist fast genauso organisiert wie bei der ursprünglichen Implementation.

Bei beiden Editoren wurden die Werkzeugleiste um zwei Buttons erweitert. Der erste parsiert das aktuelle Diagramm und meldet, ob es korrekt oder inkorrekt ist, der zweite schaltet in den Simulationsmodus um.

A.5.3 Simulation

Die Simulationskomponente ist derzeit noch recht spartanisch ausgestattet (Abb. A.10). Die Strukturansicht listet alle Simulationsspezifikationen samt ihrer Simulationschritte auf, die Arbeitsfläche zeigt das simulierte Diagramm.

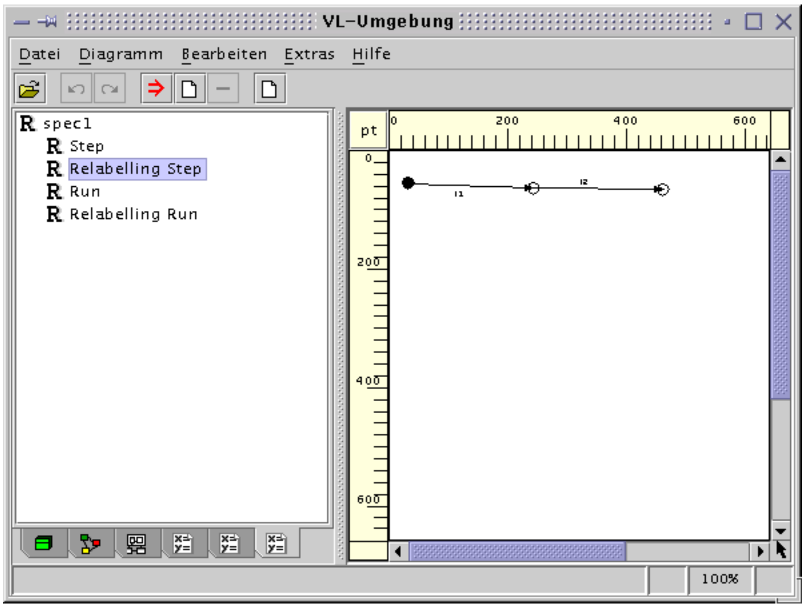

Abbildung A.10: VL-Umgebung: Simulation

Wichtig zu verstehen ist, dass die Simulation mit einer Kopie des aktuellen Diagramms durchgeführt, das Original also nicht verändert wird. Des Weiteren arbeitet der Undo-Mechanismus in der Simulationskomponente nur lokal, sprich, es werden nur die Benutzeraktionen aufgezeichnet, die dort ausgeführt worden. Jedes Mal, wenn vom Editiermodus zur Simulation umgeschaltet wird, befindet sich die Komponente wieder im initialen Zustand, das heißt eine Kopie des aktuellen Diagramms wird angezeigt und der Undo-Mechanismus ist zurückgesetzt.

Im Wesentlichen lassen sich drei verschiedene Aktionen auslösen: Das Anstoßen eines Simulationsschritts, das Abbrechen eines Simulationsschritts und das Zurücksetzen der Simulation auf den Anfangszustand. Dies sind die Menüpunkte:

Bearbeiten:Rückgängig: Macht den letzten Editierschritt des Benutzers rückgängig.

***Bearbeiten:Wiederherstellen*:** Führt den zuletzt rückgängig gemachten Editierschritt erneut aus.

***Bearbeiten:Simulationsschritt*:** Der in der Strukturansicht ausgewählte Simulationsschritt wird auf das simulierte Diagramm angewendet.

***Bearbeiten:Simulation zurücksetzen*:** Die Simulation wird zurückgesetzt, das heißt, es wird wieder eine Kopie des aktuellen Diagramms angezeigt.

***Bearbeiten:Simulation stoppen*:** Ist gerade ein Simulationsschritt in der Ausführung begriffen, so wird diese unverzüglich abgebrochen.

In der Werkzeugleiste sind Buttons, die genau die gleichen Aktionen auslösen, und ein weiterer, der wieder in den Editiermodus zurückschaltet.