Konzeption und Implementierung einer Kontrollkomponente für attributierte Graphersetzung

Diplomarbeit

von

Murat Kurt

Matr.-Nr: 147536

März 2000

VORGELEGT AN DER
TECHNISCHEN UNIVERSITÄT BERLIN
FACHGEBIET THEORETISCHE INFORMATIK
FACHBEREICH INFORMATIK

BETREUUNG DURCH
PROF. DR. HARTMUT EHRIG
DR. GABRIELE TAENTZER
DR. STEFAN GRUNER

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Graphen sind eine der wichtigsten Datenstrukturen in der Informatik, mit denen sich Informationen darstellen lassen. Gerade durch die Weiterentwicklung grafischer Sprachen zur objektorientierten Modellierung von Softwaresystemen, wie zum Beispiel UML [UML99] (Unified Modeling Language), wird deutlich, dass Systeme, die nicht nur die statischen Graphenstrukturen modellieren, sondern auch dynamische Aspekte einbinden, an Bedeutung gewinnen.

Graphen sind bei der Darstellung komplexer Strukturen sehr verbreitet. Schon die intuitive Verwendung von Skizzen bei der Problemdarstellung zeigt, dass Graphen die bevorzugt gewählten Datenstrukturen sind. Insbesondere bei der Darstellung von komplexeren Informationen werden Graphen zur Veranschaulichung herangezogen. In der UML gibt es mehrere Diagrammtypen (wieder Graphen), die das statische und dynamische Verhalten von Systemen beschreiben. Sobald aber in die Berechnung von grafischen Datenstrukturen übergegangen wird, findet ein Bruch von grafischer zu textlicher Darstellung statt. Bei der UML zum Beispiel werden statische und dynamische Diagramme letztlich auf textbasierte Repräsentationen überführt (in Java [http://www.javasoft.com] oder C++ beispielsweise).

Graphgrammatiken hingegen erlauben es, durch ein Set von Produktionsregeln, die auf Graphen anwendbar sind, diese zu manipulieren. Systeme, die auf einer graphgrammatischen Grundlage basieren, wie zum Beispiel AGG (The **A**ttributed **G**raph **G**rammar System) ([http://tfs.cs.tu-berlin.de/agg/], Grrr (A **G**raph **R**ewriting Programming Language for G**R**aph D**R**awing) [Grrr] oder PROGRES (**PRO**grammierte **GR**aph **E**rsetzungs**S**ysteme) [Rodgers], können auf die seit etwa 25 Jahren gewonnen theoretischen Erkenntnisse aus diesem Gebiet zurückgreifen.

Kontrollstrukturen bieten die Möglichkeit Graphersetzungen zu steuern, um so globale Bedingungen, die für Graphen gelten sollen, zu sichern.

AGG wird an der Technischen Universität seit etwa 1991 entwickelt. Mit diesem System können Graphmodelle erzeugt werden, die dann durch Graphersetzung verändert werden [Michael97]. Die am Graphmodell beteiligten Graphobjekte können durch eine Attributierungskomponente [MEL97] attributiert werden. Während in Grrr und in PROGRES ausgeprägte Kontrollstrukturen für die Steuerung von Graphersetzungen zur Verfügung stehen, fehlt in AGG eine Entsprechung. Daher soll in dieser Diplomarbeit eine Kontrollkomponente für das Graphersetzungssystem AGG entworfen und implementiert werden.

Etappen in dieser Diplomarbeit

Zur Verwirklichung einer Kontrollkomponente in AGG sollen folgende Etappen helfen, geeignete Kontrollstrukturen zu finden.

1. *Etappe*: Einbettung des graphgrammatischen Kontextes in die Kontrollkomponente.
2. *Etappe*: Suche nach geeigneten Kontrollstrukturen, um Steuerungen in AGG zu realisieren.
3. *Etappe*: Suche nach einer geeigneten grafischen Beschreibungssprache für die Steuerung der Graphersetzung.

4. *Etappe*: Berücksichtigung der in der 1. bis 3. Etappe gewonnenen Erkenntnisse beim Entwurf einer Kontrollkomponente für AGG.

Ziel dieser Diplomarbeit

Diese Diplomarbeit setzt sich zum Ziel einen Steuerungseditor zu entwickeln, mit dem man (aus Knoten und Kanten bestehende) Steuerungsgraphen, die an eine grafische Notationssprache angelehnt sind, entwerfen kann, um so die Graphersetzung zu steuern. Der Steuerungseditor soll als Komponente für das Graphersetzungssystem AGG implementiert werden.

Aufbau dieser Diplomarbeit

Alle beschriebenen Etappen spiegeln sich im Aufbau der Diplomarbeit wieder:

1. *Kapitel*: Einleitung der Diplomarbeit.
2. *Kapitel*: Hier werden Grundlagen vermittelt, um den Entwurf des Steuerungseditors zu fundieren. Anfänglich wird beschrieben, was unter Graphersetzung zu verstehen ist und wie sie in AGG realisiert wird. Danach werden Kontrollstrukturen aus dem Gebiet der Programmschemata [Albayrak98] auf Verwendbarkeit hinsichtlich der Steuerung von Graphersetzungen analysiert. Der letzte Abschnitt in diesem Kapitel sucht nach Kontrollstrukturen in Aktivitätsdiagrammen aus der UML, die einerseits differenzierte Kontrollstrukturen aufweisen und andererseits auf einer grafischen Notationssprache basieren.
3. *Kapitel*: Hier findet der Entwurf des Steuerungseditors statt, indem die im 2. Kapitel gewonnenen Erkenntnisse berücksichtigt werden. Es wird entschieden, Kontrollstrukturen, die in Gestalt von Steuerungsgraphen durch den Steuerungseditor spezifizierbar sind, in Anlehnung an das Aktivitätsdiagramm zu entwerfen. Begründet ist diese Entscheidung durch die Standardisierung der grafischen Notationssprache UML, seitens der OMG (Object Management Group) im November 1997; Denn damit lehnt sich die Notation des Steuerungsgraphen an eine weit verbreitete und anerkannte Sprache an.
4. *Kapitel*: In diesem Kapitel werden zwei Ansätze des Automaten dargestellt, der interpretative und der kompilative Ansatz. Von diesen realisiert die Kontrollkomponente – für das Graphersetzungssystem AGG – lediglich den interpretativen Ansatz. Der kompilative Ansatz wird skizziert.
5. *Kapitel*: Zusammenfassung und Ausblick der Diplomarbeit.

2 Grundlagen

Dieses Kapitel beschreibt was unter Graphgrammatiken zu verstehen ist und wie sie in AGG [http://tfs.cs.tu-berlin.de/agg/] realisiert wurden. AGG ist im Institut für Kommunikations- und Softwaretechnik im Fachgebiet theoretische Informatik – Formale Spezifikation (TFS) entwickelt worden. Im Hinblick auf den Entwurf von Kontrollstrukturen für dieses Graphersetzungssystem, das in der Programmiersprache Java [http://www.javasoft.com] entwickelt wurde, sollen Programmschemata und Aktivitätsdiagramme diesbezüglich analysiert werden. Grundlage für die Analyse bilden die graphgrammatischen Erkenntnisse aus folgendem Unterkapitel.

2.1 Algebraische Graphgrammatiken und AGG

In diesem Kapitel findet eine informelle Einführung in algebraische Graphgrammatiken statt. Des weiteren wird das Graphersetzungssystem AGG vorgestellt, das die Spezifizierung von Graphgrammatiken am Computer ermöglicht, sowie die Transformation [Michael97] von Graphen, durch spezifizierte Regeln der Graphgrammatik, realisiert. Daneben besitzt AGG noch eine Attributierungskomponente [Mel97], welche die Vergabe von Attributen an Graphobjekte (Knoten und Kanten, aus denen Graphen zusammengesetzt werden können) ermöglicht. Die Einführung in Graphgrammatiken beginnt mit der informellen Einführung in die Datenstruktur Graph.

2.1.1 Algebraische Graphgrammatiken

Algebraische Graphgrammatiken kommen aus dem Gebiet der formalen Sprachen. Jede Grammatik beschreibt eine sogenannte Sprache. Die Wörter der Graphgrammatiken sind Graphen. Hier eine informelle Beschreibung der Graphen:

- Graphen bestehen aus Knoten und Kanten.
- Der leere Graph, also der Graph ohne Knoten und Kanten, gehört auch zu den Graphen und ist Teilgraph jedes Graphen.
- Eine Kante besitzt immer einen Quell- und Zielknoten.
- Kanten sind immer in Richtung des Zielknotens gerichtet und werden durch einen Pfeil repräsentiert.
- Knoten werden grafisch durch einen Kreis repräsentiert.
- Ziel- und Quellknoten einer Kante können gleich sein, so dass Schlingen erzeugt werden können.
- Knoten können mehrere eingehende und ausgehende Kanten besitzen.

Folgende Abbildung zeigt Beispiele von Graphen:

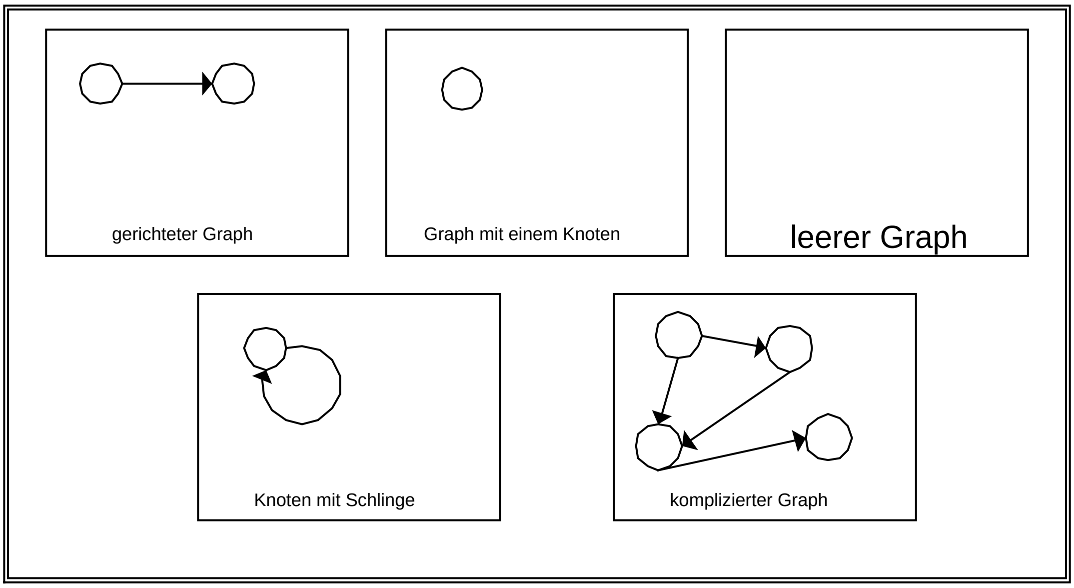


Abbildung 1 Beispiel für Graphen

Der hier vorgestellte Graphenbegriff soll erweitert werden, so dass jedem Graphobjekt (also Knoten oder Kanten) ein Typ aus seiner Typmenge (entweder Knotentypmenge oder Kantentypmenge) zugeordnet wird. Die Definition 1 beschreibt den Graphenbegriff, wie er bis jetzt vorgestellt wurde, formal:

Definition 1 (Graphen):

- Ein Graph G = (V, E, N, A, s: E $\rightarrow$ V, t: E $\rightarrow$ V, a: E $\rightarrow$ A, n: V $\rightarrow$ N) besteht aus einer Menge von Knoten V, einer Menge von Kanten E, einer Menge von Knotentypen N, einer Menge Kantentypen A, sowie vier totalen Abbildungen:
    1. s : ordnet jeder Kante einen Quellknoten zu
    2. t : ordnet jeder Kante einen Zielknoten zu
    3. n : bildet jeden Knoten auf ihren Knotentyp ab
    4. a : bildet jede Kante auf ihren Kantentyp ab

Die Definition 1 erweitert durch die Zuordnung von Typen an Graphobjekte, das sind die Objekte in der Menge *V* (Knoten) und *E* (Kanten), das Graphenverständnis. In AGG sind Graphobjekte auch an Typen gebunden. Dadurch können verschiedene Typen von Knoten oder Kanten spezifiziert werden, wie zum Beispiel ein Knoten, der ein Haus repräsentiert oder eine Kante, die Geschwisterbeziehungen repräsentiert. Siehe dazu folgende Abbildung:

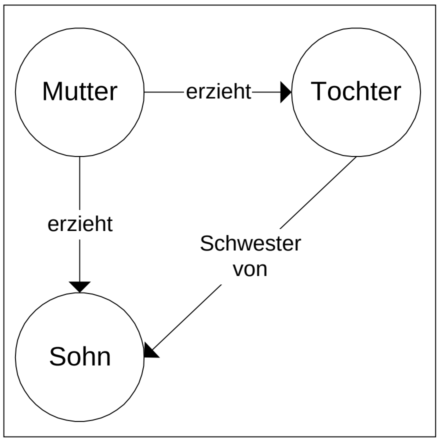


Abbildung 2 Graphobjekte mit Typen

Die Abbildung 2 verdeutlicht was unter Graphobjekttypen verstanden wird. *Mutter, Tochter* und *Sohn* sind Typen für Knotenobjekte, während *erzieht* und *'Schwester von'* Typen von Kanten sind. Kanten beschreiben Relationen zwischen den Knoten. Diese Abbildung verdeutlicht aber auch, dass keine Aussagen über Namen, Alter oder Beruf der Mutter getroffen werden. Daher ermöglicht die Attributkomponente in AGG die Vergabe von Attributen an Knoten und Kanten.

Einem Graphobjekt können, durch die Attributkomponente, mehrere Attribute vergeben werden. Attribute sind Tupel, die aus zwei Komponenten bestehen, nämlich dem „*type*" und dem „*value*". Attribute in AGG können aber auch Variable sein. Die Vergabe von Attributen ist im Gegensatz zu der Vergabe von Typen optional. Beiden, Typen und Attributen, kommen bei der Graphersetzung in AGG eine besondere Rolle zu. Hier der durch Attribute erweiterte Graph aus der Abbildung 2:

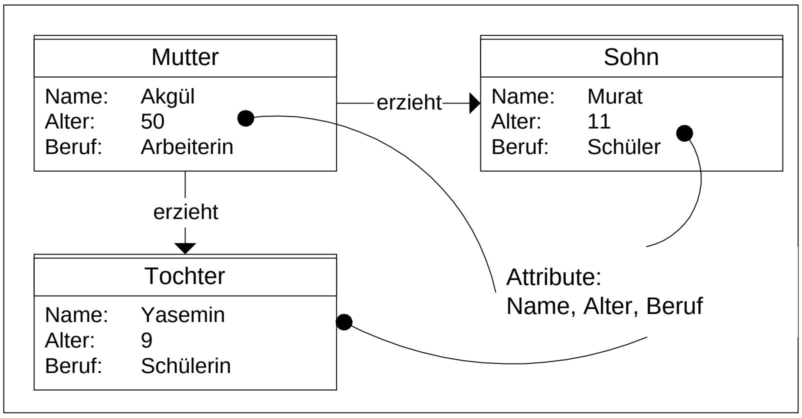


Abbildung 3 Attributerweiterung von Graphen

In AGG werden Knoten in rechteckiger Form dargestellt, weil die grafische Repräsentation der Attribute dadurch vereinfacht wird. Semantisch unterscheiden sie sich dadurch von dem bis hier eingeführten Knotenbegriff nicht.

Bis hierher wurde beschrieben, wie die Wörter der Graphgrammatiken aussehen dürfen, die in AGG spezifizierbar sind. Nun sollen jedoch die Bestandteile der Graphgrammatiken eingeführt werden. Das sind der Startgraph und die ihn durch Graphersetzung manipulierenden Regeln (oder Graphersetzungsregeln oder Produktionsregeln).

Der Startgraph ist ein initialer Graph, der als Basis für die Graphersetzung dient. Dieser besondere Graph wird durch Graphersetzung verändert. Nach der ersten Veränderung des Startgraphen durch eine Graphersetzungsregel spricht man dann nicht mehr von einem Startgraphen, sondern von einem Arbeitsgraphen. Die Veränderungsvorschrift wird in Form von Produktionsregeln spezifiziert. Der Startgraph modelliert Informationen durch eine Datenbasis. So könnte ein Startgraph, wie er in Abbildung 2 dargestellt ist, als Datenbasis für die Beschreibung von Verwandtschaftsbeziehungen aufgefasst werden. Die Manipulation solcher Arbeitsgraphen wird durch die Anwendung von Produktionsregeln auf den Startgraphen erreicht.

Jede Produktionsregel besitzt eine linke und eine rechte Regelseite (siehe Abbildung 4). Beide Regelseiten sind Graphen. Bei der Graphersetzung, also der Manipulation des Arbeitsgraphen durch die Anwendung der Produktionsregeln auf ihn, wird der Graph in der linken Regelseite als Teilgraph im Arbeitsgraphen gesucht. Wird ein Teilgraph gefunden, kann er durch die rechte Regelseite ersetzt werden. In einem Arbeitsgraphen können mehrere Teilgraphen gefunden werden, die der linken Regelseite entsprechen. Alle Teilgraphen können als Ansatz bezeichnet werden. Um spezifizieren zu können, welche Graphobjekte ersetzt, gelöscht oder erhalten bleiben sollen, müssen Morphismen zwischen der linken Regelseite und der rechten Regelseite definiert werden. Morphismen in Produktionsregeln bilden Graphobjekte der linken Regelseite auf Graphobjekte der rechten Regelseite ab. Diese Abbildung unterliegt folgenden Voraussetzungen:

1. *Voraussetzung*: Kanten können nur auf Kanten und Knoten nur auf Knoten abgebildet werden. Werden Kanten von der linken Regelseite auf die rechte Regelseite abgebildet, so müssen auch Quell- und Zielknoten dieser Kante von der linken Regelseite auf die Quell- und Zielknoten der rechten Regelseite abgebildet werden (Strukturgleichheit).
2. *Voraussetzung*: Nur Graphobjekte gleichen Typs können aufeinander abgebildet werden (Typgleichheit).
3. *Voraussetzung*: Nur Graphobjekte mit gleicher Anzahl von Attributen, die wiederum in *type* und *value* übereinstimmen, können aufeinander abgebildet werden (Attributgleichheit). Möchte man sich nicht auf eine bestimmte Instanz eines Attributes festlegen, so kann man mit Variablen arbeiten. Diese werden dann bei der Graphersetzung belegt.

Meistens ist der Morphismus zwischen der linken und der rechten Regelseite partiell. Existieren auf der linken Regelseite Graphobjekte, die nicht durch eine Morphismusabbildung gebunden sind, so werden sie bei der Graphersetzung gelöscht. Gibt es solche Graphobjekte auf der rechten Regelseite, so werden sie bei der Graphersetzung neu eingefügt. Werden zwei oder mehr Graphobjekte von der linken Regelseite auf das selbe Graphobjekt in der rechten Regelseite abgebildet, so werden sie bei der Graphersetzung „verklebt“ (siehe Abbildung 4 und Abbildung 5). Dies würde einer nicht injektiven Morphismusabbildung entsprechen.

Bei der Graphersetzung versucht man im Startgraphen einen totalen Morphismus zwischen der linken Regelseite und dem Arbeitsgraphen zu finden. Findet man eine solche Abbildung, so hat man einen Teilgraphen gefunden, der durch die rechte Regelseite ersetzt werden darf. Totaler Morphismus bedeutet, dass jedes Graphobjekt aus der linken Regelseite auf ein Graphobjekt im Arbeitsgraphen abgebildet wird. Des weiteren kann der Morphismus zwischen linker Regelseite und Arbeitsgraphen injektiv sein, dass heißt es dürfen nicht zwei oder mehr Graphobjekte auf das selbe Graphobjekt abgebildet werden.

Folgendes Beispiel zeigt eine kleine Graphgrammatik:

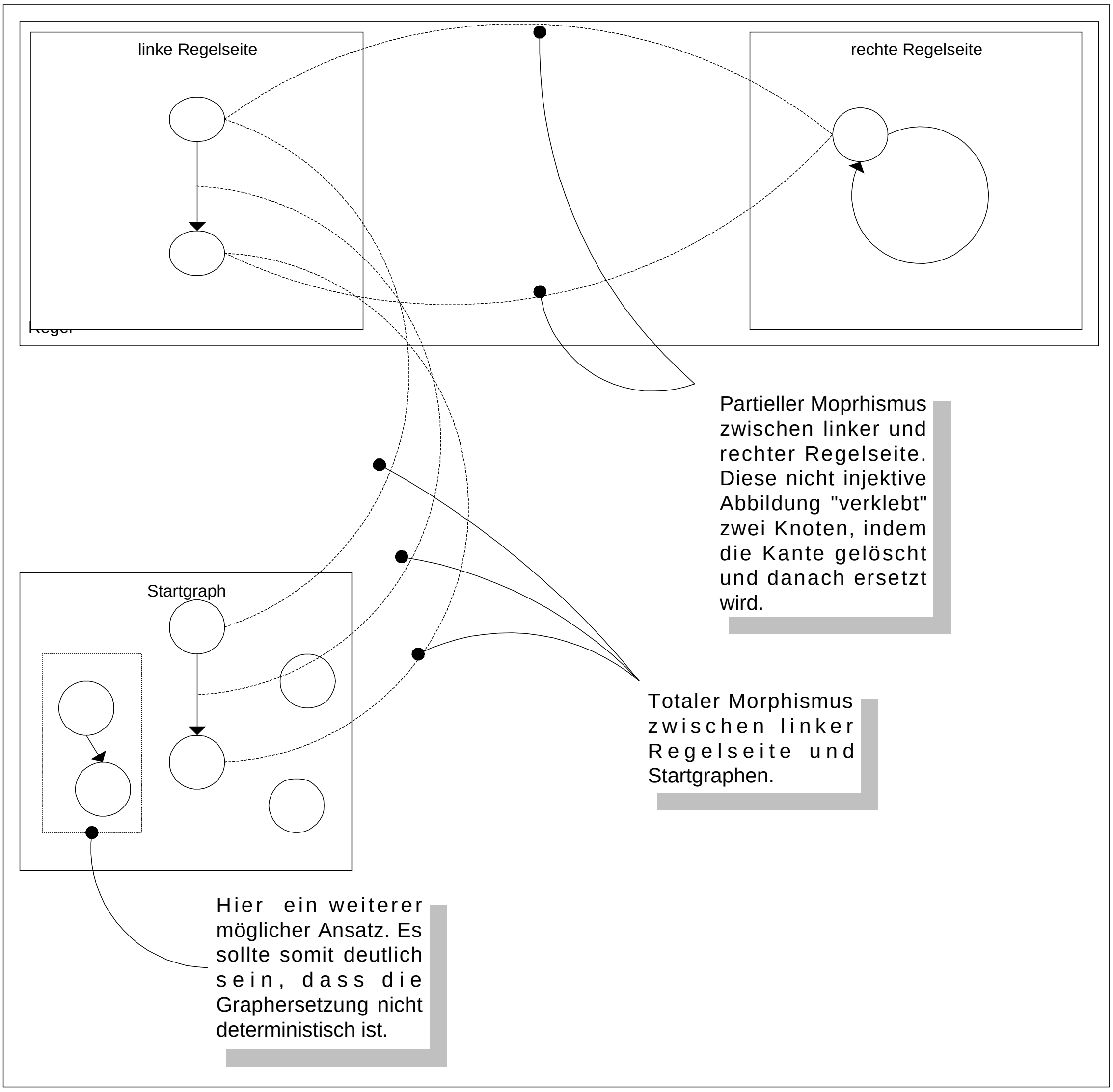


Abbildung 4 Beispiel einer Graphgrammatik

Als Ergebnis der in Abbildung 4 gezeigten Graphersetzung entsteht folgender modifizierter Arbeitsgraph:

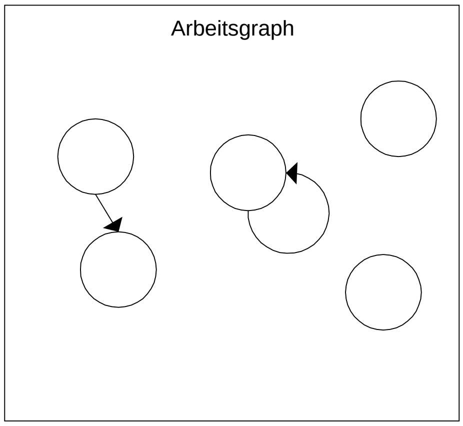


Abbildung 5 Modifizierter Startgraph aus Abbildung 4

Es können mehrere einer Graphgrammatik angehörende Produktionsregeln spezifiziert werden. Die Anwendungsreihenfolge der Regeln ist nicht vorgegeben. Sofern irgendeine Regel anwendbar ist, kann sie auch angewendet werden.

2.1.2 Das Graphersetzungssystem AGG

Das Graphersetzungssystem AGG besitzt im wesentlichen zwei Bestandteile. Erster Bestandteil ist der Editor. Er erlaubt menügesteuert eine oder mehrere Graphgrammatiken zu spezifizieren. Dazu können Knoten und Kanten in die Regeln und in den Startgraphen platziert werden. Über zwei Textfelder können Knoten- und Kantentypen definiert werden. Außerdem können Morphismen, zwischen linker und rechter Regelseite oder linker Regelseite und Startgraphen, spezifiziert werden.
In der folgenden Abbildung sind Felder mit den verschiedenen Eingabemöglichkeiten in AGG beschriftet:

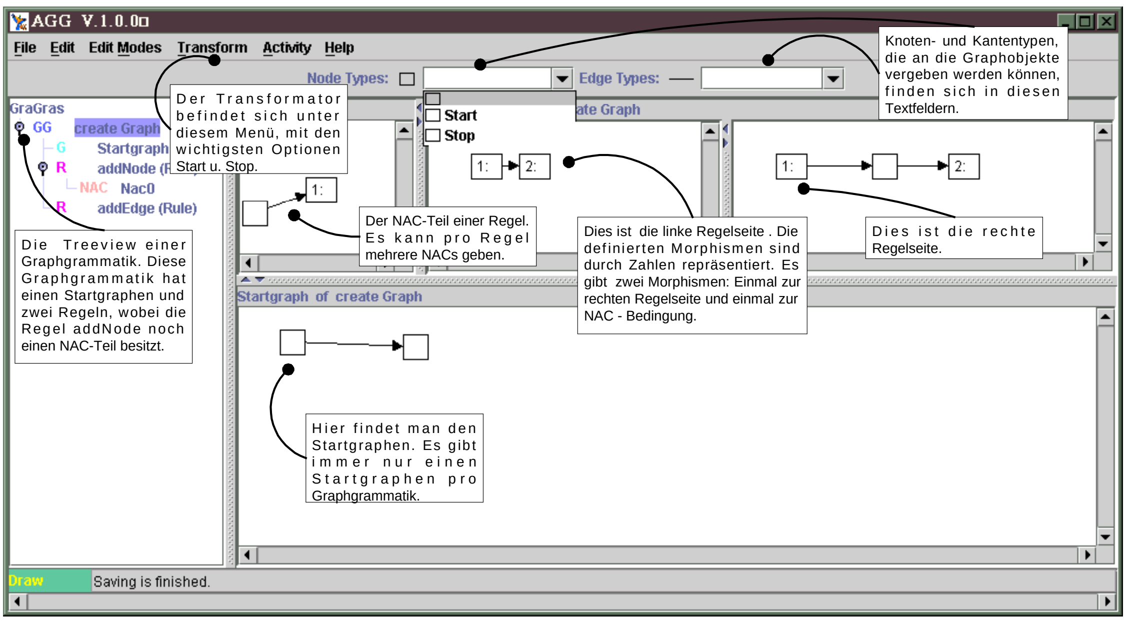


Abbildung 6 AGG - Hauptfenster

Der zweite Bestandteil ist der Transformator, also das Programmteil, das durch Anwendung der Regeln den Startgraphen verändert. Die Anwendungsreihenfolge, der zu einer Graphgrammatik gehörenden Regeln auf den Arbeitsgraphen, ist zur Zeit noch durch die Editierreihenfolge festgelegt. Es gibt lediglich die Möglichkeit die Reihenfolge der Regeln in der Treeview der Graphgrammatik zu verändern. Zuerst wendet der Transformator die erste Regel aus der Treeview so oft wie möglich an und verfährt ebenso mit den nachfolgenden Regeln. Hat der Transformator die letzte Regel erreicht beginnt er wieder von vorne, denn es könnte sein, dass eine der zuvor angewendeten Produktionsregeln wieder anwendbar ist. Die Transformation hört erst dann auf, wenn es keine Produktionsregel mehr gibt, die auf den Arbeitsgraphen anwendbar ist (siehe Abbildung 7).

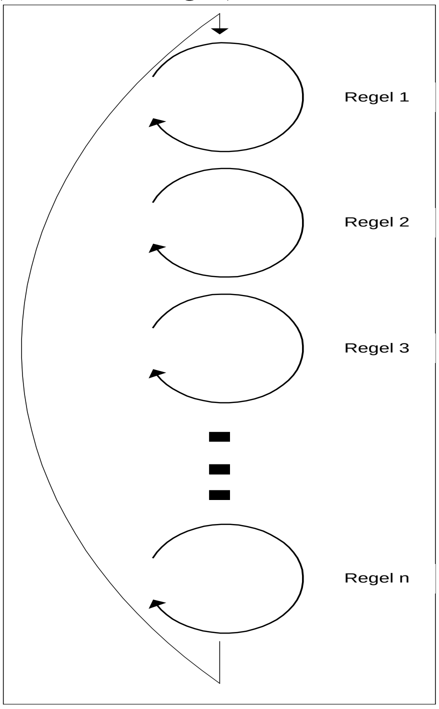


Abbildung 7 Transformator in AGG

Negative Anwendungsbedingungen

In AGG findet man eine weitere Komponente der Produktionsregel, die man über den Editor spezifizieren kann. Diese Komponente heißt negative Anwendungsbedingung (s. Abbildung 6). Zu einer Produktionsregel können mehrere sogenannte NAC (Negative Application Condition) spezifiziert werden. Negative Anwendungsbedingungen sind ebenfalls Graphen. Von der linken Regelseite wird zu jeder negativen Anwendungsbedingung ein partieller Morphismus spezifiziert. Die Semantik lautet dann: Ersetze im Startgraphen den gefundenen Ansatz durch rechte Regelseite, wenn keines der NACs erfüllt ist. Hier ein Beispiel zur besseren Verdeutlichung:

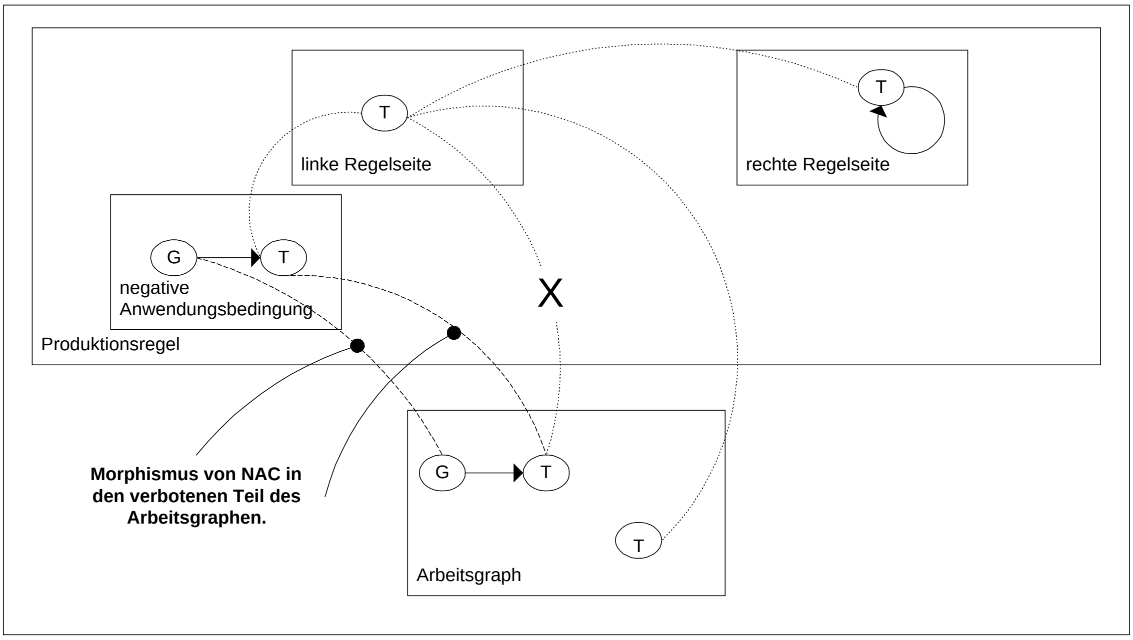


Abbildung 8 Negative Anwendungsbedingung

Die negative Anwendungsbedingung in der Abbildung 8 lässt nur den aufgezeichneten Ansatz zu, nicht aber den mit *X* markierten (siehe Abbildung 8). Negative Anwendungsbedingungen spezifizieren also, wann eine Regel nicht angewendet werden darf.

Das folgende Kapitel gibt eine Motivation dafür warum Kontrollstrukturen, die Graphersetzungen steuern, wichtig sind.

2.1.3 Motivation der Kontrollstrukturen

Graphgrammatiken bestehen aus dem Startgraphen und Produktionsregeln. Die Graphersetzung wird hauptsächlich durch den Transformator gesteuert, der dabei eine Schleife durchläuft (siehe Abbildung 7). Es ist festzustellen, dass bei jeder Graphersetzung der Arbeitsgraph verändert wird, indem Produktionsregeln auf ihn lokal angewendet werden. Für den Erhalt von globalen Bedingungen im Arbeitsgraphen ist eine Kontrollkomponente nötig. Folgendes Beispiel soll diesen Zusammenhang verdeutlichen:

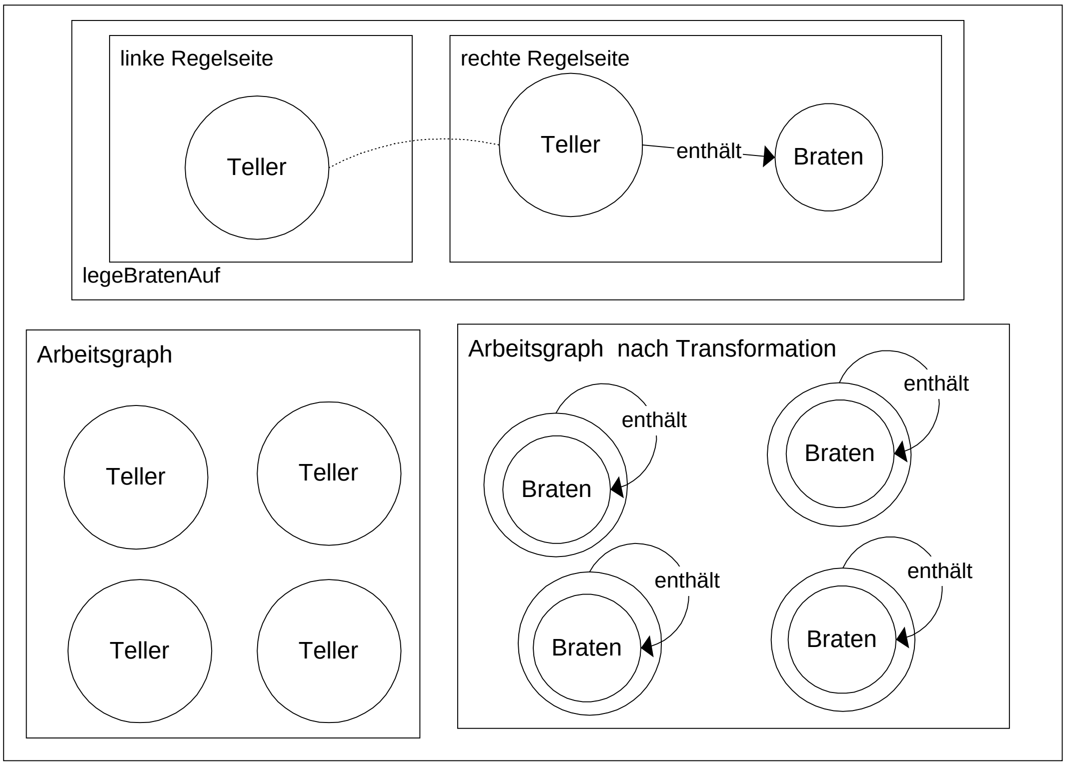


Abbildung 9 Globale Bedingungen

In Abbildung 9 sind ein Arbeitsgraph und eine Produktionsregel spezifiziert. Knoten sollen Teller (Typ *Teller*) oder Braten (Typ *Braten*) repräsentieren. Daneben gibt es einen Kantentyp (Typ *enthält*), der beschreibt, dass ein Teller einen Braten enthält. Der Arbeitsgraph besteht aus vier Knoten, die vier Teller auf einem Esstisch repräsentieren sollen:

1. *Fall:* Da die Teller leer sind haben sie noch keine Relation *enthält* zu einem Braten. Es werden Gäste erwartet, weshalb die Teller mit Braten belegt werden sollen. Daher verbindet die Produktionsregel (*legeBratenAuf*) einen leeren Teller mit einem Braten, indem eine Kante (vom Typ *enthält*) verwendet wird. Die globale Bedingung soll sein: Alle Teller enthalten Braten oder kein Teller enthält Braten. Durch die globale Bedingung wird sichergestellt, dass alle Gäste einen Braten bekommen. Falls nicht genügend Braten in der Küche sind, stellt diese Bedingung auch noch sicher, dass keiner der Gäste einen Braten bekommt, um somit alle Gäste gleichberechtigt zu behandeln. Der Transformator in AGG würde die Produktionsregel (*legeBratenAuf*) so oft anwenden wie er kann. Nach Beendigung der Transformation enthalten dann auch tatsächlich alle Teller einen Braten. Die globale Bedingung bleibt erfüllt. In diesem Fall hat der Transformator in AGG Glück gehabt.
2. *Fall*: Nun stelle man sich vor, dass die Teller nicht für Gäste bestimmt sind, sondern für die eigenen vier Kinder, von denen zwei in der Schule sehr ungezogen waren und deshalb zur Strafe, beim Mittagessen zu Hause, keinen Braten bekommen. Die globale Bedingung lautet daher: Zwei Teller enthalten keinen Braten. Die Transformation in AGG beginnt ein weiteres Mal. Wenn der Transformator mit der Transformation aufgehört hat stehen aber wieder vier Braten auf dem Tisch. AGG hat diesmal die globale Bedingung nicht erfüllen können (Glück für die zwei ungezogenen Kinder).

Hätte AGG die Produktionsregel *legeBratenAuf* nur zweimal angewendet und danach aufgehört, so wäre die globale Bedingung erfüllt gewesen (und die zwei Kinder müssten hungern).

Aus dem letzten Beispiel wird deutlich, dass die Kontrollierung von Graphersetzung für den Erhalt von globalen Bedingungen sehr wichtig ist. Kontrollstrukturen sollen helfen die Graphersetzung zu steuern. Welche Kontrollstrukturen verwendet werden können, um Graphersetzungen zu steuern, soll im folgenden Abschnitt diskutiert werden. Im Hinblick auf die grafische Darstellung dieser Kontrollstrukturen findet eine weitere Analyse statt, denn die Kontrollkomponente soll grafische Kontrollstrukturen erzeugen, um ein intuitives Programmieren von Graphersetzungen zu gewährleisten.

2.2 Programmschemata

Die Programmschemata sind etwa 20 Jahre älter als die Aktivitätsdiagramme (siehe 2.3.1) aus der UML und können als Vorläufer der Kontrollstrukturen betrachtet werden, die in den Aktivitätsdiagrammen auch verwendet werden. Beide, Programmschemata und Aktivitätsdiagramme, besitzen ähnliche Kontrollstrukturen, weshalb es Sinn macht, Klassen von Programmschemata, wie zum Beispiel das GOTO – Schemata oder das WHILE – Schemata zu betrachten. Für den Entwurf einer Kontrollkomponente, die Graphersetzungen steuern soll, sind Erkenntnisse aus diesem Gebiet als eine Grundlage aufzufassen, um Kontrollstrukturen zu bewerten.

Wenn sich Programmierer über Programmiersprachen unterhalten, so treffen sie oft Aussagen, wie zum Beispiel die, dass die eine Programmiersprache ausdruckskräftiger sei als eine andere Programmiersprache. Die Theorie der Programmschemata beschäftigt sich im Zentrum ihres Interesses genau mit diesen Fragen und versucht Methoden bereitzustellen, um die Entscheidung solcher Aussagen zu ermöglichen. Erkenntnisse aus diesem Zweig der Theoretischen Informatik entnehme ich insbesonders aus der Dissertation von Can Adam Albayrak mit dem Titel „Die WHILE Hierarchie für Programmschemata“ [Albayrak98].

Doch zunächst soll der Begriff Programmschema eingeführt werden.

2.2.1 Was sind Programmschemata?

Zum Verständnis dieses Begriffes ist es notwendig, Programme, wie wir sie kennen, in ihre zwei grundlegenden Bestandteile zu trennen.

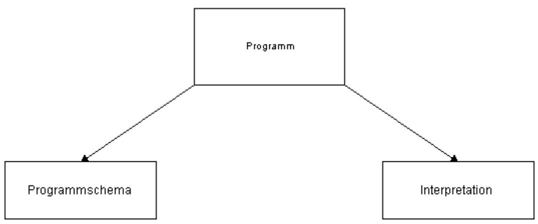


Abbildung 10 Zerlegung eines Programms [Albayrak98/S. 2]

Neben dem Kontrollfluss (Programmschema), der zum Beispiel aus Schleifen und bedingten Verzweigungen bestehen kann, existieren in Programmen auch noch Funktionen mit bestimmten Grundbereichen, die es erst ermöglichen, Programme zu interpretieren. Nun ist es also denkbar, dass zwei Programme die gleichen Kontrollstrukturen besitzen aber unterschiedliche Funktionen berechnen. Daher gelten Programme nur dann als äquivalent, wenn sie die gleiche Funktion berechnen [Albayrak98]. Das Programmschema besitzt auf Grund dieser Definition keine primäre Rolle, wichtig ist nur die berechnete Funktion des Programms.

In der Theorie der Programmschemata hingegen interessiert man sich für die Äquivalenz von Programmschemata, um zum Beispiel Vereinfachungen zu erkennen. Folgendes Programmschema soll helfen, Programmschema und Interpretation einerseits zu trennen und andererseits den Äquivalenzbegriff für Programmschemata zu verdeutlichen:

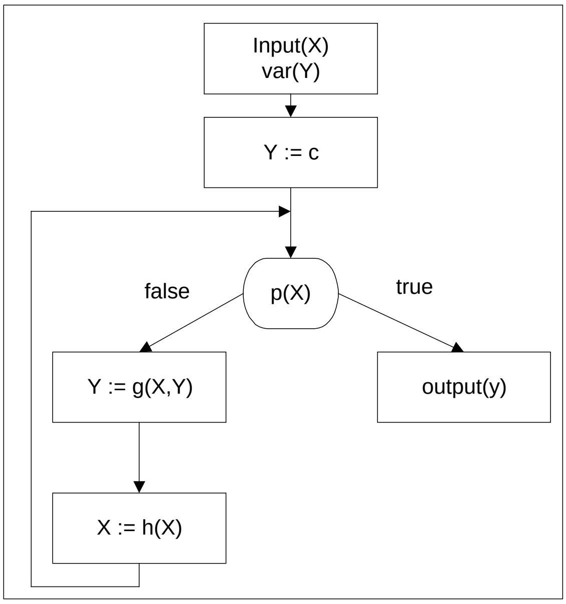


Abbildung 11 Beispiel eines Programmschema [Albayrak98/S. 3]

Es ist erkennbar, dass das Programmschema aus einer bedingten Verzweigung, einer Schleife und mehreren sequentiellen Programmanweisungen besteht. Das Programmschema geht aber erst in ein Programm über, wenn den Symbolen *c*, *p*, *g* und *h* eine Interpretation hinzugefügt wird. Folgende Interpretation für diese Symbole ist denkbar:

- Die Konstante *c* wird durch die natürliche Zahl eins ersetzt.
- Das einstellige Prädikat *p* ist der Test auf das neutrale Element hinsichtlich der Addition.
- Das zweistellige Operationssymbol g soll die Multiplikation von zwei natürlichen Zahlen berechnen.
- Das einstellige Operationssymbol h vermindert ihr Argument um die Zahl eins.

[Albayrak98]

Diese Interpretation ist also auf den Grundbereich der Natürlichen Zahlen zugelassen. Erst jetzt ist es möglich dieses Programm vollständig zu interpretieren, nämlich als die Berechnung n! (Fakultät von n) zu einer gegebenen natürlichen Zahl.

In der Theorie der Programmschemata möchte man als einziges Unterscheidungsmerkmal von Programmiersprachen die Kontrollstrukturen zulassen und von der konkreten Interpretation absehen. Somit definiert man die Äquivalenz von Programmschemata wie folgt:

- „ Zwei Programmschemata S1 und S2 heißen genau dann äquivalent, wenn für alle Interpretationen I der Grundfunktionen gilt, dass das Programm P1, bestehend aus S1 und I, und das Programm P2, bestehend aus S2 und I, die gleiche Funktion berechnen.

[Albayrak98/S. 2]

Demzufolge impliziert die Programmschemaäquivalenz die gewöhnliche Programmäquivalenz. Es ist festzustellen, dass Programmschemata eine wichtige Voraussetzung machen, nämlich die, dass Prädikatensymbole und Funktionssymbole deterministische sein müssen. Bei der Graphersetzung wird der Startgraph jedoch durch Produktionsregeln nicht deterministisch verändert. Folgende Abbildung verdeutlicht diesen Zusammenhang:

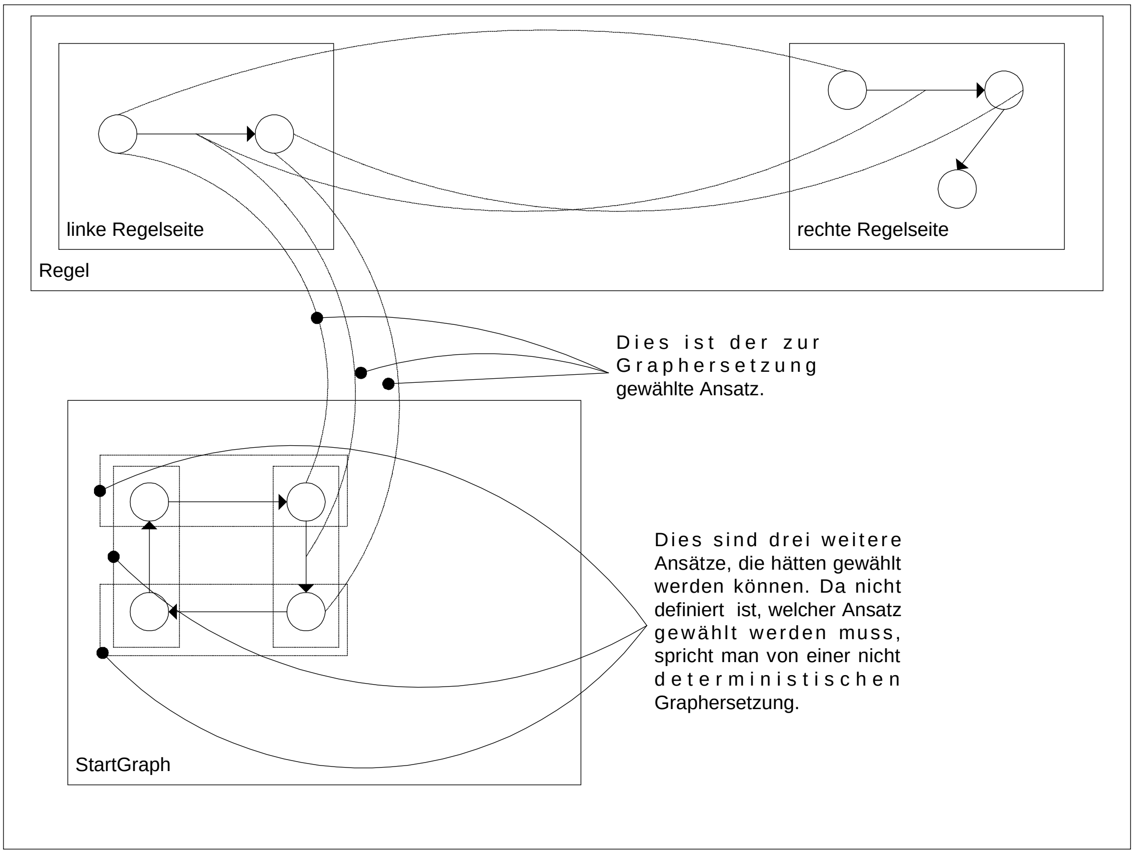


Abbildung 12 Nicht deterministische Graphersetzung

Die Abbildung 12 zeigt, dass ein Ansatz zur Graphersetzung gewählt wird. Daneben gibt es noch weitere Ansätze, die nicht gewählt wurden. Welcher Ansatz gewählt werden muss ist nicht definiert. Es hätte ebenso gut einer der anderen Ansätze gewählt werden können.

Das Konzept der Programmschemata die Kontrollstruktur und die Interpretation zu trennen (jedes Programm enthält diese zwei Komponenten) lässt im Hinblick auf den Entwurf einer Kontrollkomponente, die Graphersetzungen steuern soll folgern, dass Kontrollstruktur und Interpretation auch für die Kontrollkomponente definiert werden muss.

Zunächst sollen Graphersetzungen, die durch die Kontrollkomponente gesteuert werden sollen, auf einer Graphgrammatik basieren, die einen Arbeitsgraphen und eine beliebige Menge (ungleich Null) an Produktionsregeln besitzt. Auf dieser Graphgrammatik soll ein nicht deterministisches Prädikat und eine nicht deterministische Funktion definiert sein:

- Das Prädikat testet, ob eine Produktionsregel im Startgraphen anwendbar ist oder nicht. Ist sie anwendbar liefert sie als Rückgabewert *true* zurück, sonst den Wert *false*. Dieses Prädikat soll die einstellige Funktion *p* sein. Als Argument erhält es eine Produktionsregel.
- Die Funktion soll eine Produktionsregel auf den Arbeitsgraphen anwenden; war diese Regel anwendbar, so liefert diese Funktion den Wert *true*, sonst den Wert *false* zurück. Im folgenden wird sie mit der einstelligen Funktion *f* bezeichnet, die eine Produktionsregel als Argument bekommt.

Die Interpretation alleine reicht nicht aus, um Graphersetzungen zu steuern. Daher müssen noch geeignete Kontrollstrukturen gewählt werden, die auf die definierten Prädikate reagieren können. Im folgenden findet eine Analyse von Programmschemaklassen statt.

2.2.2 Klassen von Programmschemata

Aus der Tatsache, dass unterschiedliche Klassen von Programmiersprachen existieren, wie etwa funktionale Programmiersprachen und imperative Programmiersprachen, ist ableitbar, dass auch unterschiedliche Klassen von Programmschemata existieren müssen:

GOTO – Schemata

Die erste Klasse, die vorgestellt werden soll, ist die Klasse der GOTO – Schemata . Das Hauptmerkmal dieser Programmschemata ist die Möglichkeit, an beliebige Stellen des Programms springen zu können (*GOTO*). Daneben gibt es die Möglichkeit der bedingten Verzweigung *(if-then-else),* sowie die Spezifikation einer Grundfunktion. Eine bekannte Programmiersprache, die solche Schemata produziert, ist zum Beispiel FORTRAN. Hier ein Beispiel einer Graphersetzung, das auf das oben definierte Prädikat *p* und die Funktion *f* basiert:

```
1.  IF p (R1) THEN GOTO 2 ELSE GOTO 4
2.  f (R1) ;
3.  f (R1) ;
4.  IF f (R2) THEN f (R2) ELSE GOTO 8
5.  f (R3) ;
6.  f (R4) ;
7.  GOTO 2
8.  f (R5)
```

Abbildung 13 GOTO Programm

Die Abbildung 13 zeigt ein GOTO – Programm, das eine Graphersetzung steuert. Als erstes wird geprüft, ob die Produktionsregel *R1* anwendbar ist. Kann sie angewendet werden, wird auf die Marke *2* gesprungen (siehe Zeile 1), andernfalls auf die Marke *4*, wo eine weitere bedingte Verzweigung (if-then-else) zu sehen ist. In Zeile 2 und 3 wird die Produktionsregel *R1* zweimal auf den Arbeitsgraphen angewendet. Beim ersten Mal (siehe Zeile 2) wird der Arbeitsgraph

verändert, denn die Produktionsregel war ja anwendbar (siehe Zeile 1). Beim zweiten Aufruf von *f (R1)* (siehe Zeile 3) kann nicht gesagt werden, ob es zu einer Graphersetzung kommt oder nicht. Die bedingte Verzweigung in der Zeile 4 bringt die Problematik der nicht deterministischen Graphersetzung auf den Punkt. Hier wird die Produktionsregel *R2* einmal angewendet. Konnte sie einmal angewendet werden, erhält die if – Anweisung ein *true* als Rückgabewert und die *then* – Anweisung wird ausgeführt. Dort wird *R2* ein zweites Mal aufgerufen, nur kann aus der *if* – Anweisung nicht geschlossen werden, dass *R2* in der *then* – Anweisung überhaupt ein zweites Mal anwendbar ist. Daher lässt sich sagen, dass die Regelanwendungen in den Zeilen 5 und 6 auf „gut Glück" probiert werden. In Zeile 7 kann wiederum nicht ausgesagt werden, ob R3 und R4 anwendbar waren oder nicht.
GOTO – Schemata zeigen als Hauptmerkmal die Eigenschaft, durch Sprungmarken Schleifen zu erzeugen. Jeder Programmierer, der solche Kontrollstrukturen schon einmal programmiert hat, weiß, dass Sprünge auf Sprungmarken zu einem unleserlichen Programmiercode führen. Würde man die Kontrollstruktur von GOTO – Schemata als einen Graphen zeichnen, hätte man ein schwer zu durchschauendes Netz miteinander verbundener Knoten. Folgende Abbildung zeigt, wie das bereits oben vorgestellte einfache GOTO – Programm stark vernetzte Graphen erzeugt:

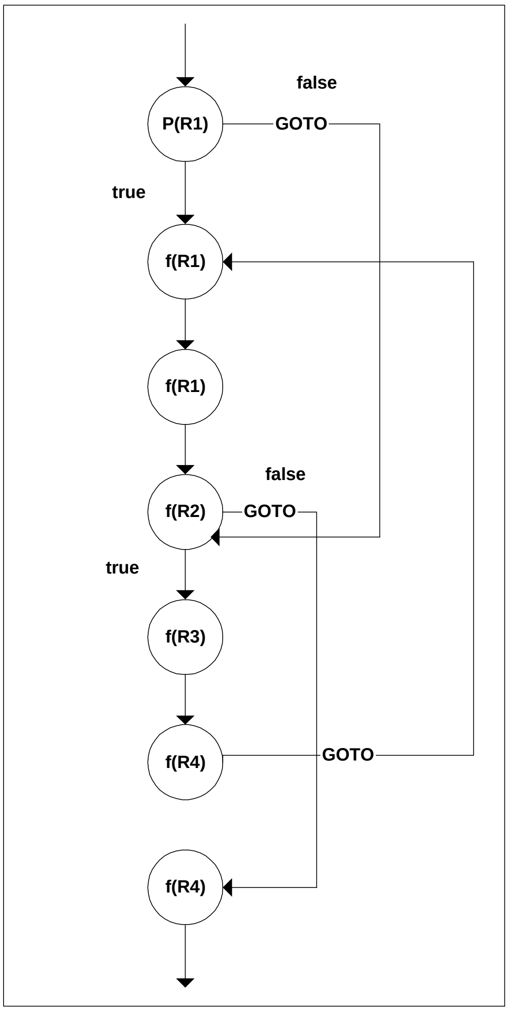


Abbildung 14 Unleserliche Programmschleifen in GOTO - Schemata

Die im folgenden beschriebenen WHILE – Schleifen zeigen dagegen strukturiertere Schleifen als GOTO – Schemata:

WHILE-Schemata

Die Klasse der While – Schemata ist von vielen Programmiersprachen, wie zum Beispiel Java, C, C++ und Modula ableitbar. Ein großer Vorteil dieser Programmiersprachen ist der Gewinn an Lesbarkeit des Programmcodes durch die darunter liegenden WHILE – Schemata. WHILE Schleifen erlauben den Eintritt in eine Schleife nur an fest vordefinierten Stellen. Diese Stellen werden meistens (daher kommt der Name dieses Schemas) mit dem Schlüsselwort „WHILE“ eingeleitet. Weiterhin ist es erlaubt WHILE – Schemata ineinander zu schachteln. Sequenzielle Anweisungen und bedingte Verzweigungen, die ja auch bei den GOTO – Schemata vorhanden sind, kennen WHILE – Schleifen auch. Im folgenden wird, unter anderem, die Graphersetzung durch eine WHILE – Schleife kontrolliert:

```
1.  if f(R2)  then
2.       while p(R1)
3.       do
4.            f ( R1) ;
5.       end
6.       f (R6) ;
7.  else
8.  f (R4) ;
9.  end
```

Abbildung 15 WHILE – Programm

Zur Abbildung 15: Der Schleifeneintritt wird mit dem Schlüsselwort *while* eingeleitet (siehe Zeile 2). Die Schleife wird in Abhängigkeit des Prädikats *p(R1)* bearbeitet. Ist also die Produktionsregel *R1* anwendbar, so wird durch *f (R1)* eine Graphersetzung veranlasst. Dies wird so lange wiederholt, wie das Prädikat *p(R1)* den Rückgabewert *true* liefert.

Durch den definierten Schleifeneintritt werden die Kontrollstrukturen dieser Klasse bei der grafischen Darstellung wesentlich lesbarer:

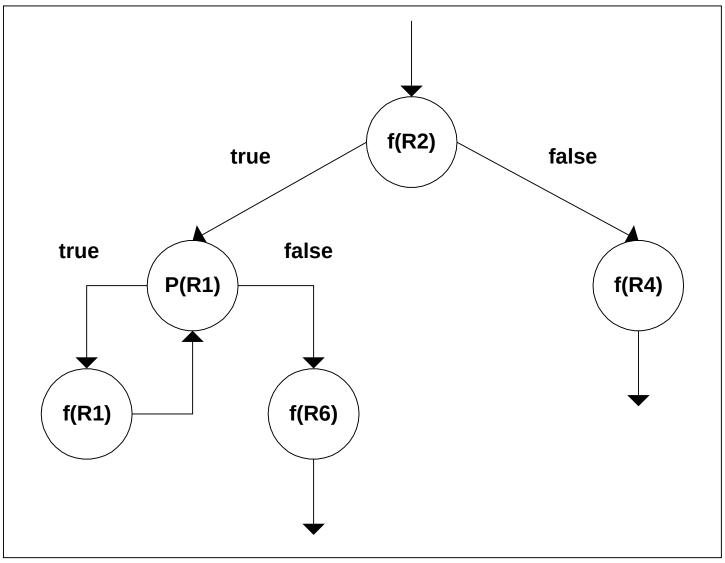


Abbildung 16 Lesbare WHILE – Schleifen

Der Vorteil der WHILE – Schleifen liegt auf der Hand, einerseits beschreiben sie gut leserliche Programme, andererseits ist auch die grafische Notation der Kontrollstrukturen dieser Klasse gut lesbar.
Die folgenden Rekursionsschemata zeigen hierarchische Kontrollstrukturen.

Rekursionsschemata

Eine weitere Klasse der Programmschemata ist die Klasse der Rekursionsschemata. Diese Klasse der Programmschemata lässt sich aus den funktionalen Programmiersprachen, wie zum Beispiel Lisp oder Gofer, ableiten. Das Programm solcher Programmiersprachen besteht aus einer Reihe von Funktionsdefinitionen, die wiederum aus vorgegebenen Grundfunktionen generiert werden können. Neben der bedingten Verzweigung besteht bei dieser Klasse das Hauptmerkmal darin, dass die zu definierende Funktion sich selber rekursiv aufrufen kann. Folgende Graphersetzung verdeutlicht die Rekursionsschemata:

1. Solange (Regelname) = { f (Regelname) ; Solange (Regelname) , falls p (Regelname) = true }
   { *exit*, falls p (Regelname) = false }

Abbildung 17 Rekursionsschemata

Die Abbildung 17 zeigt, wie Rekursionen hierarchische Kontrollstrukturen aufbauen, indem der Wiederaufruf einer wohldefinierten Funktion gestattet wird. Die Funktion in diesem Beispiel heißt *Solange.* Als Argument erhält sie den Namen einer Produktionsregel. Die Funktionsdefinition findet in dem Abschnitt statt, der mit geschweiften Klammern umschlossen ist. Vor dem Komma in diesem Abschnitt wird eine sequenzielle Anweisungsfolge spezifiziert, die in Abhängigkeit der Bedingung nach dem Komma ausgeführt wird oder nicht. Daher wird die Produktionsregel *Regelname* so oft angewendet, wie sie anwendbar ist. Die zweite Zeile spezifiziert dieselbe Funktion *Solange* und gibt an, wann die Funktion verlassen werden darf, nämlich dann, wenn *Regelname* nicht mehr anwendbar ist.
Die grafische Darstellung solcher Kontrollstrukturen führt zu komplizierten Graphenstrukturen, wie in einem späteren Zusammenhang näher erläutert wird (siehe Kapitel 3.3.1). Andererseits zeigt sie, dass die textliche Darstellung sehr kurz und verständlich ist.
Der folgende Abschnitt zeichnet formal bewiesene Erkenntnisse hinsichtlich Programmschemata mit deterministischen Funktionen auf. Diese Erkenntnisse sind auf Grund der nicht deterministischen Eigenschaft der Graphersetzung nicht ohne weiteres auf Programme anwendbar, die Graphersetzungen steuern (für die das Prädikat *p* und die Funktion *f*). Aber um das Verständnis für wichtige Begriffe wie Äquivalenz und Übersetzbarkeit von Programmschemata zu erlangen, soll das folgende Kapitel eingebracht werden.

2.2.3 Monadische Programmschemata

In der Praxis hat sich gezeigt, dass die Vereinfachung der Programmschemata, insofern als, dass die Operationssymbole und Prädikatensymbole auf den eindimensionalen Zustandsraum abgebildet werden (siehe Abbildung 18), Beweisführungen vereinfachen. Somit sind Operations- und Prädikatensymbole alle nur noch einstellig und das Programm arbeitet auf einer einzigen monadischen Variablen, die konventioneller Weise bei der Notation von monadischen Programmschemata weggelassen wird. Diese Variable ist auf den Bereich der natürlichen Zahlen definiert. Daher stellt diese Tatsache ein zusätzliches Hindernis dar, die folgenden Sätze auf die Steuerungsstrukturen für Graphersetzungen anzuwenden, da hier Graphen als Datenstrukturen verwendet werden.

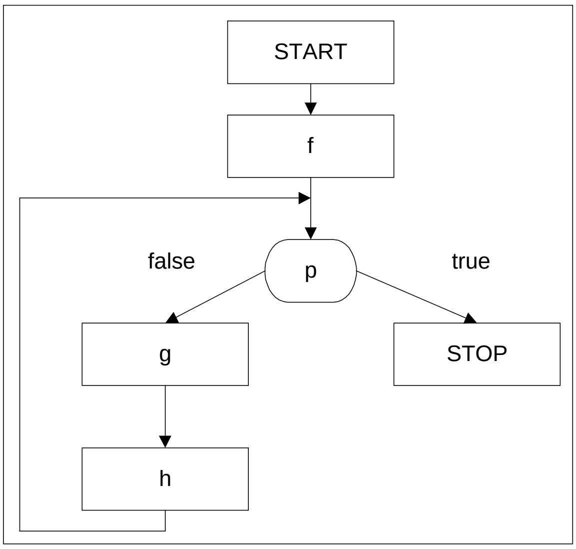


Abbildung 18 Monadisches Programmschemata

Die zuvor vorgestellten Klassen von Programmschemata sind nach den Namen der Theoretiker, die diese Schemata untersuchten, benannt. Folgende Tabelle soll die Bezeichnung der monadischen Schemata aufzeigen:

	Monadische Bezeichnung
GOTO – Schemata	Ianov – Schemata
Rekursionsschemata	De – Bakker – Scott – Schemata
WHILE – Schemata	Dijkstra – Schemata

2.2.4 Sätze über Programmschemata

In dieser Diplomarbeit soll nicht darauf eingegangen werden, wie diese Erkenntnisse bewiesen wurden. Besteht mehr Interesse hinsichtlich der Beweisführung so empfehle ich [Albayrak98], wo auch detaillierte Verweise auf Literatur zu entnehmen sind.

Äquivalenz von Programmschemata

Im mehrdimensionalem Zustandsraum sind Goto – Schemata und Rekursionsschemata in bezug auf ihre Äquivalenzaussagen unentscheidbar. Betrachtet man jedoch ihre monadischen Programmschemata, so ändern sich einige Aussagen:

- Ianov – und Dijkstra – Schemata sind entscheidbar bezüglich ihres Äquivalenzproblems.
- De – Bakker – Scott – Schemata sind auch entscheidbar bezüglich ihres Äquivalenzproblems, sofern keine Konstanten betrachtet werden, dann nämlich wird dieses Problem unentscheidbar.

Hier noch ein Beispiel für Äquivalente Programmschemata:

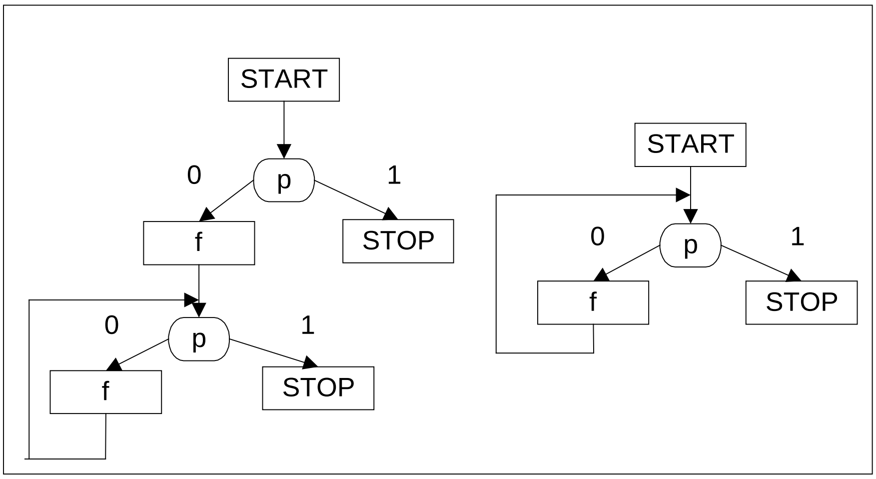


Abbildung 19 Äquivalente Programmschemata

Beginnt man beide Programmschemata gleichzeitig zu durchlaufen, so erkennt man, dass nach jedem Bearbeitungsschritt der gleiche Teilgraph in beiden Programmschemata, bearbeitet wird.

Die Übersetzbarkeit von Programmschemata für den Datenraum der Natürlichen Zahlen

Die zentrale Intention bei dieser Problematik liegt darin, eine Aussage treffen zu können, ob ein Schema S1 aus ihrer Klasse KS1 übersetzbar in ein anderes Schema S' aus einer anderen Klasse KS' ist. Folgende Schaubilder aus [Albayrak98] verdeutlichen diesen Zusammenhang:

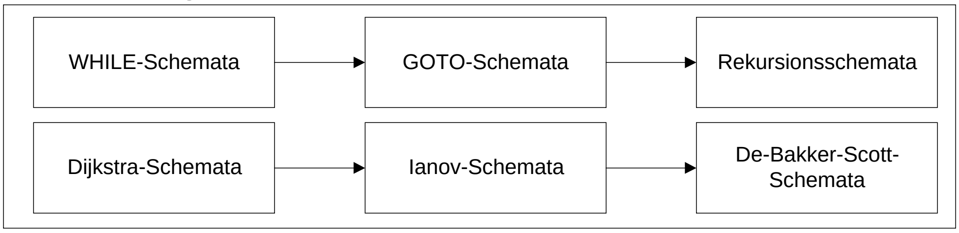


Abbildung 20 Übersetzbarkeit von Programmschemata

Die Pfeilrichtung soll zeigen, dass die Umkehrrichtung im allgemeinen nicht möglich ist.

Das Problem der Freiheit von Programmschemata

Zur informellen Beschreibung des Begriffs Freiheit soll folgende Abbildung dienen:

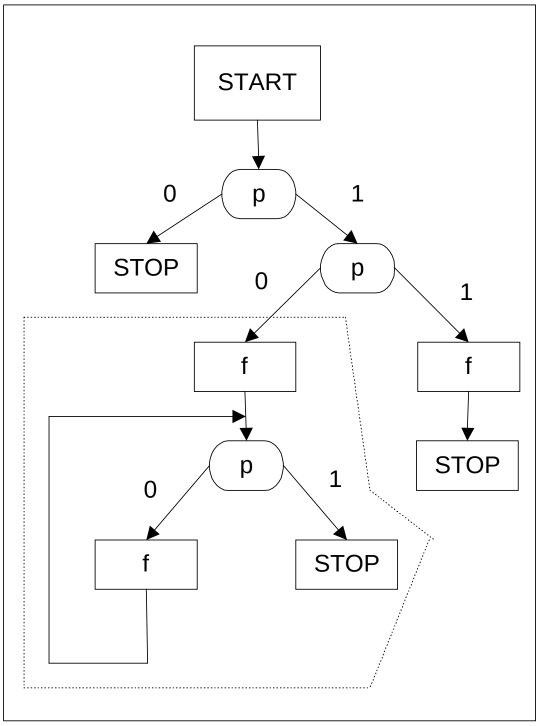


Abbildung 21 Freiheit von Programmschemata

Der umrandete Teilgraph kann, auf Grund der deterministischen Eigenschaft von *p*, nie durchlaufen werden. Dieses Programmschema ist nicht frei, da es keinen Pfad gibt, bei dem dieser Programmabschnitt durchlaufen wird.

Beliebige GOTO – Schemata sind im allgemeinen unentscheidbar bezüglich ihrer Freiheit. Betrachtet man jedoch wiederum monadische Schemata so können folgende Eigenschaften aufgelistet werden:

- Die Klasse der Ianov – Schemata ist genau dann nicht frei, wenn es einen Pfad mit zwei identischen Prädikatsymbolen gibt, zwischen denen kein Funktionssymbol auftritt.
- Die Klasse der De – Bakker – Schemata ist bezüglich der Freiheit entscheidbar, während sie mit Konstanten nicht entscheidbar ist.
- Die Klasse der Dijkstra – Schemata ist entscheidbar bezüglich ihrer Freiheit.

Programmschemata können konzeptionell nicht auf Kontrollstrukturen, die nicht deterministische Funktionen berechnen, übertragen werden. Die Kontrollkomponente, für die Steuerung von Graphersetzungen in AGG, besitzt eine nicht deterministische Funktion *f* und ein nicht deterministisches Prädikat *p* (siehe Kapitel 2.2.1). Die grafische Darstellung der Dijkstra – Schemata durch Knoten und Kanten ist auf Grund der Leserlichkeit sehr attraktiv. Jedoch ist es nicht möglich über die Funktions- oder Prädikatensemantik hinaus zu interpretieren. In den Aktivitätsdiagrammen hingegen, die aus der UML [UML99] stammen, sind spezielle Knoten (Aktivitäten) definiert, die für die Interpretation von verschiedenen Anwendungen offengehalten wurden. Daher sollen im folgenden Kapitel Aktivitätsdiagramme vorgestellt werden. Da sie einen einzelnen Diagrammtyp aus der UML darstellen, wird versucht, Aktivitätsdiagramme innerhalb der UML einzuordnen.

2.3 Unified Modeling Language

Die UML ist eine objektorientierte Modellierungssprache, die auf einer grafischen Notationssprache basiert, weil komplexe Informationen, die grafisch dargestellt werden, für den Menschen intuitiver zugänglich sind. Andererseits stellen Graphen eine beliebte Datenstruktur dar, um Informationen zu vermitteln. Daher werden in der UML Grundsymbole spezifiziert, die dann in unterschiedlichen Diagrammen eingesetzt werden dürfen, um Zusammenhänge zu modellieren.
Zum Zeitpunkt dieser Diplomarbeit lag sie in der Version 1.3 vor (s. [Rati99]) und wurde seitens der OMG (Object Management Group) im November 1997 zum Standard erhoben.
Die UML besteht aus verschiedenen Diagrammtypen, die wiederum verschiedene grafische Elemente besitzen. Die Semantik der Elemente ist mehr oder weniger genau festgelegt. Es kann in der UML aber vorkommen, dass für ein und denselben Vorgang mehrere Darstellungsarten verwendet werden können. Die UML geht aus mehreren objektorientierten Modellierungsmethoden hervor. Die wichtigsten sind Object Modeling Technique (OMT), Object oriented Software Engineering (OOSE), und Booch (vgl. [Jaco92], [Rumb93] und [Booc94]).

Die Notation der UML umfasst Diagramme für die Darstellung der verschiedenen Sichten auf Systeme. Für eine spezielle Aufgabe ist meist eine Diagrammart besser geeignet als andere. Insgesamt umfasst die UML folgende Diagrammtypen:

- Klassendiagramm
- Objektdiagramm
- Anwendungsfalldiagramm
- Sequenzdiagramm
- Zustandsdiagramm
- Aktivitätsdiagramm
- Kollaborationsdiagramm
- Komponentendiagramm
- Einsatzdiagramm

Hinzu kommen noch in jedem Diagramm verschiedene Stilelemente mit verschiedenen Beschriftungen. Insgesamt ist also die UML eine umfangreiche, grafische Sprache. Das Klassendiagramm und das Objektdiagramm modellieren die statische Struktur eines Systems. Die restlichen Diagramme hingegen modellieren das dynamische Verhalten eines Systems. Diagramme sind im allgemeinen zusammengesetzte Graphen, bestehend aus Knoten und Kanten, welche die Visualisierung eines Systems aus verschiedenen Sichten ermöglichen.
Die Steuerung von Graphersetzungen stellt einen dynamischen Aspekt der Kontrollkomponente dar. Die Dijkstra – Schemata haben sich als attraktive Schemata für die Steuerung von Graphersetzungen herausgestellt. Die Aktivitätsdiagramme erlauben es, ähnliche

Kontrollstrukturen aufzubauen, weshalb sie im folgenden Kapitel behandelt werden. Dabei wird versucht, die Semantik der Graphersetzung auf Aktivitätsdiagramme zu übertragen.

2.3.1 Aktivitätsdiagramme

Aktivitätsdiagramme gehören zu den Darstellungstechniken, in der UML, die das dynamische Verhalten von Systemen beschreiben, jedoch den Schwerpunkt in der Dokumentation von Abläufen setzen. Sie sind den Flussdiagrammen (s. Programmschemata) sehr ähnlich, besitzen aber weitaus mehr Modellierungsmöglichkeiten. Der wesentliche Unterschied zwischen Aktivitätsdiagrammen und Flussdiagrammen liegt darin, dass mit Hilfe von Aktivitätsdiagrammen auch parallele Abläufe und die Synchronisation mehrerer *Threads* (siehe Kapitel 4.1) dargestellt werden können. Ein Aktivitätsdiagramm ist im wesentlichen ein Flussdiagramm, das den Kontrollfluss von Aktivität zu Aktivität zeigt. Eine Aktivität ist ein andauernder nicht atomarer Ablauf in einem Automaten. Aktionen hingegen können als Spezialfall der Aktivitäten aufgefasst werden, die atomar sind und daher nicht unterbrochen werden können. Um zu entscheiden, ob die Kontrollkomponente Aktivitäten oder Aktionen modelliert, müssen das in dem Kapitel 2.2 vorgestellte Prädikat p und die Funktion f diesbezüglich interpretiert werden. In den folgenden Abschnitten findet eine Zuordnung dieser Prädikate in Aktivitätsdiagrammen statt.

Das Grundelement in den Aktivitätsdiagrammen stellt der Aktionszustand dar (zur Veranschaulichung siehe Abbildung 22):

Aktions- und Aktivitätszustände

Während des Ablaufs des Kontrollflusses, den ein Aktivitätsdiagramm modelliert, finden Ereignisse statt. Die Ereignisse die in der Kontrollkomponente geschehen können, sind durch die Funktion f definiert, denn diese Funktion wendet eine Produktionsregel auf den Arbeitsgraphen an. In bezug auf das System, das Graphersetzungen steuern soll (die Kontrollkomponente in AGG), stellt dieses Ereignis ein atomares Ereignis dar. Auf Aktivitätsdiagramme übertragen handelt es sich dann dabei um Aktionen, die ja ausführbare atomare Verarbeitungsschritte sind. Als Aktionszustand bezeichnet man ein Symbol (durch eine Pastillenform dargestellt), das einen Zustand repräsentiert, indem eine Aktion ausgeführt werden soll. Die Aktion selbst kann beliebig beschriftet werden. Damit führt der Aktionszustand die Aktion *f (Regelname)* aus. Möchte man auf den Rückgabewert der so spezifizierten Aktion reagieren, können zwei Zustandsübergänge (siehe unten) aus dem Aktionszustand spezifiziert werden, die dann entsprechende Wächterbedingungen ([*true*] oder [*false*]) erhalten.
Aktivitätszustände hingegen unterscheiden sich nur dadurch von Aktionszuständen, dass sie unterbrechbare und daher nicht atomare Aktivitäten beschreiben. Sie werden ebenfalls in einer Pastillenform dargestellt. UML lässt als einziges Unterscheidungsmerkmal zwischen Aktivitätszuständen und Aktionszuständen die Beschriftung in der Pastillenform zu.

Zustandsübergänge

Ist die Aktion (oder Aktivität) eines Zustandes beendet, geht die Kontrolle automatisch an den nächsten Aktionszustand (oder Aktivitätszustand) über. Die Spezifikation eines Zustandsüberganges, also der Übergang von einem Aktivitätszustand in den nächsten, wird durch einen Pfeil (gerichtete Kante) vorgenommen. Die Zustandsübergänge vollziehen sich meistens ohne Auslöser (s. Abbildung 22), sie folgen automatisch nach Beendigung eines Aktivitätszustandes.

Treten aus einem Aktivitätsdiagrammknoten mehrere Zustandsübergänge aus, so muss eine deterministische Wahl dieser Kanten beim Zustandsübergang möglich sein. Daher können sogenannte Wächter (boolesche Ausdrücke in eckigen Klammern) den Zustandsübergängen mitgegeben werden. Es darf aber immer nur einer dieser Wächter als *true* ausgewertet werden dürfen. Die Kontrollkomponente verwendet als Wächter den Test auf die Anwendbarkeit einer Produktionsregel (das Prädikat *p*) als booleschen Wächter.

Verzweigungen und Schleifen

In der Regel können Systeme nicht allein durch eine lineare Abfolge von Aktionen, beschrieben werden, weshalb Aktivitätsdiagramme Verzweigungen erlauben. Verzweigungen sind grafisch durch eine Raute dargestellt. Aufgrund des Verzweigungscharakters des Verzweigungssymbols, treten zwei oder mehr Kanten aus dem Verzweigungssymbol aus. Das Verzweigungssymbol entspricht einer bedingten Verzweigung, sofern keiner seiner Zustandsübergänge auf Ausgangszustände zurückführt.

Aktivitätsdiagramme stellen keine wohlstrukturierten Schleifen zur Verfügung. Schleifen entstehen einfach dadurch, dass beliebige Zustandsübergänge auf Ausgangszustände zurückführen können (zum Beispiel auf ein Verzweigungssymbol oder ein Aktionszustand). Dies entspricht den GOTO – Schemata aus Kapitel 2.2.2. Die Nachteile, der so entstehenden Schleifen, in Hinsicht auf die grafische Darstellung, sind sehr stark vernetzte Graphenstrukturen. Aus diesem Grund wird folgende Modelliermöglichkeit in der Kontrollkomponente eingeschränkt:

- Verzweigungssymbol und Aktionssymbol dürfen keine Zustandsübergänge haben, die auf ihre Ausgangssymbole zurückführen. Somit werden Schleifen aus diesen Zuständen verboten. Aus dem Verzweigungssymbol und dem Aktionszustand, der auf den Rückgabeparameter des Prädikats *f* reagiert, werden dadurch bedingte Verzweigungen (if-then-else) - Anweisungen. Das Verzweigungssymbol reagiert dabei auf eine Vorbedingung und der Aktionszustand auf eine Nachbedingung.

Da es unter dieser Voraussetzung keine Schleifen mehr geben kann wird in Anlehnung an die WHILE – Schleifen der Dijkstra – Schemata ein Schleifensymbol eingeführt, welches eine WHILE – Schleife realisiert. Die Bedingung, ob in eine Schleife eingetreten wird oder nicht, ist, wie beim Verzweigungssymbol auch, eine Vorbedingung.

Start- und Stopzustand

Jedes Aktivitätsdiagramm beginnt mit einem eindeutigen Startzustand. Der Startzustand hat nur einen Zustandsübergang. Dieser wird automatisch ausgelöst. Der Startzustand spezifiziert lediglich den Eintritt in einen Kontrollfluss. Die Terminierungspunkte hingegen werden vom Stopzustand spezifiziert.

Die Kontrollkomponente beginnt an einem Startzustand mit der Transformation und beendet sie an einem Stopzustand.

Nebenläufigkeit in Aktivitätsdiagrammen

Der wesentlichste Unterschied zwischen Aktivitätsdiagramm und Programmschema liegt darin, dass das Aktivitätsdiagramm parallele Abläufe darstellen kann. Zur Aufspaltung des Kontrollflusses wird das Gabelungssymbol verwendet. Eine Gabelung besitzt einen Eingang und mehrere Ausgänge. Alle Nachfolgeaktionen werden von diesem Zeitpunkt an parallel ausgeführt.

Zur Zusammenführung des Kontrollflusses kann das sogenannte Synchronisationssymbol verwendet werden, das in Analogie zum Gabelungssymbol mehrere Eingänge und nur einen Ausgang hat. Die Aktion, die am Ende des Ausganges eines Synchronisationssymbols steht, wird erst dann ausgeführt, wenn alle Aktionen an den Eingängen des Synchronisationssymbols beendet sind.

Es gibt Theorien über parallele Graphersetzung[1]. Das Graphersetzungssystem AGG kann jedoch keine parallele Graphersetzung ausführen; daher wird für die Kontrollkomponente eine andere Semantik eingeführt. Das Gabelungssymbol soll nämlich einen der Ausgänge zufällig wählen und einen Zustandsübergang ausführen.

Die folgende Abbildung zeigt, wie sich durch Aktivitätsdiagramme die Steuerung von Graphersetzungen modellieren lässt:

[1] G. Rozenberg et al. (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3 (Concurrency). World Scientific, Singapoore, 1999.

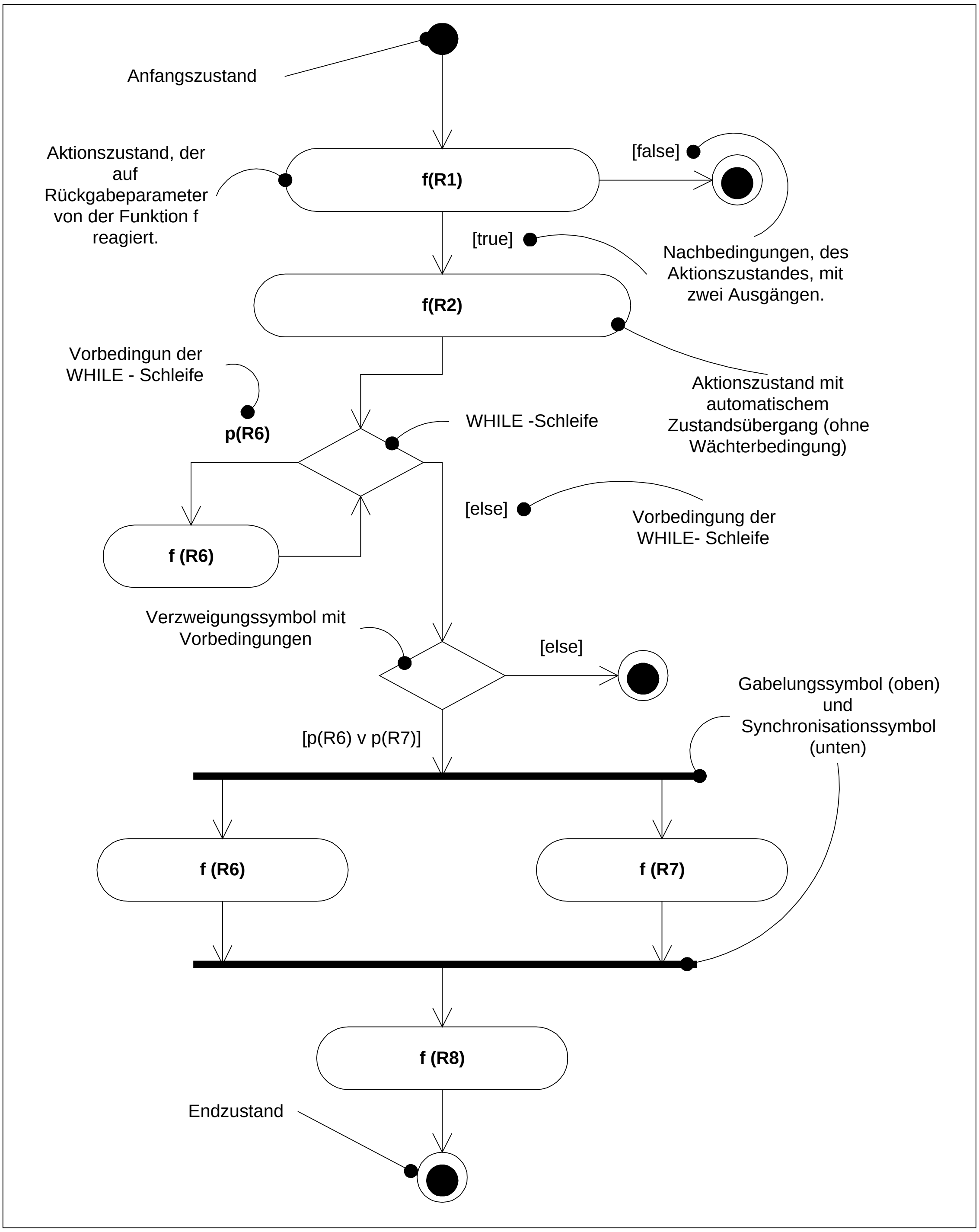


Abbildung 22 Grundelemente der Kontrollkomponente

Im nächsten Kapitel wird die Konzeption der Kontrollkomponente vorgestellt. Die Kontrollkomponente soll auf Aktivitätsdiagrammen basieren, die auf die Steuerung der Graphersetzung übertragen wurden. Die Abbildung 22 zeigt ein solches Aktivitätsdiagramm.

3 Der Steuerungseditor

Bislang wurden unterschiedliche Kontrollstrukturen (Programmschemata u. Aktivitätsdiagramme) vorgestellt. Die in Kapitel 2.2 vorgestellten Programmschemata (WHILE – Schemata, GOTO – Schemata u. Rekursionsschemata) bieten verschiedene Möglichkeiten an, um den Kontrollfluss eines Programms zu spezifizieren. Insbesondere ist festzustellen, dass die WHILE – Schemata (siehe Kapitel 2.2.1) sehr lesbare Kontrollstrukturen anbieten, weswegen sie auch in allen höheren Programmiersprachen wie C, C++ oder Java angeboten werden. Weiterhin wurde festgestellt, dass die Programmschemata nur bedingt für die Steuerung von Graphtransformationen einsetzbar sind, denn Programmschemata arbeiten auf einer einzigen monolithischen Variablen, die für die Datenstruktur der natürlichen Zahlen definiert ist. Das für den Steuerungseditor zugrunde liegende Anwendungsszenario, nämlich die Steuerung von Graphersetzungen, hat jedoch als Basisstruktur den Graphen (siehe Definition 1). Ein weiterer kritischer Punkt ist der, dass die Graphersetzung nicht deterministisch ist, während aber die Prädikaten- und Funktionssymbole in den Programmschemata ausschließlich deterministisch sind (siehe Kapitel 2.2.1). Es ist aber nicht möglich eine Funktion, die deterministisch sein muss, zu spezifizieren, um eine nicht deterministische Graphersetzung zu berechnen.

In Kapitel 2.3 wurden die Aktivitätsdiagramme eingeführt, die ähnlich wie Programmschemata auch Kontrollstrukturen anbieten. Es ist festgestellt worden, dass Aktivitätsdiagramme einerseits differenziertere Kontrollstrukturen aufweisen können (Gabelungs- und Synchronisationssymbole), aber andererseits keine traditionell bewährten Kontrollstrukturen wie das WHILE anbieten. Daher wird in diesem Kapitel eine an das Aktivitätsdiagramm angepasste WHILE – Schleife eingeführt. Weiterhin wurde gezeigt, dass die Aktivitätsdiagramme den Kontrollfluss von Aktivität zu Aktivität spezifizieren und die Graphersetzung, dass heißt die Anwendung einer Regel (vorausgesetzt es gibt einen Ansatz) auf den Startgraphen, als Aktivität verstanden werden kann. Da Aktivitätsdiagramme den Kontrollfluss von Aktivitätszustand zu Aktivitätszustand modellieren und ein Aktivitätszustand nichts weiter ist als ein Zustand, in dem eine Aktivität ausgeführt wird, konnten Aktivitätsdiagramme als geeigneter Diagrammtyp für die Steuerung von Graphersetzungen bezeichnet werden. Die Aktivitätsdiagramme unterscheiden, wie erläutert (siehe Kapitel 2.3.1), zwei Zustandstypen, nämlich Aktivitätszustände und Aktionszustände. Sie unterscheiden sich in erster Line dadurch, dass Aktivitätszustände, im Gegensatz zu Aktionszuständen, nicht atomare Aktivitäten ausführen und daher durch Ereignisse unterbrechbar sind. Im Hinblick auf den Steuerungseditor, der die Steuerung von Graphersetzung spezifizieren soll, kann festgestellt werden, dass die Graphersetzung (die Anwendung einer Regel - sofern es einen Ansatz gibt - auf den Startgraphen) als atomare Aktivität aufgefasst werden kann und somit der Steuerungseditor nur Aktionszustände modelliert.

In diesem Kapitel sollen die soeben gezeigten Erkenntnisse, beim Entwurf des Steuerungseditors berücksichtigt werden.

Im ersten Unterkapitel wird ein Anforderungskatalog definiert, dem der Steuerungseditor genügen soll. Dabei wird festgelegt, welche Funktionalitäten der Steuerungseditor bieten wird.

Danach soll in einem weiteren Unterkapitel darauf eingegangen werden, wie die Kontrollstrukturen im Steuerungseditor aussehen, indem eine Graphgrammatik vorgestellt wird, welche die gesamten Einfüge- und Löschoperationen von Steuerungssymbolen im Steuerungseditor spezifizieren.
Ein weiterer Abschnitt widmet sich dem Vergleich mit anderen Graphersetzungssystemen, die auch Kontrollstrukturen anbieten, wie zum Beispiel in PROGRES und Grrr. In erster Linie sollen Gemeinsamkeiten und Unterschiede ausgearbeitet werden.
Der letzte Abschnitt führt ein Systemdesign für den Steuerungseditor durch. Es soll mit Hilfe der UML - Modellierungstechnik der Steuerungseditor und seine Implementierung dokumentiert werden.

Voraussetzungen des Steuerungseditors

Bevor aber mit den Anforderungen an den Steuerungseditor fortgefahren wird, müssen noch einige wichtige Voraussetzungen gemacht werden. Der Steuerungseditor erhält als Eingabe eine vollständige Graphgrammatik. Die Graphgrammatik enthält den Startgraphen und eine Menge von Regeln, die durch einen eindeutigen Namen repräsentiert sind. Mit anderen Worten: Es darf kein Name doppelt vorkommen. Außerdem sollen Graphgrammatiken folgende Operationen kennen:

1. [*boolean*] *führeRegelAus*(*Regelname*): Die Regel mit dem Namen ‚Regelname' wird auf den Startgraphen angewendet. Ist ein Ansatz gefunden und kann eine Graphersetzung durchgeführt werden, so liefert diese Operation *true* als Rückgabewert zurück, sonst *false* (Funktionssymbol *f* aus Kapitel 2.2.1).
2. [*boolean*] *istRegelAnwendbar*(*Regelname*): Diese Operation überprüft, ob es einen Ansatz für die Regel mit dem Namen ‚Regelname' gibt. Wenn ja liefert diese Operation *true* als Rückgabewert, sonst *false* (Prädikatensymbol *p* aus Kapitel 2.2.1)..

Diese Voraussetzungen werden deshalb gemacht, weil der Steuerungseditor auf ein schon vorhandenes Graphersetzungssystem (AGG [http://tfs.cs.t-berlin.de/agg/]) aufgesetzt wird. AGG übergibt den Steuerungseditor eine Graphgrammatik, die alle oben aufgeführten Voraussetzungen erfüllt.

3.1 Anforderungen an den Steuerungseditor

Der Anforderungskatalog an den Steuerungseditor beginnt damit, dass die syntaktische Notation, mit der die Graphersetzung in AGG gesteuert werden soll, an das Aktivitätsdiagramm angelehnt sein soll. Wie schon erwähnt (siehe Einleitung zu Kapitel 3) kann auf Grund der Tatsache, dass einerseits ein neues grafisches Symbol (nämlich das WHILE – Symbol) eingeführt wird und andererseits die Interpretierung der nebenläufigen Abläufe (eingeleitet durch das Gabelungssymbol) im Steuerungseditor anders ist, als sie in der UML beschrieben wurde, nur von einer Anlehnung an das Aktivitätsdiagramm die Rede sein. Die so in Anlehnung an das Aktivitätsdiagramm entstehenden Graphen werden von nun an Steuerungsgraphen genannt. Steuerungsgraphen bestehen aus Knoten, die aus einem Set von Steuerungssymbolen gewählt werden können und aus Kanten, die Zustandsübergänge repräsentieren.
Die zweite wichtige Anforderung an den Steuerungseditor ist die Syntaxsteuerung beim Erzeugen von Steuerungsgraphen. Unter der Syntaxsteuerung soll verstanden werden, dass der Editor zu jedem Editierzeitpunkt für einen konsistenten Steuerungsgraphen sorgt, der durch den

Automaten erst interpretiert und danach ausgeführt werden kann. Der Automat selbst wird in Kapitel 4 beschrieben, doch muss erwähnt werden, dass der Automat der Programmteil des Steuerungseditors ist, der den Steuerungsgraphen interpretiert und somit Graphersetzungen durch AGG veranlasst. Mit anderen Worten: Zu jedem Editierzeitpunkt entsteht ein Steuerungsgraph, der durch den Automaten ausgeführt werden kann.
Die dritte Anforderung ist die Unterstützung des Benutzers bei der grafischen Darstellung der Steuerungsobjekte auf den Editierbereich des Steuerungseditors (Layoutunterstützung). Alle durch Einfügeoperationen spezifizierbare Steuerungssymbole sollen durch den Steuerungseditor, im Sinne späterer Editierbarkeit, günstig auf den Editierbereich platziert werden. Steuerungssymbole werden durch Zustandsübergänge (Kanten) verbunden. Zustandsübergänge werden implizit, beim Erzeugen von Steuerungssymbolen, mit erzeugt. Die Layoutforderung soll den Benutzer in der grafischen Darstellung der Steuerungssymbole unterstützen.
Die letzte Anforderung bezieht sich auf die Editierung selbst: Es wird gefordert, dass alle Operationen, wie zum Beispiel Einfügeoperationen, durch sogenannte Kontextmenüs wählbar sind. Somit kann eine flüssige Editierung des Steuerungsgraphen gewährleistet werden und der Bennutzer wird vor Editierfehlern bewahrt.
Nun folgt im Kapitel 3.1.1 die Erfüllung der ersten Anforderung.

3.1.1 Spezifizierung der Steuerungssymbole im Steuerungseditor

Die erste Anforderung an den Steuerungseditor, die in Kapitel 3.1 aufgelistet wurde, ist die Anlehnung der Steuerungssymbole an das Aktivitätsdiagramm. Daher werden im folgenden alle grafischen Beschreibungsmittel für Aktivitätsdiagramme, die ihrerseits auf Steuerungsgraphen ohne weiteres übertragen werden können, übernommen. Außerdem werden neue Steuerungssymbole, die nicht in Aktivitätsdiagrammen vorkommen, an eine im Aktivitätsdiagramm übliche, grafische Notation überführt und in den Steuerungseditor integriert. Daneben gibt es eine Gruppe von Steuerungssymbolen, die zwar aus den Aktivitätsdiagrammen stammt, aber in einer anderen Bedeutung verwendet wird, als in der UML spezifiziert.
Im folgenden findet eine Spezifizierung der Steuerungssymbole statt. Einige dieser Symbole, wie das Start-, das Stop- und das Aktionssymbol, symbolisieren Zustände. Andere Symbole hingegen, wie das Verzweigungs-, das Gabelungs-, das Synchronisations- und das Wiederholungssymbol (WHILE), beschreiben den Kontrollfluss, repräsentieren also keinen Zustand. Alle diese Symbole sollen jedoch als Steuerungssymbole (oder Steuerungsobjekte) aufgefasst werden, da sie bei der Spezifikation des Steuerungsgraphen maßgeblich beteiligt sind und Steuerungseigenschaften besitzen.

Der Zustandsübergang

Ein Zustandsübergang ist grafisch durch eine gerichtete Kante repräsentiert. Zustandsübergänge können im Steuerungseditor entweder automatisch ausgelöst werden, dann besitzen sie keine Wächterbedingung, oder sie können an eine Wächterbedingung geknüpft sein, von deren Wahrheitsgehalt (Bedingung *true* oder *false*) es abhängt, ob dieser Zustandsübergang ausgelöst wird oder nicht. Alle Zustandsübergänge im Steuerungseditor (im folgenden auch Kanten genannt) sind an eine dieser beiden Zustandsübergangsformen gebunden:

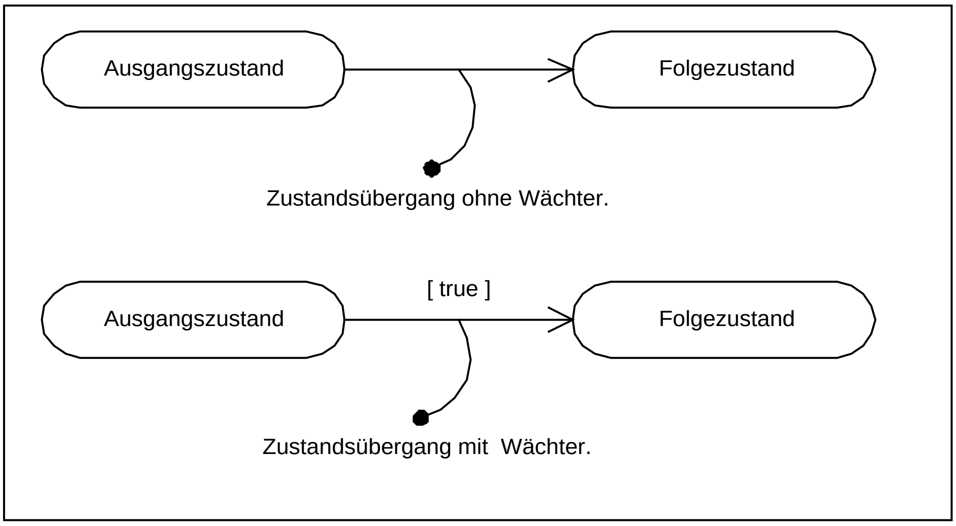


Abbildung 23 Zustandsübergangsformen

Außerdem können Zustandsübergänge niemals alleine stehen und müssen daher immer zwei Steuerungssymbole verbinden. Die Kantenrichtung geht immer vom Ausgangsymbol zum Folgesymbol. Der Steuerungseditor bietet keine explizite Möglichkeit an Zustandübergänge zu erzeugen (siehe dazu Kapitel 3.1.2). Zustandsübergänge entstehen indirekt durch die Erzeugung eines der folgenden Steuerungssymbole.

Der Startzustand

Der Steuerungsgraph des Steuerungseditors besitzt genau einen Startzustand. Dieser definiert den Eintrittspunkt, ab dem der Automat beginnen kann, Graphersetzungen durchzuführen. Die grafische Notation soll aus den Aktivitätsdiagrammen übernommen werden. Insbesondere ist festzustellen, dass der Startzustand (oder Anfangszustand) nur eine ausgehende Kante ohne Wächterbedingung hat und somit automatisch zum Folgesymbol führt.

Der Endzustand

Der Endzustand (oder Stopzustand) kann in einem Aktivitätsdiagramm mehrfach vorkommen. Gelangt der Automat an einen Endzustand, so beendet er die Graphersetzung. Man kann also mit dem Endzustand Terminierungspunkte des Automaten festlegen. An diesen Stellen hört demnach die Graphersetzung auf. Die grafische Notation soll aus den Aktivitätsdiagrammen übernommen werden. Auf Grund seiner Eigenschaft kann der Endzustand keine ausgehende Kante besitzen.

Der Aktionszustand

Der Aktionszustand ist das Hauptelement im Steuerungsgraphen, denn hier befindet sich die eigentliche Aktion (Aktionen sind Spezialfälle von Aktivitäten), die zur Graphersetzung führt. Aktionszustände enthalten Ausdrücke, die eine Aktion beschreiben. Die Aktion selbst soll definiert sein als die Anwendung einer Graphgrammatikregel auf den Startgraphen, sofern sie anwendbar ist. Formaler ist diese Aktion durch die Operation ‚*führeRegelAus*(*Regelname*)' der Graphgrammatik definiert (siehe Kapitel 3.1 Voraussetzungen des Steuerungseditors). Der Ausdruck in einem Aktionszustand ist also der Name einer Regel, die zur Graphersetzung angewendet werden soll und an die Operation ‚*führeRegelAus*(*Regelname*)' weitergegeben

wird. Daher besitzen Aktionen im Steuerungseditor Rückgabewerte vom Typ *boolean* (*true* oder *false*).
Im einfachsten Fall ist der Aktionszustand ein Knoten mit einer eingehenden und einer ausgehenden Kante (siehe Abbildung 24). Bearbeitet der Automat einen Aktionszustand, der wie beschrieben aussieht, so führt er die Aktion aus, die er als Ausdruck beinhaltet (Regelname). Danach wird die Kontrolle an das nächste Steuerungsobjekt, auf den die ausgehende Kante des Aktionszustandes zeigt, automatisch übergeben (Zustandsübergang). Ein so spezifizierter Aktionszustand soll als implizit bezeichnet werden. Folgende Abbildung veranschaulicht diesen Zusammenhang:

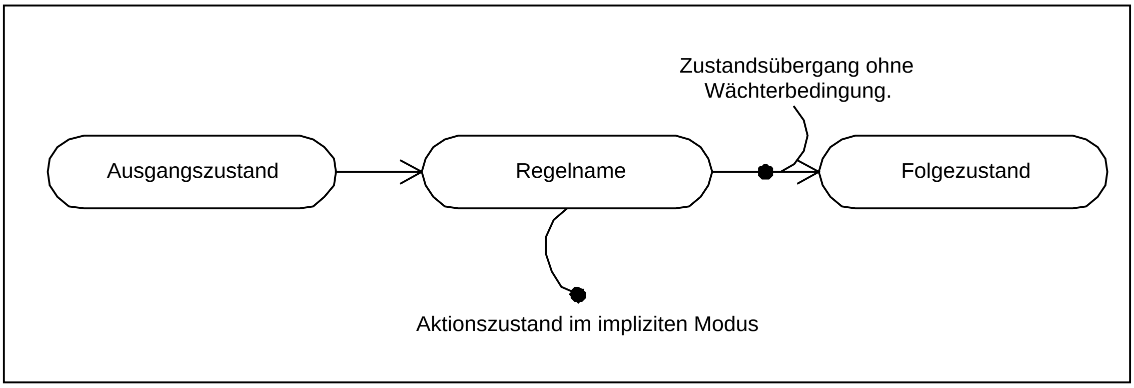


Abbildung 24 Impliziter Modus des Aktionszustandes

Neben dem impliziten Fall wird eine zweite Variante des Aktionszustandes zugelassen, nämlich der boolesche Aktionszustand. In diesem Fall besitzt der Aktionszustand zwei ausgehende Kanten (aber immer noch eine eingehende Kante). Damit die Kantenwahl bei den Zustandsübergängen deterministisch bleibt, müssen die Kanten (also die Zustandsübergänge) boolesche Wächterbedingungen haben, die zu einem Zeitpunkt immer nur einen Zustandsübergang zulassen.
Es sollen folgende Wächterbedingungen für boolesche Aktionszustände definiert sein, um deterministische Zustandsübergänge zu gewährleisten:

1. Die [*false*] Bedingung: Ist der Rückgabewert der spezifizierten Aktion *false*, so bedeutet dies, dass die Aktion nicht erfolgreich ausgeführt wurde. In diesem Fall konnte also kein Ansatz gefunden werden, um eine Graphersetzung durchzuführen.
2. Die [*true*] Bedingung: Ist der Rückgabewert der spezifizierten Aktion *true*, so konnte die Aktion erfolgreich ausgeführt werden, so dass ein Ansatz gefunden wurde und eine Graphersetzung stattgefunden hat.

Der boolesche Aktionszustand ist also in der Lage, in Abhängigkeit von dem Rückgabewert der Aktion eine Verzweigung des Kontrollflusses zu verursachen. Indem jeder der ausgehenden Kanten eine der Wächterbedingungen *true* oder *false* zugewiesen wird, kann eine deterministische Verzweigung gesichert werden. Dieselbe Wächterbedingung darf also nicht zweimal vergeben werden. Welcher Folgezustand nun durch den Automaten weiterbearbeitet wird ist von dem Rückgabewert der Aktion abhängig. Folgende Abbildung zeigt einen booleschen Aktionszustand:

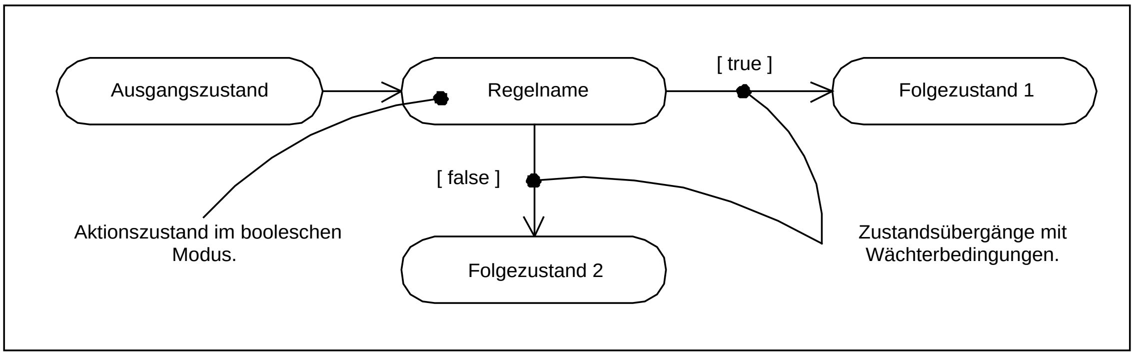


Abbildung 25 Boolescher Modus des Aktionszustandes

Die grafische Darstellung des Aktionszustandes ist wie gehabt eine Pastillenform mit einem Regelnamen als Ausdruck.

Das Verzweigungssymbol

Wie schon aus den Programmschemata bekannt, sind bedingte Verzweigungen fundamentale Kontrollstrukturen. Aktivitätsdiagramme führen zu diesem Zweck ein eigenes Symbol ein, nämlich das Verzweigungssymbol. Das Verzweigungssymbol darf nur eine eingehende Kante haben - somit sind Schlingen verboten - und zwei ausgehende Kanten, die ähnlich wie beim booleschen Aktionszustand Wächterbedingungen besitzen. Wenn der Automat ein Verzweigungssymbol bearbeitet entscheidet er in erster Line, welchen Zustandsübergang (das heißt welchen der beiden ausgehenden Kanten) er annimmt. Diese Entscheidung wiederum hängt von den Wächterbedingungen der Kanten ab. Daher sollen folgende Wächterbedingungen für Verzweigungen definiert werden:

1. Die [*guard*] Bedingung: Diese Bedingung enthält einen booleschen Ausdruck von Regelnamen. Darunter versteht man einen Ausdruck, der aus den booleschen und argumentierten Operationen *UND*, *ODER* und *NICHT* besteht. Als Argumente bekommen diese Operationen Regelnamen, die durch die Operation ‚*istRegelAnwendbar*(*Regelname*)' einen booleschen Rückgabewert erhalten. Ist die Auswertung der so entstehenden Ausdrücke wahr (*true*), so ist der *guard* ebenfalls wahr, sonst falsch (*false*). Hierfür kann in der Abbildung 26 ein Beispiel gefunden werden.
2. Die [*else*] Bedingung: Ist der *guard* wahr, so ist die *else* - Bedingung falsch. Ist der guard falsch, so ist die *else* - Bedingung wahr.

Das Verzweigungssymbol spezifiziert eine bedingte Verzweigung, die in Abhängigkeit von der Anwendbarkeit einer Regel auf den Startgraphen den Kontrollfluss verzweigt:

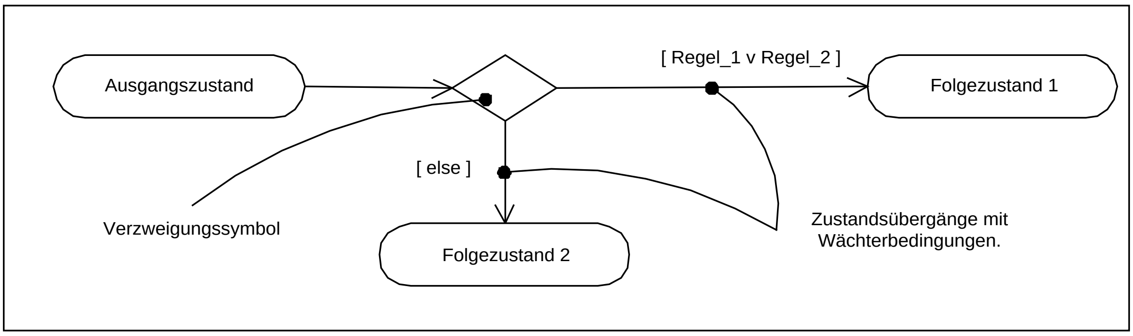


Abbildung 26 Das Verzweigungssymbol

Erklärung der Abbildung 26: Ist der Ausdruck ‚[Regel_1 oder Regel_2]' true, dann wird der Folgezustand eins aktiv, sonst der Folgezustand zwei. Dies entspricht einer ‚if,then,else' – Anweisung , wie sie aus GOTO – Schemata, beziehungsweise While – Schemata bekannt ist (siehe Kapitel 2.2.2):

if (Regel_1 v Regel_2)
then
    Folgezustand_1 ;
else
    Aktionszustand_2 ;

Abbildung 27 Entsprechung einer IF – Anweisung

Nun soll der Unterschied zwischen dem booleschen Aktionszustand und dem Verzweigungssymbol genauer betrachtet werden. Beim booleschen Aktionszustand wird in Abhängigkeit vom Rückgabewert der Aktion (*true* oder *false*) verzweigt. Beim Verzweigungssymbol wird in Abhängigkeit von dem *guard* verzweigt. Im Falle des booleschen Aktionszustandes handelt es sich um eine Nachbedingung (Rückgabewert der Aktion), während es sich beim Verzweigungssymbol um eine Vorbedingung handelt (*guard* – Bedingung). Aus diesem Grund können boolesche Aktionszustände nicht durch Verzweigungssymbole ersetzt werden. Die Abbildung 28 verdeutlicht diesen Zusammenhang grafisch.

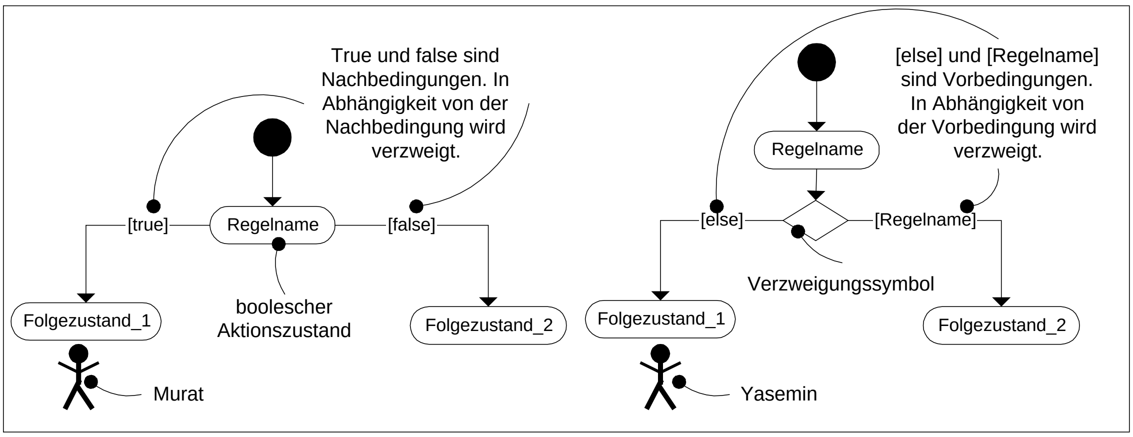


Abbildung 28 Unterschied zwischen Verzweigungssymbol und booleschem Aktionszustand

Fragte man nun *Murat*, aus der Abbildung 28, welche Aussage er treffen würde, so erhielte man folgende Antwort: Ich bin im *Folgezustand_1*, daher war die Regel *Regelname* anwendbar.

Fragte man hingegen *Yasemin*, so erhielte man folgende Antwort: Ich bin im *Folgezustand_1*, aber ich weiß nicht, ob die Regel *Regelname* anwendbar war. Auf Grund dieser Tatsache sind der linke und der rechte Steuerungsgraph in der Abbildung 28 nicht äquivalent. Daher stellen boolesche Aktionszustände eine echte Erweiterung der Kontrollstrukturen im Steuerungseditor dar.

Die symbolische Darstellung des Verzweigungssymbols ist eine Raute (angelehnt an das Aktivitätsdiagramm).

Das Schleifensymbol

Die in Kapitel 2.2.2 vorgestellten WHILE – Schemata sollen als Vorbild dienen, um ein weiteres Steuerungssymbol einzuführen, nämlich das Schleifensymbol (oder auch WHILE – Symbol). Grafisch soll das WHILE - Symbol auch eine Raute sein, im Unterschied aber zum Verzweigungssymbol mit dick gezeichneten Linien dargestellt werden.

Dieses Symbol besitzt zwei ausgehende und eine eingehende Kante. Die ausgehende Kante, die wieder auf das Schleifensymbol zurückführt, erhält als Wächterbedingung den *guard* (siehe Verzweigungssymbol). Die andere ausgehende Kante erhält die *else* - Bedingung (siehe Abbildung 29).

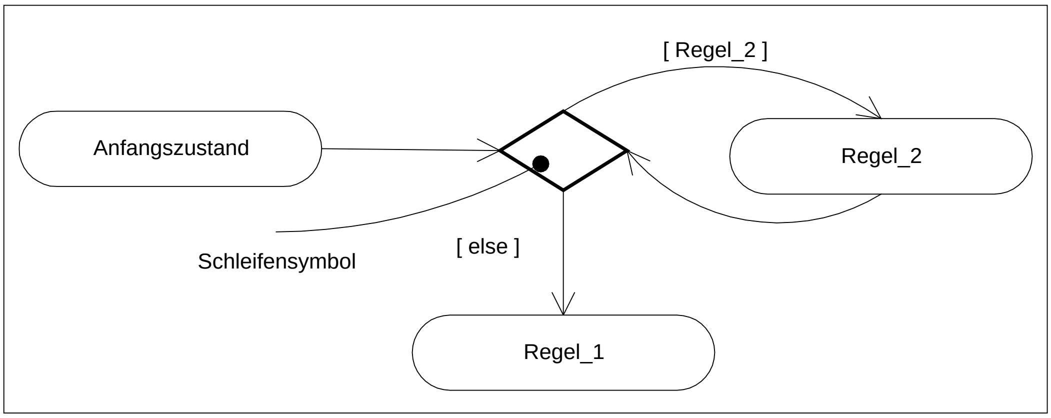


Abbildung 29 Das Schleifensymbol (WHILE -Schleife)

Das Schleifensymbol hat auch eine bedingte Verzweigung, ähnlich wie das Verzweigungssymbol; der einzige Unterschied besteht darin, dass der Zustandsübergang mit der *guard* - Bedingung (in der Abbildung 29 heißt der *guard* ‚*Regel_1*') eine Schleife einleitet. Dies entspricht einer WHILE – Schleife wie sie aus Kapitel 2.2.2 (WHILE - Schemata) bekannt ist. Daher hier die Beschreibung des Schleifensymbols durch eine WHILE Schleife (vergleiche Abbildung 29):

while (Regel_2)
do
    Aktionszustand(Regel_2)
end
    Regel_1

Abbildung 30 Entsprechung einer WHILE - Schleife

Gabelungs- und Synchronisationssymbol

Das Gabelungssymbol leitet in den Aktivitätsdiagrammen nebenläufige Kontrollflüsse ein. Jede von einem Gabelungssymbol ausgehende Kante wird als Gabelung bezeichnet. Ein Steuerungsgraph für Graphersetzungen, der die Abarbeitung nebenläufiger Aktionszustände zulässt, müsste parallele Graphersetzungen zulassen, denn es könnte vorkommen, dass zwei parallel ablaufende Aktionen zum selben Zeitpunkt eine Graphersetzung einleiten. Das Graphersetzungssystem AGG bietet keinen parallelen Graphersetzungsmechanismus (siehe Kapitel 2.3.1).

Aus diesem Grund soll eine andere Semantik diesen Symbolen zugeordnet werden. Das Gabelungssymbol soll aus einer Anzahl (2 oder mehr) von Gabelungen bestehen. Die so entstehenden Teilgraphen werden dann bei der Abarbeitung durch den Automaten in eine ODER – Beziehung gesetzt. Dies führt dazu, dass eine dieser Gabelungen zufällig zur Graphersetzung gewählt wird. Die Gabelungen werden durchnummeriert (siehe Abbildung 31).

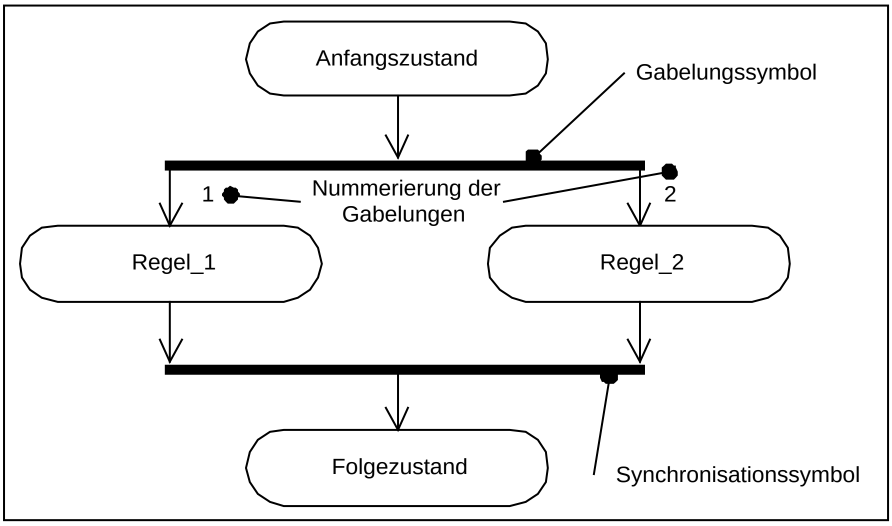


Abbildung 31 Das Gabelungssymbol

In der Abbildung 31 findet man das Beispiel eines Gabelungssymbols. Bearbeitet der Automat dieses Symbol, so führt er entweder die Regel_1 oder die Regel_2 aus (nicht deterministische Auswahl).

Das Synchronisationssymbol wird nur noch als Abschlussbalken für die durch das Gabelungssymbol erzeugten Gabelungen betrachtet.

Wird ein Gabelungssymbol erzeugt, so wird, vom Steuerungseditor automatisch ausgeführt, auch das Synchronisationssymbol eingefügt. Somit lassen sich diese Steuerungssymbole immer nur als Paar einfügen. Im Unterschied zu den anderen Steuerungssymbolen besitzt das Gabelungssymbol zwei oder mehr ausgehende Kanten ohne Wächterbedingungen. Die ausgehenden Kanten sind lediglich durchnummeriert. Das Synchronisationssymbol hat zwei oder mehr eingehende Kanten, im Gegensatz zu den anderen Steuerungsobjekten, die höchstens zwei (das Schleifensymbol) eingehende Kanten haben können.

Die grafische Notation ist auch bei diesen Steuerungssymbolen an das Aktivitätsdiagramm angelehnt.

Hiermit soll die Spezifizierung, der am Steuerungseditor beteiligten Steuerungssymbolen, abgeschlossen sein. Folgender Unterabschnitt bezieht sich auf eine Grundeigenschaft, die alle Steuerungssymbole besitzen, nämlich die Möglichkeit *aktivierte* Kanten zu benennen.

3.1.1.1 Aktivierte Kanten

Bis jetzt wurde nur gesagt, dass neue Steuerungssymbole immer zwischen zwei schon vorhandenen Symbolen erzeugt werden können. Was passiert jedoch, wenn ein Steuerungssymbol zwei ausgehende Kanten hat, wie zum Beispiel das Verzweigungssymbol (siehe Abbildung 26). In diesem Fall muss das neu einzufügende Steuerungssymbol entweder zwischen dem Verzweigungssymbol und dem Folgesymbol der einen Kante oder zwischen dem Verzweigungssymbol und dem Folgeknoten der anderen Kante eingefügt werden können.
Aus diesem Grund soll es möglich sein, festzulegen, welche ausgehende Kante eines Steuerungssymbols *aktiviert* ist. Eine Kante (Zustandsübergang) besitzt ein Quellsteuerungssymbol und ein Zielsteuerungssymbol. Ist eine Kante durch den Benutzer *aktiviert* worden, so wird zwischen Quelle und Ziel dieser *aktivierten* Kante das gewünschte Steuerungssymbol eingefügt. Nur ausgehende Kanten dürfen *aktiviert* werden. Besitzt ein Steuerungssymbol nur eine ausgehende Kante, wie zum Beispiel das Startsymbol, so ist sie automatisch *aktiviert.* Außerdem kann immer nur eine Kante *aktiviert* sein. Diese Funktionalität bieten alle Steuerungssymbole mit zwei oder mehr ausgehenden Kanten an (siehe Kapitel 3.1.4).
Nachdem die Steuerungsobjekte vorgestellt wurden, die der Steuerungseditor kennt, soll nun im folgenden Kapitel beschrieben werden, wie der Steuerungseditor dafür sorgt, dass die Steuerungsgraphen syntaktisch korrekt erzeugt werden.

3.1.2 Syntaxsteuerung durch den Steuerungseditor

Unter Syntaxsteuerung soll verstanden werden, dass der Benutzer des Steuerungseditors bei der Erstellung von Steuerungsgraphen darin unterstützt wird, nur syntaktisch korrekte Steuerungsgraphen zu erzeugen. Einerseits dürfen also keine inkonsistenten Steuerungssymbole erzeugt werden, wie zum Beispiel ein Aktionszustand ohne definierte Aktion oder ein Zustandsübergang ohne Folgeobjekt, andererseits darf das Löschen dieser Symbole nicht zu inkonsistenten Steuerungsgraphen führen. Durch diese Forderung erhöht sich die Benutzerfreundlichkeit des Steuerungseditors und verringert sich die Fehlerwahrscheinlichkeit des Benutzers.
Um dieser Forderung nachzukommen sind im Kapitel 3.2 die Einfüge- und Löschoperationen für jedes Steuerungssymbol syntaxgerecht spezifiziert. Bei dieser Spezifizierung sind folgende Regeln beachtet worden:

1. Der Initiale Steuerungsgraph besteht aus genau einem Startsymbol, das auf ein Stopsymbol zeigt (siehe Abbildung 51).
2. Ein neues Steuerungssymbol kann immer nur zwischen zwei Steuerungssymbolen eingefügt werden, die durch eine Kante verbunden sind.
3. Zustandsübergänge können nicht explizit erzeugt oder gelöscht werden. Sie entstehen vielmehr dadurch, dass Steuerungssymbole erzeugt werden und ein- und ausgehende Kanten durch den Steuerungseditor eigenständig eingefügt oder neu verbunden werden. Wird ein Steuerungssymbol gelöscht, so werden

auch die beteiligten und nicht mehr benötigten Kanten gelöscht. Übrig gebliebe Kanten werden neu verbunden.

Der Steuerungseditor soll überdies bei der Erzeugung der Steuerungssymbole dafür sorgen, dass korrekte Wächterbedingungen einerseits (sofern es sich um das Schleifen- oder Verzweigungssymbol handelt) und Aktionsausdrücke andererseits gewählt werden. Als Werkzeug zu diesem Ziel kann man sich Dialoge vorstellen, die es fehlerfrei ermöglichen, per Mausklick solche Ausdrücke zu spezifizieren (siehe Kapitel 3.1.4).

Der Steuerungseditor übernimmt nicht nur die Syntaxüberprüfung des Steuerungsgraphen, indem der Editor von vornherein die Steuerungsobjekte syntaktisch korrekt erzeugen und löschen lässt, sondern er unterstützt auch den Benutzer in der Platzierung der Steuerungsobjekte, aus denen der Steuerungsgraph besteht. Diese Funktion des Editors wird im folgenden Unterkapitel beschrieben.

3.1.3 Layoutunterstützung durch den Steuerungseditor

Die Anzahl der eingehenden und ausgehenden Kanten eines Steuerungssymbols ist maßgeblich dafür verantwortlich, wie lesbar der entstehende Steuerungsgraph wird. Je höher die Anzahl ist, um so schlechter wird die Lesbarkeit. Da jedoch folgende Punkte für den Steuerungsgraphen gelten kann eine starke Vernetzung von Steuerungssymbolen ausgeschlossen werden:

1. Die Steuerungssymbole haben immer nur eine eingehende Kante. Die einzigen Ausnahmen bilden das WHILE – Symbol, das eine Schlinge besitzt und daher eine zweite eingehende Kante hat, und das Startsymbol, das keine eingehende Kante besitzt.
2. Alle Steuerungssymbole besitzen mindestens eine, aber höchstens zwei ausgehende Kanten. Die einzige Ausnahme stellt das Stopsymbol dar, das keine ausgehende Kanten besitzt.

Diese zwei Punkte machen deutlich, dass Schleifen nur an vordefinierten Stellen im Graphen erzeugt werden können, nämlich beim Schleifensymbol. Alle anderen Steuerungssymbole haben immer nur höchstens eine eingehende Kante, weshalb sie niemals Schleifen erzeugen können.

Beim Gabelungs- und Synchronisationssymbol verhält sich dieser Zusammenhang etwas anders. Dadurch, dass zwei oder mehr Kanten durch das Gabelungssymbol erzeugt werden können, die jeweils einen unterschiedlichen Teilgraphen einleiten, muss der Benutzer durch Verwendung des Steuerungseditors nachträgliche Verbesserungen machen, wie zum Beispiel das Vergrößern des Gabelungsbalkens auf die Breite der eingeleiteten Teilgraphen.

Ein weiterer Punkt ist die Platzierung der Steuerungssymbole im Steuerungseditor, wenn sie neu erzeugt werden. Es sollte erreicht werden, dass diese Symbole nicht kreuz und quer im Steuerungseditor platziert werden.

Um das Bild des Steuerungsgraphen nachträglich zu verändern sollen die Steuerungssymbole mit der Maus an der Oberfläche verschoben werden dürfen. Außerdem sollen die Steuerungssymbole in ihrer Größe verändert werden können.

Die wichtigste Layoutunterstützung des Steuerungseditors, nämlich die automatische Platzierung von Steuerungssymbolen wird im folgenden Unterkapitel beschrieben.

3.1.3.1 Automatisches Platzieren von Steuerungssymbolen

Alle Steuerungssymbole sind in quadratische Grundformen eingebettet, was die Platzierung in den Editierbereich des Steuerungseditors vereinfacht. Der Editierbereich selbst unterliegt einem Koordinatensystem, das in der oberen, linken Ecke ihren Ursprung hat. Nun kann jede Seite der quadratischen Grundform durch eine Himmelsrichtung beschrieben werden. Die obere Seite des Steuerungssymbols (aus dem Blickwinkel des Betrachters gesehen) heißt Norden, die untere Seite heißt Süden, die rechte Seite heißt Osten und die linke Seite heißt Westen.

Alle Küsten (also der Norden, Süden, Osten und Westen) haben sogenannte Zugangsbehälter (Accesscontainer – siehe Abbildung 32). Unter einem Zugangsbehälter soll eine Datenstruktur verstanden werden, die es Kanten erlaubt, durch Vergabe von grafischen Zugangspunkten (also Punkten im Koordinatensystem) an Kanten, grafisch an ihre Quell- und Zielobjekte gebunden zu werden. Kanten sind also die Objekte, die von Steuerungsobjekten solche Zugangspunkte anfordern. Bei der Anforderung dieser Zugangspunkte muss dem Steuerungsobjekt, von dem ein Zugangspunkt angefordert wird, eine Küste mitgegeben werden, von der ein Zugangspunkt angefordert wird. Wenn eine Kante erzeugt wird, dann müssen zwei Zugangspunkte angefordert werden, einmal vom Quellobjekt und einmal vom Zielobjekt.

Dasselbe gilt für die Freigabe solcher Zugangspunkte, wenn Kanten gelöscht werden. In diesem Fall muss der Zugangspunkt vom Quell- und Zielobjekt wieder freigegeben werden (wieder unter Angabe der Küstennamen).

Die Anzahl der Zugangspunkte (Accesspoints), die ein Zugangsbehälter vergeben kann, ist schon bei der Erzeugung eines Zugangsbehälters festzulegen.

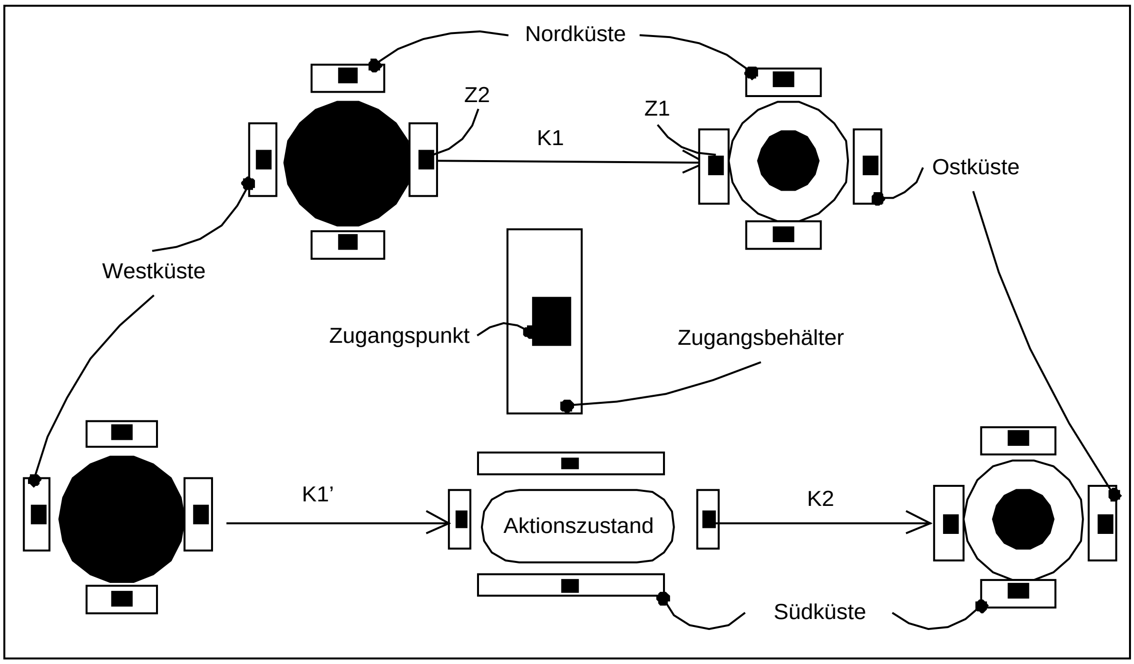


Abbildung 32 Zugangsbehälter und Zugangspunnkte

Fallbeispiel zur Verdeutlichung (siehe Abbildung 32): Es soll ein Aktionszustand eingefügt werden. Der bisher entstandene Graph besteht aus einem Anfangszustand und einem Endzustand, die durch eine Kante verbunden sind. Der neu einzufügende Aktionszustand wird zwischen dem End- und dem Anfangszustand eingefügt. Nun werden, unter der Voraussetzung, dass alle Steuerungsobjekte nur Zugangsbehälter mit nur einem Zugangspunkt besitzen, folgende Schritte vom Steuerungseditor ausgeführt:

1. Die Kante, die den End- und Anfangszustand verbindet, gibt die Zugangspunkte Z1 und Z2 frei.
2. Die hängende Kante K1 wird kopiert und anschließend gelöscht.
3. Dann wird der Aktionszustand eingefügt.
4. Die kopierte Kante K1' wird zwischen Anfangszustand und Aktionszustand eingefügt. Dabei fordert die Kante Zugangspunkte von der Ostküste des Anfangszustands und von der Westküste des Aktionszustands an. Nun weiß die Kante, wie sie sich grafisch einordnen soll.
5. Eine zweite Kante K2 wird zwischen Aktionszustand und Endzustand erzeugt. Analog zum Schritt 4 werden Zugangspunkte angefordert, diesmal jedoch von der Ostküste des Aktionssymbols und von der Westküste des Endzustands.

Kanten besitzen keine Zugangsbehälter, aber sie stehen ständig mit den Küsten der einzelnen Steuerungselemente in Wechselbeziehung.

Jetzt muss nur noch beschrieben werden, wie der Steuerungseditor entscheiden soll, welche Küsten bei der Kanteneinfügung benutzt werden sollen. Dazu soll folgende Voraussetzung gemacht werden, die beim Arbeiten mit dem Steuerungseditor maßgeblich ist:

1. Der Benutzer entwirft Steuerungsgraphen von links nach rechts (Editierrichtung Osten) und von oben nach unten (Editierrichtung Süden). Beim Platzieren der Steuerungsobjekte berücksichtigt der Steuerungseditor als erstes diese Editierrichtungen.

Die Editierrichtung ist maßgeblich dafür verantwortlich, wohin ein neues Steuerungsobjekt bezüglich des Ausgangszustandes platziert wird. Der neu eingefügte Aktionszustand aus Abbildung 32 wird beispielsweise rechts neben dem Ausgangszustand (in der Abbildung 32 ist es das Startsymbol) platziert. Aufschluss über die Editierrichtung gibt die Austrittskante. Wenn zum Beispiel eine Kante aus der Ostküste austritt (wie Kante K1), dann nimmt man an, dass von links nach rechts editiert wird. Daher wird die Westküste des Folgezustands zur Zugangspunktanforderung durch den Editor herangezogen. Folgende Tabelle soll Aufschluss darüber geben, wie der Editor diesbezüglich entscheidet:

Austrittsküste der Kante	**Zielküste des Folgezustands**	**Editierrichtung (von / nach / Richtung)**
Westküste	Ostküste	rechts / links / Westen
Ostküste	Westküste	links / rechts / Osten
Nordküste	Südküste	oben / unten / Süden
Südküste	Nordküste	unten / oben / Norden

Tabelle 1 Küstenwahl bei der Zugangspunktanforderung

Dieses Konzept erlaubt es dem Steuerungseditor, die Platzierung der Steuerungssymbole und der sie verbindenden Kanten zu übernehmen.

Der Editor erlaubt es, die Editierrichtung nachträglich zu ändern, vorausgesetzt, die Kante, die in eine andere Editierrichtung verschoben werden soll, zeigt auf einen Stopzustand.

Alle bisher beschriebenen Anforderungen des Steuerungseditors spiegeln sich in den Operationen wieder, die Steuerungsobjekte anbieten. Diese Operationen sind in Kontextmenüs der Steuerungsobjekte wiederzufinden. Eine Beschreibung dieser Kontextmenüs geschieht im folgenden Unterkapitel.

3.1.4 Kontextmenüunterstützung des Steuerungseditors

Unter einem Kontextmenü versteht man eine Sammlung von Operationen, die alle zu einem Objekt gehören. Man kann das Kontextmenü aufrufen, indem man mit der Maus auf ein Steuerungssymbol geht und die rechte Maustaste drückt (und dabei gedrückt lässt). Danach öffnet sich ein kleines Fenster, das die entsprechenden Operationen in Form eines Menüs auflistet. Die spezifizierten Steuerungssymbole sind in diesem Zusammenhang auch als Objekte aufzufassen, die ein solches Set von Operationen besitzen. In diesem Kontextmenü findet man zum Beispiel Einfüge- und Löschoperationen und Layoutoperationen, wie das Verändern der Symbolgröße.

Um einen Überblick über die Funktionalität des Steuerungseditors zu bekommen, wird im folgenden das Kontextmenü des Aktionszustandes beschrieben. Einige Operationen, wie zum Beispiel das Vergrößern eines Steuerungssymbols, finden sich in allen Kontextmenüs wieder, andere hingegen, wie zum Beispiel das Wechseln zwischen impliziter und boolescher Darstellung des Aktionszustandes, sind objektspezifisch.

Grundstruktur des Kontextmenüs

Folgende Abbildung verdeutlicht, dass das Kontextmenü prinzipiell aus drei Abschnitten besteht:

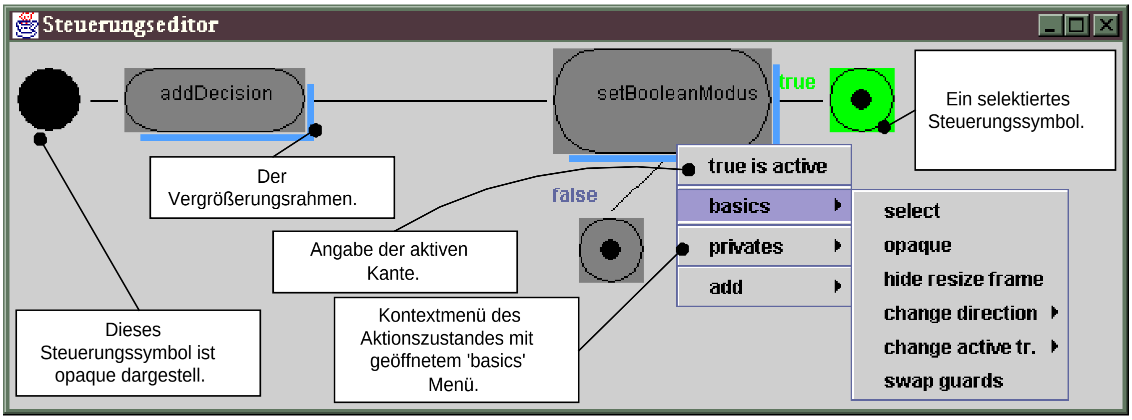


Abbildung 33 Aufbau eines Kontextmenüs

Das Kontextmenü besteht aus drei Abschnitten. Im Kontextmenü ganz oben ist die aktivierte Kante angegeben. Danach beginnt der erste Abschnitt. Die *‚basics'* umfassen Operationen, mit denen sich das Aussehen, die Editierrichtung und die Wächterbedingung des Steuerungssymbols bearbeiten lässt. Die Optionen im einzelnen sind folgende:

1. *‚resize'*: Wie schon erwähnt sollen die Steuerungssymbole in ihrer Größe veränderbar sein. Bei Wahl dieser Option wird ein Rahmen um das Steuerungsobjekt sichtbar. Dieser Rahmen erlaubt es, das Objekt in seiner Größe zu verändern, indem mit der Maus an dem Rahmen gezogen wird. Ist

der Rahmen sichtbar, dann kann mit der Option *,hide resize frame*', die jetzt anstelle der Option *,resize*' zu sehen ist, der Rahmen wieder versteckt werden.

2. *,select'*: Mit dieser Option kann man Steuerungsobjekte auswählen. Sie werden dann mit anderer Hintergrundfarbe dargestellt. Ist ein Steuerungsobjekt selektiert, dann erscheint anstelle von *,select'* ein *,deSelect'* im Kontextmenü. Damit lässt sich dann die Standardhintergrundfarbe des Steuerungssymbols wiederherstellen.
3. *,opaque'*: Mit dieser Option kann der Hintergrund eines Steuerungsobjektes versteckt werden. Dann werden nur noch Beschriftung und grafische Linienführung des Steuerungsobjektes dargestellt. Möchte man diese Darstellung rückgängig machen, wählt man die Option *,deOpaque*', die dann im Menu anstelle von *,opaque*' ersheint.
4. *,change direction'*: Zum Ändern der Editierrichtung bietet dieses Menü die Anzahl der Küstennamen (siehe Abbildung 32) an, in die die aktivierte ausgehende Kante verschoben werden kann (siehe Kapitel 3.1.3).
5. *,change aktiv transition'*: Dieses Menü wird nur dann angeboten, wenn das Steuerungsobjekt mehr als eine ausgehende Kante besitzt. Es kann damit definiert werden, welche ausgehende Kante aktiv sein soll (siehe Kapitel 3.1.1.1). Gibt es mehr als nur eine ausgehende Kante, so werden die Wächterbedingungen dieser Kanten als Optionsnamen angeboten. Beim booleschen Aktionszustand wären dann die Optionen *,true'* und *,false'* angeboten. Je nachdem welche dieser Optionen gewählt wird, ändert sich dann die *aktivierte* ausgehende Kante. Das Gabelungssymbol hat auch mehrere ausgehende Kanten (mindestens zwei). Im Gegensatz zu den anderen Steuerungsobjekten haben diese Kanten keine Wächterbedingung. Zur Unterscheidung voneinander sind sie durchnummeriert. Daher erscheinen dann die Nummern der einzelnen Kanten im Menü *,change aktiv transition'*.
6. *,swap guards'*: Diese Option wird nur dann angeboten, wenn das Steuerungsobjekt exakt zwei ausgehende Kanten hat. Bei Wahl dieser Option werden die Wächterbedingungen dieser Kanten vertauscht.

Diese Optionen sind für alle Steuerungsobjekte gleich, im Gegensatz zu den Optionen im Menü ,privates' (der zweite Abschnitt des Kontextmenüs). Folgende Abbildung zeigt die privaten Optionen des Aktionszustandes:

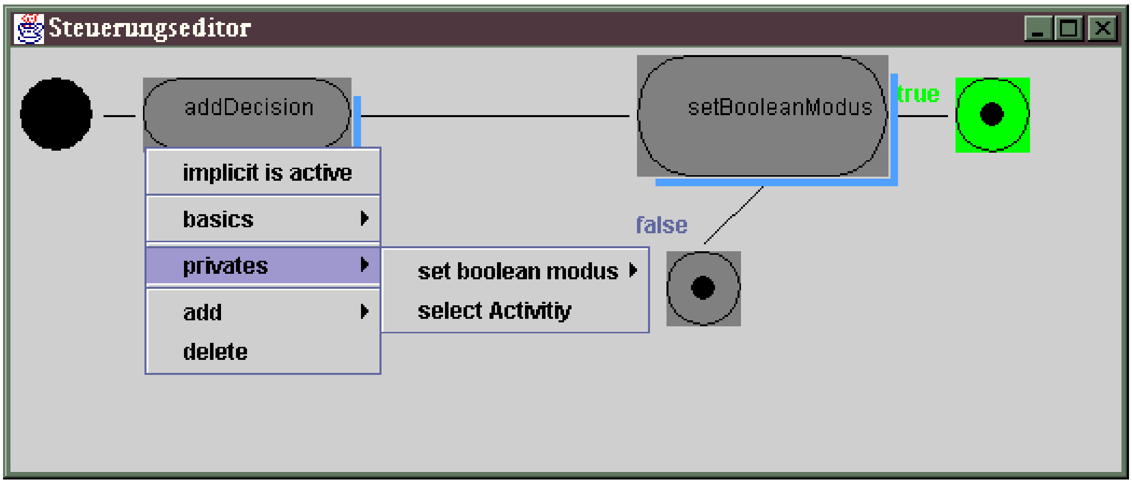


Abbildung 34 Private Optionen des Aktionszustandes

Hier sammeln sich alle steuerungssymbolspezifischen Optionen. Für den Aktionszustand sind es also folgende:

1. *‚set boolean modus'*: Mit diesem Menü kann man einen im impliziten Modus befindlichen Aktionszustand in einen booleschen Aktionszustand überführen. Soll dieser Vorgang rückgängig gemacht werden, kann das Menü *‚set implizit modus'* angewendet werden, welches dann anstelle des *‚set boolean modus'* erscheint. Beim Wechsel vom impliziten Modus in den booleschen Modus wird eine neu ausgehende Kante erzeugt, die sofort auf das Stopsymbol zeigt. Durch die Option *‚true*', beziehungsweise *‚false*' des Menüs *‚set boolean modus'* kann dann der bestehenden Kante eine Wächterbedingung gegeben werden. Die andere ausgehende Kante erhält dann die nicht selektierte Wächterbedingung. Befindet sich der Aktionszustand im booleschen Modus, so wird beim Wechsel zum impliziten Modus eine ausgehende Kante gelöscht. Um angeben zu können, welche Kante gelöscht werden soll, wählt man dann wieder eines der Optionen *‚true'* oder *‚false'* (diesmal vom Menü *‚set implizit modus'*) aus. Um in den impliziten Modus wechseln zu können, muss mindestens eine der ausgehenden Kanten auf das Stopsymbol zeigen. Zeigt nur eine der ausgehenden Kanten auf das Stopsymbol, so wird im Menü *‚set implizit modus'* auch nur eine Option angeboten; In Abhängigkeit davon ob die Wächterbedingung dieser ausgehenden Kante *true* oder *false* ist.
2. *‚select Activity'*: Mit dieser Option kann die Aktion eines Aktionszustandes verändert werden. Bei Wahl dieser Option öffnet sich ein Dialog (siehe Abbildung 35), über den man einen Regelnamen auswählen kann, der dann als Beschriftung im Aktionszustand erscheint. Wird ein Aktionszustand neu erzeugt, erscheint ebenfalls dieser Dialog.

Folgende Abbildung zeigt den Dialog, der die Auswahl einer Aktion aus einer Liste ermöglicht:

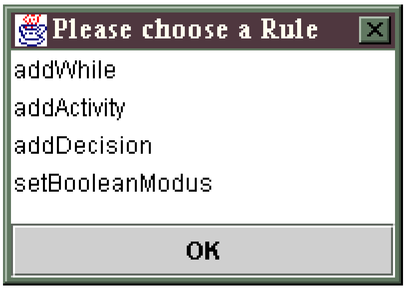


Abbildung 35 Dialog zur Auswahl einer Aktion

Der letzte Abschnitt des Kontextmenüs besteht, wie die Abbildung 33 zeigt, aus dem Menü *‚add'* und der *‚delete'* Option:

1. *‚add'*: Dieses Menü bietet eine Auswahl der Steuerungsobjekte an, die zwischen dem Quellobjekt und dem Zielobjekt der aktivierten ausgehenden Kante eingefügt werden soll. Das Stopsymbol bietet keine *‚add'*- Option an, da es ein Terminierungspunkt des Automaten repräsentiert.
2. *‚delete'*: Löscht einen Aktionszustand. Man kann einen Aktionszustand nur dann löschen, wenn er sich im impliziten Modus befindet. Daher wird diese Option im booleschen Modus des Aktionszustandes nicht angeboten. Start- und Stopsymbol können nicht gelöscht werden. Das Stopsymbol wird automatisch durch den Steuerungseditor gelöscht und das Startsymbol ist immer im Steuerungsgraphen enthalten, weshalb es nicht gelöscht werden darf.

Im Anhang befinden sich die Spezifikationen des Kontextmenüs der anderen Steuerungssymbole. Dabei soll aber nur auf die Unterschiede eingegangen werden, das heißt es werden nur die privaten Optionen aufgelistet. Der sonstige Aufbau des Kontextmenüs bleibt gleich.

Das Erscheinen der Löschoption im Kontextmenü ist davon abhängig, ob das Steuerungsobjekt gelöscht werden kann oder nicht. Es bedarf daher einer genaueren Spezifikation, wie das Löschen von Steuerungsobjekten definiert ist. Diese Spezifikation der Lösch- und Einfügeoperationen findet im folgenden Unterkapitel (Kapitel 3.2) statt. Der Vorteil einer solchen Spezifikation ist, dass somit alle erzeugbaren Steuerungsgraphen formal beschrieben sind, die durch den Editor spezifizierbar sind.

3.2 Graphgrammatische Spezifizierung des Steuerungsgraphen

Der Benutzer des Steuerungseditors erzeugt Steuerungsgraphen. Im Kapitel 3.1.4 ist beschrieben, wie diese Steuerungsgraphen, durch Anwendung der Kontextmenüs, erzeugt werden können. Um erkennen zu können, wie diese Steuerungsgraphen im allgemeinen aussehen, macht es Sinn, den Steuerungsgraphen formal zu beschreiben. Es soll zu diesem Zweck ein Graphersetzungssystem beschrieben werden, das Graphstrukturen des Steuerungsgraphen beschreibt. Um es nochmal zu betonen: Diese formale Spezifikation beschreibt nicht nur die Graphstrukturen des Steuerungsgraphen, sondern auch die Kontrollstrukturen, die für die Graphersetzung durch den Automaten verwendet werden.

Das Graphersetzungssystem selbst soll in der im AGG üblichen Form spezifiziert werden (siehe dazu Kapitel 2.1). Um Gruppen von Graphgrammatikregeln zusammenfassen zu können, werden zwei zusätzliche Konzepte, die in AGG fehlen aber in PROGRES [Schürr98] realisiert sind, eingeführt.

Zum einen sollen Metatypen [Gruner] eingeführt werden. Sie erlauben es, Knoten mit unterschiedlichen Knotentypen einem übergeordneten Knotentyp (Metatyp) zuzuweisen. Dadurch wird es möglich, Morphismen zwischen gleichen Metatypen zu bilden. Aufgrund dieser Tatsache verringert sich die Anzahl der nötigen Graphgrammatikregeln zur Beschreibung des Steuerungsgraphen stark. Knotentypen werden durch eine ‚ist ein' – Relation mit Metatypen verbunden. Die Abbildung 36 zeigt eine solche Metatypisierung für die am Steuerungseditor beteiligten Steuerungsobjekte. PROGRES [Schürr] kennt aber keine Metatypen für Kanten. Daher sollen Metatypen für Kanten, analog zu den Metatypen von Knoten, eingeführt werden (siehe Abbildung 36).

Zum anderen wird das *redirect* von PROGRES eingeführt. Das *redirect* ermöglicht es, hängende Kanten, das sind Kanten mit fehlendem Ziel- oder Quellknoten, auf einen anderen Knoten umzuhängen. Während solche Kanten beim Graphersetzungssystem AGG gelöscht werden, was den Verlust von Information des Graphen zur Folge hat, kann in PROGRES diese Kante durch *redirect* geschützt werden. Das *redirect* ist durch eine durchgezogene Linie repräsentiert, die von der linken Regelseite zur rechten Regelseite führt und unterscheidet sich damit grafisch von einem Morphismus, der gepunktet dargestellt wird. Das *redirect* darf nur zwischen Kanten gleichen Typs (oder Metatyps) angewendet werden.

Als erstes werden die Metatypen bekannt gegeben. Es gibt Metatypen für Kanten und für Knoten. Die Grundtypen, aus denen sich diese Metatypen zusammensetzen, entsprechen den Steuerungssymbolen des Steuerungseditors. Siehe dazu folgende Abbildung:

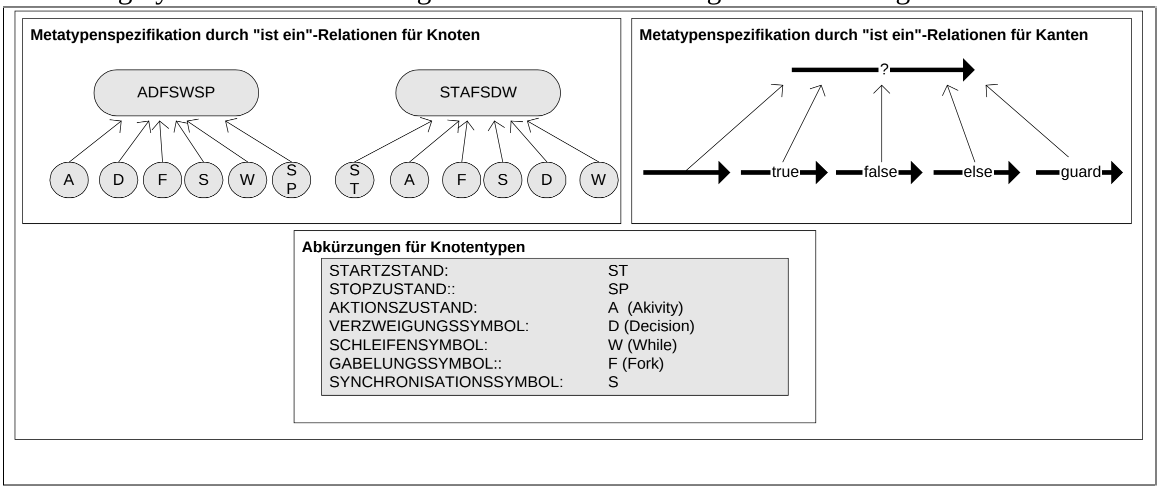


Abbildung 36 Metatypen

Aus der Abbildung 36 ist zu erkennen, dass es zwei Metaknotentypen für Knotentypen gibt, nämlich *ADFSWSP* und *STAFSDW*. Die zu diesen Metatypen gehörenden Knotentypen stehen in einer „ist ein" – Relation mit den Metatypen (repräsentiert durch die Pfeilrichtung von Knotentyp zu Metaknotentyp). Der Metatyp kann als ein Knotentyp interpretiert werden, der ihn definiert. In Analogie dazu sind die Metakantentypen definiert.

Aufgrund der hohen Anzahl der Graphgrammatikregeln, die den Steuerungsgraphen beschreiben, sollen im folgenden nur zwei Regeln näher beschrieben werden. Die restlichen

Regeln können dann analog interpretiert werden und befinden sich im Anhang (siehe Kapitel 6.2) zur Kenntnisnahme. Um das Einfügen und das Löschen zu beschreiben werden nachfolgend das Einfügen eines impliziten und das Löschen eines booleschen Aktionszustands beschrieben.

Einfügen eines impliziten Aktionszustands in einen Steuerungsgraphen

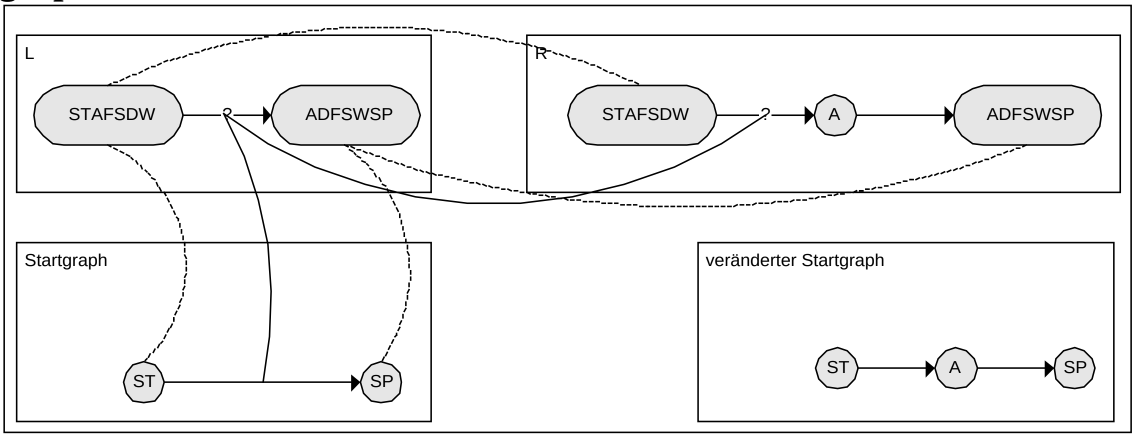


Abbildung 37 Einfügen eines impliziten Aktionszustandes

Auf der linken Regelseite in der Abbildung 37 erkennt man einen Graphen, der aus zwei Metaknotentypen und einem sie verbindenden Metakantentyp besteht. Der Metatyp *STAFSDW* kann als ein Aktionszustand, Startzustand, usw. interpretiert werden (siehe Abbildung 36). Interpretiert man diesen Metatypen als ein Verzweigungssymbol, so wäre die austretende Kante entweder eine *else* – Kante oder eine *guard* – Kante. Um diese Flexibilität der Kanteninterpretation in einem Graphen zu erreichen wurden Metakantentypen eingeführt. Die Kante mit der Beschriftung ‚?' repräsentiert einen solchen Metakantentyp, der auf den Metatyp ADFSWSP zeigt. Die Unterscheidung zwischen den Metaknotentyp ADFSWSP und *STAFSDW* musste deshalb getroffen werden, weil einerseits eine Kante niemals aus einem Stopsymbol austreten kann und andererseits eine Kante niemals in einem Startsymbol eintreten kann (siehe Kapitel 3.1.1). Daher wurde das Stopsymbol nicht dem Metatyp *STAFSDW* zugeordnet; Entsprechend wurde das Startsymbol nicht dem Metatyp *ADFSWSP* zugeordnet.

Auf der rechten Regelseite in der Abbildung 37 sieht man wieder die drei Metatypen aus der linken Regelseite. Die zwei Metaknotentypen sind durch Morphismusabbildungen von der linken Regelseite auf die rechte Regelseite gebunden. Der Metakantentyp auf der linken Regelseite wird mit dem Aktionssymbol auf der rechten Regelseite verbunden (redirect von PROGRES). Da das Aktionssymbol und die austretende Kante auf der rechten Regelseite neu erscheinen, werden die Objekte bei einer Graphersetzung neu erzeugt, wie es der veränderte Startgraph aus der Abbildung 37 zeigt. Die eingefügten Knoten- oder Kantentypen sind keine Metatypen. Würden im Startgraphen Metatypen erzeugt, könnten inkonsistente Steuerungsgraphen entstehen.

Die Abbildung 37 zeigt, dass die Morphismen, von der linken Regelseite auf den Startgraphen, Metatypen auf die ihnen zugrunde liegenden Grundtypen abbilden. Zum Beispiel wird der Metatyp *STAFSDW* auf das Startsymbol im Startgraphen abgebildet. Diese Abbildungsmöglichkeit wird erst durch die Einführung der Metatypen möglich. Dasselbe gilt

für das *redirect*, wo auch Metakantentypen auf die ihnen zugrundeliegenden Kantentypen abgebildet werden.
Der folgende Teil zeigt das Löschen eines booleschen Aktionszustands.

Löschen eines booleschen Aktionszustands

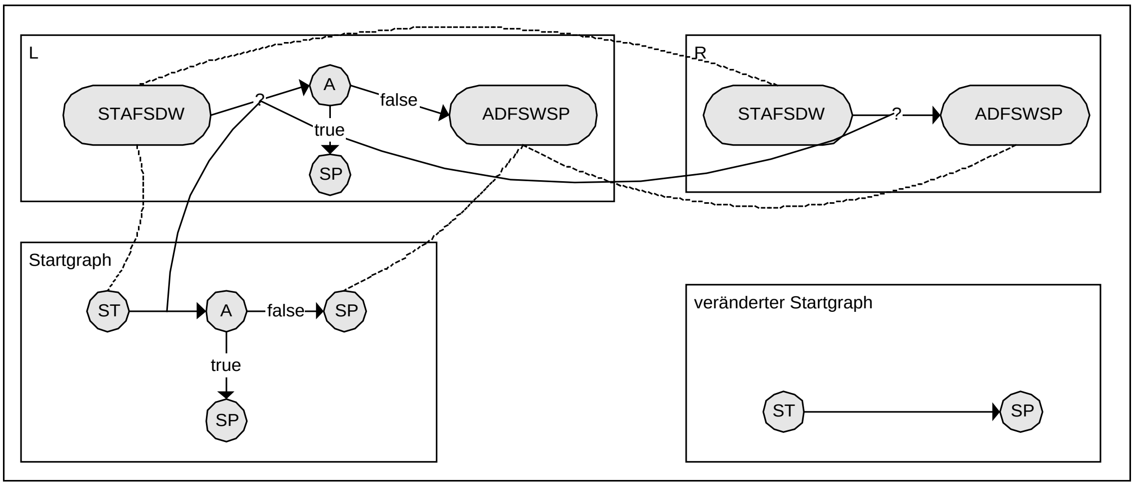


Abbildung 38 Löschen eines booleschen Aktionszustands

Die Abbildung 38 zeigt die graphgrammatische Spezifikation des Löschens eines booleschen Aktionszustands. Alle Objekte, die nicht durch einen Morphismus oder *redirect* zwischen der linken Regelseite und der rechten Regelseite gebunden sind, werden bei der Graphersetzung im Startgraphen gelöscht (siehe *veränderter Startgraph* in Abbildung 38).
Interpretation dieser Löschregel hinsichtlich der Kontextmenüoption *'delete'* (siehe Kapitel 3.1.4): Ein boolescher Aktionszustand kann gelöscht werden, sobald mindestens eine ausgehende Kante auf das Stopsymbol zeigt. In dieser Regel ist es die *true* – Kante. Würde nur die *false* – Kante auf ein Stopsymbol zeigen, könnte diese Regel den boolesche Aktionszustand dann nicht löschen. Für den zweiten Fall muss demnach eine neue Regel spezifiziert werden (siehe Kapitel 6.2). Wenn eine Löschregel für ein Steuerungsobjekt anwendbar ist wird die *'delete'* – Option im Kontextmenü angeboten, sonst nicht.
Es wird deutlich, dass eine ganze Reihe von Graphgrammatikregeln erforderlich ist, um den kompletten Steuerungsgraphen zu beschreiben. Im Anhang 6.2 kann man diese Regeln nachlesen. Die dort beschriebenen Regeln sind mit den in diesem Kapitel gewonnenen Erkenntnissen leicht zu verstehen.
Im folgenden Unterkapitel werden Kontrollstrukturen von PROGRES [Schürr98] und Grrr [King/Rodgers] vorgestellt, um einen Einblick darüber zu geben, wie andere Graphersetzungssysteme Kontrollstrukturen realisieren.

3.3 Kontrollstrukturen in PROGRES und Grrr

Bislang ist beschrieben worden, wie der Steuerungseditor die Kontrollmöglichkeiten von AGG erweitert. AGG arbeitet intern mit einer fest implementierten Transformationsschleife. In der Treeview des AGG – Fensters (siehe Abbildung 6) ist die Reihenfolge der Produktionsregeln (Graphersetzungsregeln) zu sehen, in der sie vom Transformator zur Graphersetzung angewendet werden. Der Benutzer kann lediglich diese Reihenfolge, die von oben nach unten abgearbeitet wird, verändern. Weitere Einflussmöglichkeiten gibt es nicht. Der Transformator bearbeitet diese Regeln in der Weise, wie es in Kapitel 2.1.2 beschrieben wurde, nämlich die erste ganz oben stehende Produktionsregel wird so oft wie möglich angewendet. Ist sie nicht mehr anwendbar wird die nächste Produktionsregel genommen und auf den Arbeitsgraphen (Startgraphen) angewendet, wieder so oft es geht. Wurde die letzte Regel bearbeitet, beginnt der Transformator wieder von vorne (nach demselben Verfahren). Die Transformation endet erst dann, wenn keine der Produktionsregeln aus der Treeview mehr anwendbar ist.

Der Steuerungseditor hingegen ermöglicht es, durch den Entwurf von Steuerungsgraphen, die Transformation durch den Automaten (siehe Kapitel 4) flexibler zu gestalten. Die wichtigsten Eigenschaften des Steuerungsgraphen sind, in Abhängigkeit von der Anwendbarkeit der Produktionsregeln bedingte Verzweigungen oder Schleifen beschreiben zu können. Der boolesche Aktionszustand verzweigt in Abhängigkeit von dem Rückgabewerte der Aktion. Produktionsregeln können explizit zur Graphersetzung angewendet werden (der Ansatz, der im Startgraphen gefunden wird, bleibt das Ergebnis einer nicht deterministischen Ansatzsuche).

Das Attributkonzept (siehe Kapitel 2.1.2) in AGG erlaubt es, die Ansatzsuche auf spezifisch attributierte Graphobjekte (Knoten und Kanten) des Arbeitsgraphen zu begrenzen. Durch die zusätzliche Forderung der Attributgleichheit, bei Morphismusabbildungen, kann der Suchraum im Arbeitsgraphen verkleinert werden.

Es ist interessant, wie der Einfluss des Benutzers, hinsichtlich der Graphersetzungssteuerung, in anderen Graphersetzungssystemen realisiert ist. Es sollen daher zwei Graphersetzungssysteme vorgestellt werden, die unterschiedliche Ansätze von Kontrollstrukturen für Graphersetzungen realisieren. Das erste Graphersetzungssystem ist Grrr, das eine visuelle Beschreibungssprache zur Spezifikation und Transformation von Produktionsregeln bietet. Grrr hat implizite Kontrollstrukturen. Das zweite Graphersetzungssystem ist PROGRES, das eine textliche und explizite Kontrollkomponente realisiert.

Der Steuerungseditor von AGG kombiniert diese beiden Konzepte in der Form, dass einerseits explizite Kontrollstrukturen realisiert werden, wie in PROGRES, und andererseits eine vollständig grafische Notation der Kontrollstrukturen gesichert wird, wie in Grrr. Die folgende Tabelle veranschaulicht dies noch einmal:

	PROGRES	**Grrr**	**Steuerungseditor**
Notation	Textlich	grafisch	grafisch
Kontrolle	Explizit	implizit	explizit

Tabelle 2 Kontrollkonzepte im Vergleich

Im folgenden Kapitel wird das Kontrollkonzept von Grrr vorgestellt. Da dieses Konzept auch auf einer grafischen Notation basiert, wie AGG, wird näher auf Grrr und seine Beschreibungssprache eingegangen.

3.3.1 Grrr

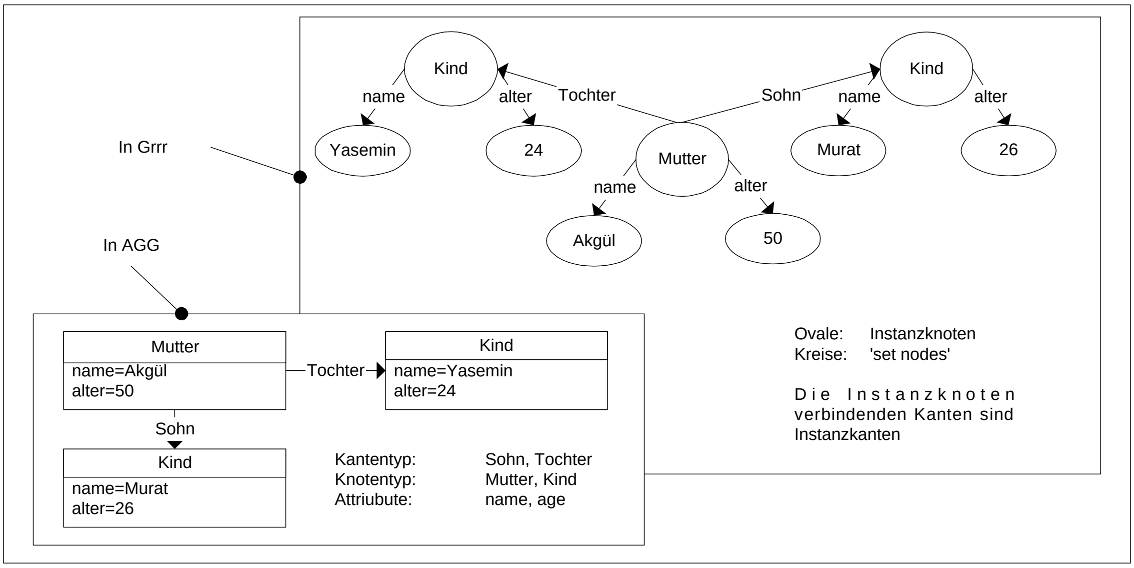


Abbildung 39 Datenbasis in Grrr und AGG

Grrr [King/Rodgers] bietet zur Spezifikation der Graphgrammatik drei unterschiedliche Ebenen an, die auf einer grafischen Beschreibungssprache basieren und durch Applikationsfenster (*Windows*) realisiert sind:

1. Ebene: Zum Entwurf des Arbeitsgraphen (in Grrr als Datenbasis bezeichnet) wird im *Schemawindow* (*schema window*) ein Graph zusammengesetzt, der aus Knoten und Kanten besteht. Im *Schemawindow* können ausschließlich Knoten vom Typ *‚set nodes'* eingefügt werden. Sie haben eine Kreisform. Kanten, die Knoten verbinden, sind immer gerichtet. Beide, Kanten und Knoten im *Schemawindow*, sind durch eine Beschriftung etikettiert. Die Etikettierung in Grrr entspricht den Knoten- und Kantentypen von AGG (siehe Abbildung 39).
2. Ebene: Zur Spezialisierung der *‚set nodes'* dürfen in Grrr spezielle Knoten und gerichtete Kannten den *‚set nodes'* zugewiesen werden. Dies geschieht im *Instanzwindow*. Die sogenannten *Instanzknoten* (*instance nodes* – ovale Repräsentation) werden mit den *‚set nodes'* durch spezielle *Instanzkanten* (*instance arcs*), die implizit beim Erzeugen eines *Instanzknotens* entstehen, verbunden. Von einem *‚set node*' können mehrere *Instanzkanten* ausgehen*;* Es ist aber nicht möglich, dass ein zweites *‚set node'* auf den gleichen *Instanzknoten* zeigt, wie das erste. Eine Parallelität in AGG ist mit der Attributierung der Graphobjekte (sowohl Kanten als auch Knoten) gegeben (siehe Abbildung 39). Dabei würden die Attribute der Graphobjekte in AGG den *Instanzknoten* in Grrr entsprechen. Folgender Unterschied ist aber beim Vergleich wichtig: Da keine Kanten aus Kanten austreten dürfen, können dementsprechend den Kanten, aus der Datenbasis in Grrr, keine *Instanzknoten* zugewiesen werden. In AGG jedoch sind Kanten und Knoten gleichermaßen attributierbar.

3. Ebene: Zur Spezifizierung der Transformationsregeln verwendet man das *Transaktionswindow*. Jede Regel besteht aus einer rechten Regelseite und aus einer linken Regelseite. Hier finden sich die *Instanzknoten*, *Instanzkanten* und *‚set nodes'* wieder, die bei der Spezifikation der Regelseiten benutzt werden. Soll ein Graphobjekt eine Variable sein, ist sie in Grrr von einem normalen Graphobjekt durch unterschiedliche Farbe und Font unterschieden. Bei der Ansatzsuche kann dann die Variable durch einen im Arbeitsgraphen gefundenen Ansatz belegt werden. Morphismusabbildungen werden durch die Namensgleichheit der Knoten und Kanten aus der linken Regelseite zur rechten Regelseite definiert. Soll es zwei Knoten auf der linken Regelseite mit gleichem Namen geben, dann können diese Namen mit einer Zahl indiziert werden, um sie voneinander zu unterscheiden. Grrr kennt im *Transaktionswindow* noch einen weiteren Knotentyp, der bei der Steuerung der Graphersetzung maßgeblich ist, nämlich den *Applikationsknoten* (*application node*). Sie repräsentieren eine Gruppe von Produktionsregeln, mit denen sie assoziiert sind. Der in den Arbeitsgraphen zuletzt eingefügte *Applikationsknoten* wird als erstes bearbeitet. *Applikationsknoten* sind also Auslöser einer Transaktion. Ein Arbeitsgraph ohne *Applikationsknoten* kann folglich nicht durch Transformationen verändert werden, daher muss im Arbeitsgraphen mindestens ein *Applikationsknoten* enthalten sein. *Applikationsknoten* können Argumente durch spezielle auf sie zeigende Kanten erhalten, damit können Konstanten oder Variablen durch *Applikationsknoten* verarbeitet werden. Möchte man in der linken Regelseite ausdrücken, dass ein gefundener Ansatz im Arbeitsgraphen, mit bestimmten *Instanzknoten*, nicht zur Graphersetzung verwendet werden soll, dann können sogenannte *‚negative' – Instanzknoten* erzeugt werden. Diese *Instanzknoten* (und die dadurch implizit entstehenden Instanzkanten) werden dann in dicker Linienführung grafisch repräsentiert und als *negative* bezeichnet. Negative aus Grrr entsprechen der negativen Anwendungsbedingung aus AGG (siehe Kapitel 2.1.2).

Im *Applikationswindow* kann dann eine Transformation durchgeführt werden. Als erstes sucht sich Grrr im Arbeitsgraphen einen *Applikationknoten*; Findet Grrr einen *Applikationknoten,* versucht es, eine Produktionsregel, aus der Gruppe die mit dem Applikationsknoten assoziiert wird, zu finden und anzuwenden. Das heißt es wird versucht für eine der linken Seiten dieser Produktionsregeln einen Ansatz im Arbeitsgraphen zu finden.
Produktionsregeln in Grrr können im Arbeitsgraphen neue *Applikationsknoten* erzeugen. Dabei entsteht eine hierarchische Struktur der Graphtransformation. Wie gesagt, der als neuestes in den Arbeitsgraphen eingefügte *Applikationsknoten* wird zur Transformation ausgelöst.

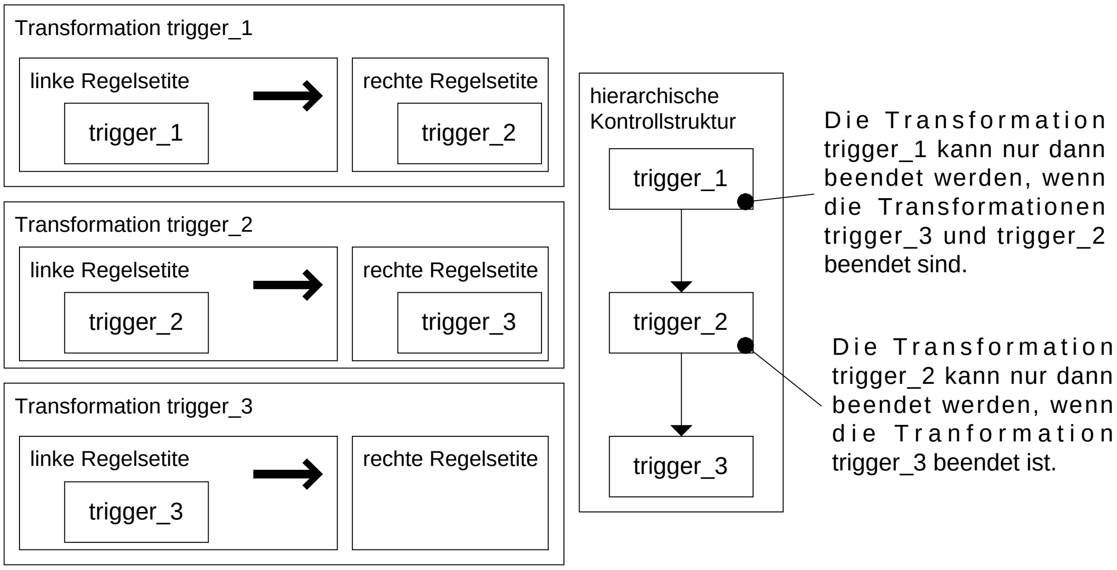


Abbildung 40 Hierarchische Kontrollstrukturen in Grrr

Die Abbildung 40 verdeutlicht die hierarchische Kontrollstruktur, die durch den Einsatz von *Applikationsknoten* (hier sind sie rechteckig dargestellt) erreicht werden kann. Damit die in der Hierarchie ganz oben liegende Transformation (*trigger_1*) ablaufen kann, muss im Arbeitsgraphen ein entsprechender *Applikationsknoten* eingefügt sein. Dann beginnt die Transformation *trigger_1*. Sie fügt in den Arbeitsgraphen einen anderen *Applikationsknoten* mit dem Namen *trigger_2* ein. *Trigger_2* wurde in den Arbeitsgraphen als neuestes eingefügt, daher wird nun die Transformation an *trigger_2* übergeben, die wiederum einen neuen *Applikationsknoten* (*trigger_3*) einfügt. Die Transformation *trigger_3* verursacht, dass der Applikationsknoten *trigger_3* im Arbeitsgraphen gelöscht wird. Nun befinden sich nur noch die *Applikationsknoten trigger_2* und *trigger_1* im Arbeitsgraphen. Von diesen beiden ist *trigger_2* als neuestes eingefügt worden, daher geht die Kontrolle wieder an *trigger_2* zurück.

Hierarchische Kontrollstrukturen, wie sie hier beschrieben wurden, finden sich auch in Rekursionsschemata wieder (siehe Kapitel 2.2.2). Durch das Auslösen eines neuen *Applikationsknoten* wird in Grrr eine neue Hierarchieebene erzeugt. In dieser Ebene kommen nur die Produktionsregeln zum Einsatz, die in der entsprechenden Transformation spezifiziert wurden. Der Steuerungseditor bietet keine hierarchischen Kontrollstrukturen an, um ähnliche Steuerungen zu realisieren. Eine grafische Lösung, um solche Hierarchien in den Steuerungseditor zu integrieren, wäre, Aktivitätszustände einzuführen, die einen neuen Steuerungsgraphen spezifizieren (siehe Abbildung 41). Aktionszustände waren ja Zustände, die atomare Aktionen ausführen (eine Graphersetzung). Aktivitätszustände könnten, weil sie nicht atomar sind, die Kontrolle an den Steuerungsgraphen weitergeben, den sie enthalten. Außerdem müsste noch ein neuer Aktionszustand spezifiziert werden, der Gruppen von Produktionsregeln ein- und ausschalten kann. In Abhängigkeit von dem zu interpretierenden Steuerungsgraphen könnten die neuen Aktionszustände gruppierte Produktionsregeln, zur Graphersetzung, aktivieren.

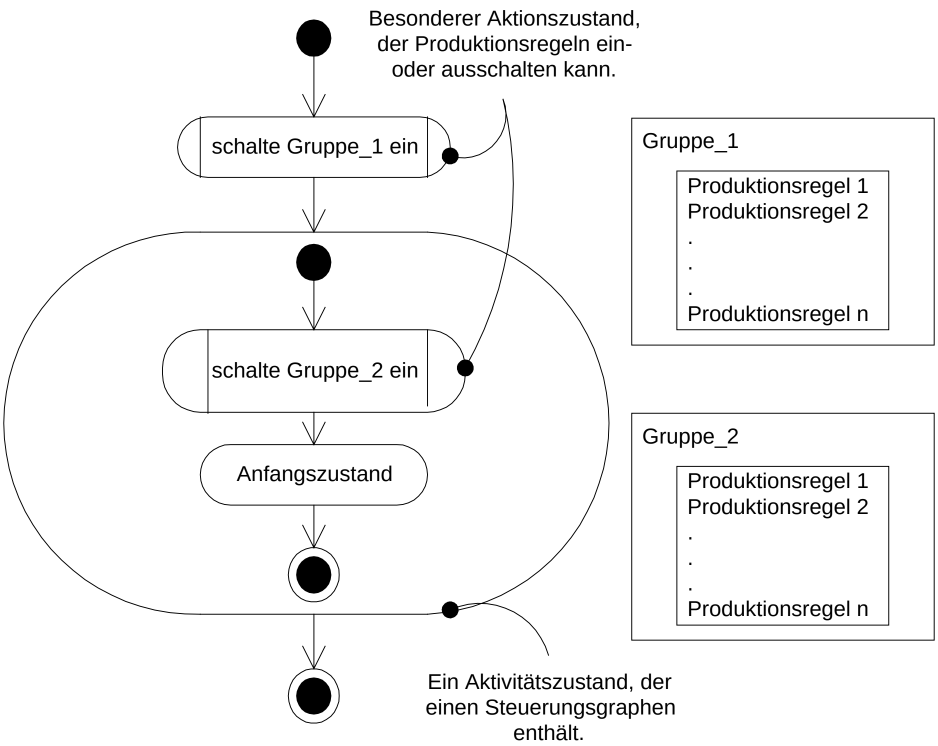


Abbildung 41 Hierarchische Steuerungsgraphen

Grrr bietet noch weitere Kontrollmöglichkeiten an, um Graphersetzungen zu steuern. Zum einen den *'single match rewrite'*, der auf Produktionsregeln operiert und den *'once only'* – Knoten, der in der linken Regelseite spezifiziert werden kann [Rodgers21]. Unter einem *'single match rewrite'* versteht man eine Produktionsregel, die nur einmal ausgeführt werden darf (sofern sie anwendbar ist), so dass diese Regel bei der nächsten Graphersetzung ignoriert wird. Die Entsprechung dieses Features im Steuerungsgraphen ist einfach, da sämtliche Aktionszustände explizite Graphersetzungen verursachen (sofern es natürlich einen Ansatz im Arbeitsgraphen gibt).

Das zweite Feature, die *'once only'* – Knoten, ermöglichen die Anwendung einer Produktionsregel (höchstens einmal), die auf der linken Regelseite höchstens einen *'once only'* – Knoten haben darf, auf alle Knoten im Arbeitsgraphen. Im Steuerungsgraphen kann man diese Kontrollstruktur dadurch erreichen, indem eine Schleife spezifiziert wird, die so oft einen Aktionszustand ausführt, wie er anwendbar ist.

Ein weiteres Graphersetzungssystem ist PROGRES, das ebenfalls umfangreiche Kontrollstrukturen anbietet. Graphersetzungen können hier, wie in einer Programmiersprache auch, programmiert werden.

3.3.2 **PRO**grammierte **GR**aph **E**r**S**etzungs**S**ysteme (**PROGRES**)

Es soll nicht verschwiegen werden, dass die PROGRES [Schürr98] – Kontrollstrukturen mächtiger sind, als sie in diesem Abschnitt beschrieben werden. Die hier vorgestellten Grundelemente dienen dem Vergleich mit dem Steuerungseditor.

In PROGRES ist die Definition der Datenbasis ausgeprägter als in Grrr oder AGG, weil es neben den Knotentypen auch Knotenklassentypen gibt. Ein Knotenklassentyp wird in PROGRES folgendermaßen spezifiziert:

```
node class PERSON
  intrinsic
        Name :     string ;
        Alter:     integer;
        Beruf:     string;
end ;
```

Abbildung 42 Knotenklassendeklaration in PROGRES

Die Abbildung 42 beschreibt keinen konkreten Knotentyp, sondern eine Klasse von Knotentypen (Schlüsselwörter sind kursiv). Ähnliche Strukturen findet man auch in objektorientierten Programmiersprachen wie Java oder C++. Knotenklassen werden durch das Schlüsselwort ‚*node* class' eingeleitet. Danach folgt die Attributierung dieser Klasse. Das Schlüsselwort *intrinsic* leitet die Deklaration eines ganz bestimmten Attributtyps ein. PROGRES kennt folgende Attributtypen:

1. *intrinische*: Diesen Attributen können zur Laufzeit beliebige Werte zugewiesen werden und sie können in der Klassendeklaration mit Werten vorbelegt werden.
2. *derived:* Es kann eine Berechnungsvorschrift angegeben werden, die zur Laufzeit berechnet wird.
3. *meta:* Sind Konstanten, die zur Laufzeit nicht veränderbar sind. Sie dürfen bei der Unterklassendefinition jedoch neu belegt werden.

Durch Vererbung ist es möglich neue Klassen zu entwerfen, die sich von der Klasse *PERSON* ableiten (siehe Metatypen Kapitel 3.2). Die Knotenklasse *AUTOFAHRER* in der folgenden Abbildung leitet sich von der Knotenklasse *PERSON* ab:

```
Node class AUTOFAHRER is a PERSON
 intrinsic
     SchadensfreieJahre:    integer ;
end ;
```

Abbildung 43 Vererbung von Knotenklassen in PROGRES

Die Klasse *AUTOFAHRER* erhält als zusätzliches Attribut *SchadensfreieJahre*. Das Schlüsselwort ‚*is a'* wurde schon in Kapitel 3.2 zur Metatypenspezifikation von Graphobjekten eingeführt. Knoten sind eigenständige Objekte, die an Parameter übergeben werden oder in einer Variablen festgehalten werden können.

Diese flexible Eigenschaft reduziert sich jedoch nur auf Knoten. Kanten können nicht als Klassen definiert werden. Sie werden nicht als eigenständige Objekte aufgefasst, sondern dienen nur zur Modellierung globaler Zusammenhänge. Kanten werden durch das Schlüsselwort ‚*edge type*' deklariert. Im folgenden soll ein Kantentyp deklariert werden, welcher Autofahrer, die sich kennen, in Relation setzen:

```
edge type kennenSich : PERSON [1:1] -> PERSON [0:n] ;
```

Abbildung 44 Kantendeklaration in PROGRES

Die Kantenrelation *kennenSich* besagt, dass eine *PERSON* keine oder mehrere *PERSONen* kennen kann.
Man stelle sich einen Arbeitsgraphen vor, der aus einem Netzwerk von Autofahrern besteht, die sich kennen oder die sich nicht kennen (je nachdem ob sie durch eine *kennenSich* Kante verbunden sind oder nicht).
In PROGRES können an zwei verschiedenen Orten Graphersetzungen durchgeführt werden. Die erste Stelle sind Produktionen. Produktionen in PROGRES bestehen, wie in Grrr und AGG auch, aus einer linken und rechten Regelseite, die durch Morphismusabbildungen verbunden sind. Bei der Ansatzsuche wird ein Teilgraph im Arbeitsgraph gesucht, der durch die rechte Regelseite ersetzt wird.
Daneben gibt es Transaktionen. Transaktionen sind kontrollierte Kombinationen aus einzelnen Graphersetzungen. Transaktionen werden komplett ausgeführt oder gar nicht ausgeführt. Transaktionen werden in PROGRES durch das Schlüsselwort *transaction* eingeleitet. Folgende Abbildung soll eine Transaktion zeigen:

```
transaction EliteAutofahrerMarkieren ( ) =
  use
        alleAutofahrer : AUTOFAHRER [0:n]
 do
        GetAllNodes ( AUTOFAHRER, out alleAutofahrer )
 &      for all index : AUTOFAHRER  := elem (alleAutofahrer)
        do
                    choose
                        when        index. SchadensfreieJahre > 10 then EinGuterFahrer (index)
                    else
                        Anfänger(index)
                    end;
        end ;
end;
```

Abbildung 45 Transaktionen in PROGRES

Bevor die Abbildung 45 beschrieben wird, sollen die wichtigsten Kontrollstrukturen von PROGRES kurz vorgestellt werden:

1. *&, and, or*: Um in der Transaktion spezifizieren zu können, dass einzelne Graphersetzungen hintereinander ausgeführt werden sollen, kann die Operation *&*, verwendet werden. Der *and* – Operator unterscheidet sich vom *&* - Operator darin, dass die Ausführung der Transaktion erst dann aufhört, wenn einer der *and* – Operatoren gescheitert ist. Die Bearbeitungsreihenfolge der and – Operatoren ist nicht deterministisch. Beim *or* – Operator verhält es sich so ähnlich, nur dass die Transaktion erst dann aufhört, wenn alle der *or* – Operatoren scheitern. Die Bearbeitungsreihenfolge der *or* – Operatoren ist ebenfalls nicht deterministisch.

2. *choose:* Damit können bedingte Verzweigungen formuliert werden. Die Bedingung kann durch das Konstrukt *‚when then'* spezifiziert werden.
3. *loop:* Erlaubt es, bedingte oder unbedingte Schleifen zu konstruieren. Das *‚when then'* – Konstrukt kann ebenfalls verwendet werden.
4. *for all:* Wiederholt für alle Einträge in einer Mengenvariable den Rumpf einer Schleife.
5. *skip:* Weist an, dass in einem Block nichts getan werden soll.

Die Abbildung 45 zeigt eine Transaktion, die alle Autofahrer, die länger als zehn Jahre unfallfrei waren, markiert, indem eine neue Kante vom Typ *guterFahrer* eingefügt wird, die als Ziel und Quelle diesen bezeichneten Autofahrer erhält. Die Produktionsregel, die diese Graphersetzung durchführt, heißt *EinGuterFahrer* und bekommt als Argument einen Autofahrer. Alle anderen Autofahrer werden analog mit einer Anfängerkante markiert (durch die Produktion *Anfänger*). Die Transaktion, die dies alles steuert, heißt *EliteAutofahrerMarkieren*. Da uns hier die Kontrollstrukturen interessieren, soll diese Transaktion noch einmal genauer betrachtet werden:

1. Es wird eine Variable mit dem Namen *alleAutofahrer* definiert. Die Kardinalität in der Eckigen Klammer sagt aus, dass diese Variable leer sein kann oder aus einer Menge von AUTOFAHRER bestehen kann.
2. *GetAllNodes* ist eine Anfrage (in PROGRES *query* genannt), die alle AUTOFAHRER aus der Datenbasis selektiert und in die Ausgabevariable *alleAutofahrer* zuweist.
3. Der Standardkonkatenator *&* zeigt an, dass die nächste *‚for all'* – Anweisung nach der *GetAllNodes* Anfrage stattfinden soll.
4. Die *‚for all'* – Anweisung weist nacheinander der Variablen *index* einen Wert aus der Menge *alleAutofahrer* zu. Für jede Belegung wird der Schleifenrumpf durchlaufen.
5. Die *choose* Bedingung ist eine generalisierte *if-then-else* – Anweisung. Wenn ein selektierter Autofahrer länger als 10 Jahre unfallfrei gefahren ist, dann wird die Produktion *EinGuterFahrer* ausgeführt. Sonst die Produktion *Anfänger*.
6. Ob ein Autofahrer länger als 10 Jahre unfallfrei war oder nicht entscheidet die *when-then* – Konstruktion.

Bedingte Verzweigungen, wie das *choose* – Element, sind auch im Steuerungseditor (nicht in dieser generalisierten Form) bekannt. Der Standardoperator *&* zeigt an, dass Programmteile nacheinander ausgeführt werden sollen, dies wird durch die Reihenfolge der Zustandsübergänge im Steuerungsgraphen spezifiziert. Der *or* – Operator in PROGRES kann durch das Gabelungssymbol im Steuerungsgraphen eine Entsprechung finden. Das Transaktionsmodell aus PROGRES ist im Steuerungseditor nicht realisiert.

PROGRES bietet viele Möglichkeiten an, in Abhängigkeit von Attributen der Knotentypen, den Programmverlauf zu steuern. Dies stellt einen wesentlichen Unterschied zu der in dieser Arbeit entworfenen Kontrollkomponente dar. Es ist denkbar, diesbezüglich neue Steuerungsobjekte zu definieren, die in Abhängigkeit von Attributen, der etwa am Ansatz beteiligten Knoten, den Programmverlauf steuern. Die Schwierigkeit wird jedoch darin liegen, durch prägnante grafische Steuerungsobjekte, komplizierte Anfragen an Attributbelegungen zu beschreiben, wie sie in PROGRES üblich sind. Vermutlich würde eine Lösung dieses Features die grafische Notation des Steuerungsgraphen durchbrechen, indem textliche Kontrollstrukturen, die auf Attributbelegungen reagieren können, in spezielle Aktionszustände als Ausdruck mitgegeben würden.

Im zweiten Kapitel wurde der Steuerungseditor entworfen, indem ein Anforderungskatalog erfüllt wurde. Der Steuerungsgraph wurde formal spezifiziert und Kontrollstrukturen mit anderen Kontrollstrukturen aus verschiedenen Graphersetzungssystemen verglichen. Dabei ist festgestellt worden, dass Kontrollstrukturen, in Abhängigkeit davon, ob sie auf grafischer oder textlicher Repräsentation basieren, mehr oder weniger differenziert sind. Insbesondere die Verschachtelung von Kontrollstrukturen ist in PROGRES prägnanter darzustellen als in Grrr oder in der Kontrollkomponente, die in dieser Arbeit entworfen wurde.

Der nächste Abschnitt beschreibt, wie die Kontrollkomponente softwaretechnisch umgesetzt wurde.

3.4 Das implementierte System

Dieses Kapitel soll mit einem screenshot eingeleitet werden, welches das Zusammenspiel des Steuerungseditors mit dem Graphersetzungssystem AGG veranschaulicht:

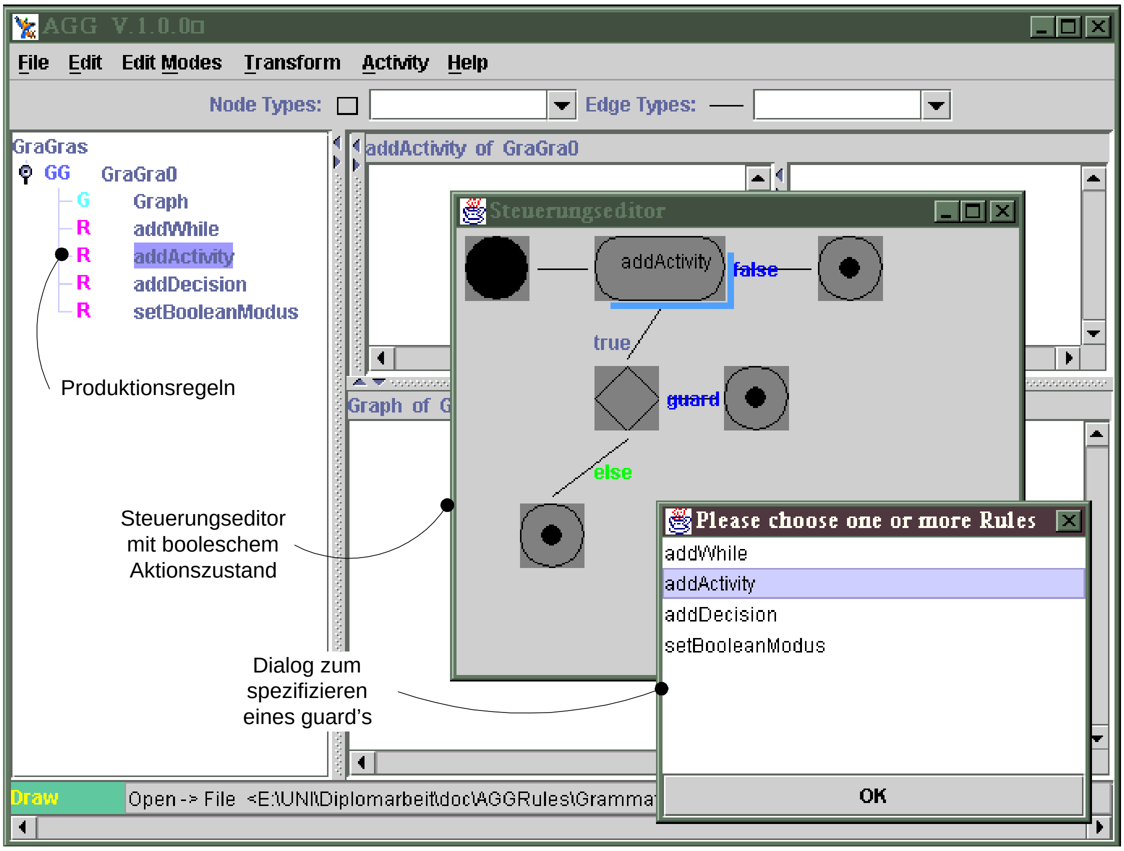


Abbildung 46 AGG und Steuerungseditor

In der Abbildung 46 ist das Hauptfenster des Graphersetzungssystems AGG zu erkennen. In der Treeview der Graphgrammatik *GraGra0* kann man die Produktionsregeln (*addWhile, addDecision* ....) erkennen, die vom Benutzer des Graphersetzungssystems spezifiziert wurden. Ist die Spezifizierung der Graphgrammatik beendet, kann man über das Menü *Activity* den Steuerungseditor öffnen. Der Steuerungseditor erhält als Übergabe die Graphgrammatik, die vom Benutzer spezifiziert wurde. Nach dem der Steuerungseditor aktiv ist, kann der Startgraph (siehe Abbildung 50 und Abbildung 51) durch den Benutzer bearbeitet werden. Im Steuerungseditor ist ein fortgeschrittener Steuerungsgraph zu erkennen, der aus sechs Steuerungsobjekten besteht. Beschließt der Benutzer, eine Transformation zu starten, würde dieser Steuerungsgraph durch den Automaten (siehe Kapitel 4) interpretiert. Der Steuerungsgraph enthält drei Stopsymbole, ein Startsymbol, einen booleschen Aktionszustand und ein Verzweigungssymbol. Das Verzweigungssymbol hat noch keine Wächterbedingung zugewiesen bekommen. Dieser Umstand ist daran zu erkennen, dass die eine ausgehende Kante des Verzweigungssymbols noch als Wächterbezeichnung *guard* enthält. Der Benutzer des Steuerungseditors wird deshalb durch den abgebildeten Dialog aufgefordert, einen oder mehrere Regelnamen auszuwählen (zur Zeit werden diese Regelnamen lediglich mit der ODER – Operation verknüpft). Nachdem die Wächterbedingung der *guard* – Kante spezifiziert wurde, kann die Transformation gestartet werden. Dazu wählt der Benutzer im Kontextmenü des Startsymbols die Option *start* aus dem *privates* – Abschnitt. Die Transformation des Steuerungsgraphen beginnt damit, dass das Startsymbol interpretiert wird. Als Folge dieser Interpretation wird der Zustandsübergang des Startsymbols ausgelöst. Der Folgezustand ist ein boolescher Aktionszustand mit der spezifizierten Aktion *addActivity*. Auf Grund dieser Tatsache besitzt dieser Aktionszustand zwei ausgehende Kanten (erste Kante mit *false* – Wächter, zweite Kante mit *true* – Wächter). War die Aktion im booleschen Aktionszustand erfolgreich (es gab eine Graphersetzung), wird die *true* – Kante ausgewählt und die Transformation hört auf, weil der Automat als nächstes das Stopsymbol, das einen Terminierungspunkt repräsentiert, interpretiert. War die Aktion des booleschen Aktionszustandes nicht erfolgreich (es gab keine Graphersetzung), wird als nächstes das Verzweigungssymbol interpretiert. Beim Verzweigungssymbol wird getestet, ob der *guard (*hier *addActivity)* erfüllt werden kann, das heißt, ob es für diese Produktionsregel einen Ansatz im Arbeitsgraphen gibt. Kann ein Ansatz gefunden werden, wird die *guard* – Kante ausgewählt, ansonsten die *else* – Kante. In beiden Fällen würde als nächstes das Stopsymbol interpretiert werden und die Transformation aufhören.

Im Anhang 6.4 befindet sich die API – Dokumentation der wichtigsten Klassen, die den Steuerungseditor und den Automaten implementieren. Das Konzept der Kontrollkomponente wird im folgenden Kapitel beschrieben.

3.4.1 Das statische Modell der Kontrollkomponente

Das statische Modell eines Systems beschreibt Relationen zwischen den im System vorhandenen Akteuren.

Die Kontrollkomponente besteht aus zwei Bestandteilen, dem Steuerungseditor, mit dem man Steuerungsgraphen spezifizieren kann, und dem Automaten, der diesen Steuerungsgraphen interpretiert und somit Graphersetzungen durch das AGG – Graphersetzungssystem veranlasst. Der Automat, zu finden in der Bibliothek (*package*) *automat*, ist ein Teil des Startsymbols, denn das Startsymbol implementiert die Methoden, um Transformationen zu starten, zu stoppen und den Automaten zu initialisieren. Der Steuerungseditor wird von der Klasse *DiagramContainer* implementiert und ist in der Bibliothek (*package*) *controlshapes* zu finden (siehe Abbildung 47). Im Anhang 6.4 befindet sich die Dokumentation der Klassen, die im Folgenden vorgestellt werden.

Der Steuerungseditor wird durch die Klasse *DiagramContainer* implementiert und erhält von AGG die Referenz des AGG Hauptfensters (*agg.gui.GraGraEditor*). *GraGraEditor* kann als die Schnittstelle zwischen AGG und der Kontrollkomponente aufgefasst werden. Aus dieser Referenz kann der Steuerungseditor, über die Methode *getGraGra*(), die vom Benutzer spezifizierte Graphgrammatik (*agg.editor.impl.EdGraGra*) erhalten. Der Steuerungseditor stellt die Referenz der Graphgrammatik über die Methode *getContentGraGra*() (mit Rückgabewert *EdGraGra*) allen Steuerungsobjekten zur Verfügung, die eine Referenz auf ihn haben.

Alle Steuerungsobjekte sind als Klassen in der Bibliothek *controlshapes* wiederzufinden. Die Steuerungsobjekte heißen hier *Start* (Startsymbol), *Stop* (Stopsymbol), *Activity* (Aktionszustand), *Transition* (Zustandsübergang), *Decision* (Verzweigungssymbol) und *While* (Schleifensymbol). Alle Steuerungsobjekte leiten sich von der Oberklasse *BaseShape* ab. Um ein Objekt der Klasse *BaseShape* zu instanziieren, muss die Referenz des Steuerungseditors (*DiagramContainer)* übergeben werden. Somit erhalten alle Steuerungsobjekte auch die Referenz auf die Graphgrammatik, denn *DiagramContainer* stellt die Methode *getContentGraGra*() zur Verfügung. Mit der Referenz *EdGraGra* interagieren folgende Steuerungssymbole:

- *Aktionszustand (controlshapes.Activity)*: Die Aktion des Aktionszustands kann über einen Dialog gewählt werden. Als Aktion wählt der Benutzer den Namen einer Produktionsregel aus, den der Dialog in Form einer Liste zur Verfügung stellt. Daher braucht *Activity* die Referenz *EdGraGra*, um Produktionsregeln (und somit die Namen) zu erhalten und an den Dialog übergeben zu können.
- *Verzweigungssymbol (controlshapes.Decision)* und *Schleifensymbol (controlshapes.While)*: Die beiden ausgehenden Kanten beider Symbole haben Wächterbedingungen. Eine Wächterbedingung erhält als Ausdruck eine Folge von Regelnamen, die wiederum aus einem Dialog ausgewählt werden können. Dazu muss wiederum *EdGraGra* an den Dialog Regelnamen übergeben werden.
- *Startzustand(controlshapes.Start*): Das Startsymbol besitzt den Automaten als ein Teil von sich. Damit der Automat einerseits die Aktion eines Aktionszustands bearbeiten kann und andererseits den Test auf die Anwendbarkeit einer Produktionsregel machen kann, benötigt er die entsprechenden Methoden der Referenz *EdGraGra*. Diese Methoden heißen *isApplicableRule*(*EdRule Regelname*) und *applyRule*(*EdRule Regelname*).

Der verwendete Dialog ist im statischen Modell der Kontrollkomponente nicht aufgeführt, weil er zum Verständnis des Konzepts nicht wichtig ist.

Die Klasse *BaseShape* realisiert die Schnittstelle *BaseInterface*. Diese Schnittstelle spezifiziert die Einfüge- und Löschprotokolle, die jedes einzelne Steuerungsobjekt einhalten muss. Daher sollen die Methoden dieser Schnittstelle genauer beschrieben werden:

- *addObject(Object obj):* Hier implementieren die Steuerungsobjekte das Einfügen eines anderen Steuerungsobjekts (*obj*). Dieses Objekt darf aber nur zwischen dem Ziel- und Quellobjekt der Kante eingefügt werden, die von der Methode *getActiveTransition*() zurückgegeben wird.
- *deleteObject(Object obj):* Hier implementiert das Steuerungsobjekt die eigene Löschprozedur. Dabei darf kein inkonsistenter Steuerungsgraph hinterlassen werden.
- *getActiveTransition():* Diese Methode soll die *aktivierte* Kante der Steuerungsobjekte zurückgeben. Neue Steuerungsobjekte können nur zwischen dem Ziel- und Quellobjekt dieser Kante (*controlshapes.Transition*) eingefügt werden.
- *setActiveTransition(Transition transition)*: Hiermit wird die *aktivierte* Kante spezifiziert.
- *updateDialog()*: Jedes Steuerungsobjekt implementiert ein eigenes Kontextmenü, das mit Aufruf dieser Methode aktualisiert wird. Daher sollten hier die Prozeduren berücksichtigt werden, die maßgeblich dafür sind, ob eine Option angeboten wird oder nicht.

Die Schnittstelle *BaseInterface* wird zunächst von der Klasse *BaseShape* implementiert. Steuerungsobjekte können auf diese Implementation zurückgreifen, weil sie *BaseShape* erben. Lediglich die steuerungsobjektspezifischen Methoden werden in den Klassen *Start*, *Stop*, *Transition*, *Activity*, *While* und *Decision* zusätzlich implementiert.

Alle Steuerungsobjekte haben eine grafische Repräsentation, die von der Klasse *BaseJComponent* implementiert wird. In dieser Klasse sind Methoden enthalten, wie zum Beispiel Steuerungsobjekte zu selektieren (dann wird eine andere Hintergrundfarbe gewählt) oder Steuerungsobjekte transparent darzustellen (dann wird der Hintergrund versteckt). Die symbolische Unterscheidung der Steuerungsobjekte geschieht dadurch, dass von der Klasse *BaseJComponent* geerbt wird, und die Methode *paintComponent*(*Graphics* g) überschrieben wird. Diese Methode erlaubt es, den Grafikkontext einer grafischen Komponente zu spezifizieren. Im Falle des Startsymbols zum Beispiel, wird ein ausgefüllter Kreis gezeichnet. Alle Steuerungsobjekte implementieren eine von *BaseJComponent* abgeleitete Klasse, die *JStart*, *JStop*, *JActivity*, *JDecision* und *JWhile* heißen. Diese Klassen werden von der Klasse *DiagramFeature* umhüllt. *DiagramFeature* implementiert den Vergrößerungsrahmen, der es dem Benutzer erlaubt, per Maus die Objektgröße zu ändern.

Die oberste grafische Klasse, die der Steuerungseditor (*DiagramContainer*) darstellt, ist *DiagramFeature*. Es sollte betont werden, dass die Steuerungsobjekte, wenn sie im Steuerungseditor dargestellt werden, vom Steuerungseditor nicht auf die Konsistenz den Steuerungsgraphen betreffend geprüft werden. Die Steuerungsobjekte sorgen selber, durch die Implementierung der Schnittstelle *BaseInterface*, dafür, dass korrekte Steuerungsgraphen entstehen.

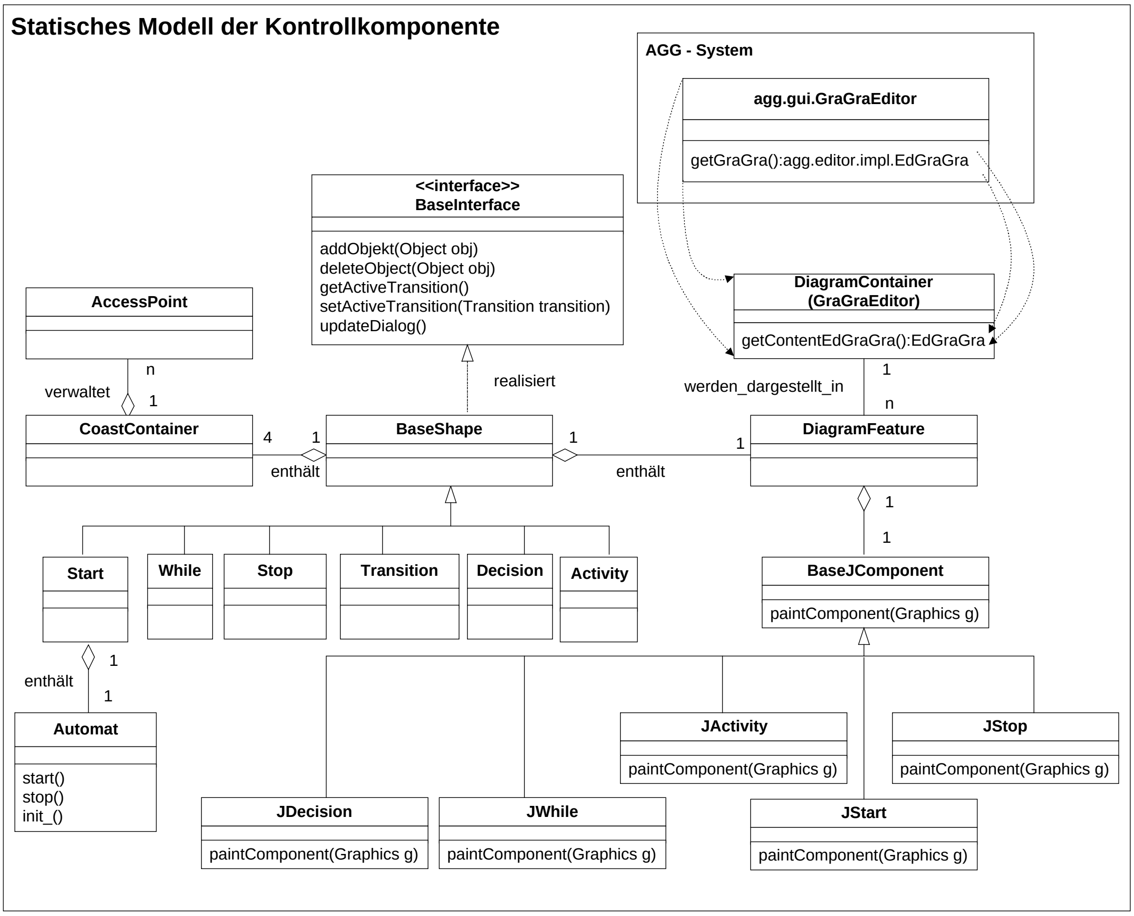


Abbildung 47 Das statische Modell der Kontrollkomponente

Im folgenden Kapitel wird die Realisierung des Automaten beschrieben. Es gibt zwei Ansätze: Der erste, der nicht realisiert wurde, ist der kompilative Ansatz. Der zweite Ansatz ist der interpretative. Die technische Realisierung beider Ansätze wird erläutert.

.

4 Der Automat

Der Steuerungseditor erlaubt es Steuerungsgraphen zu spezifizieren. Der Programmteil, der diesen Steuerungsgraphen interpretiert und in Abhängigkeit davon Graphersetzungen veranlasst, heißt Automat. Wird eine Transformation durch den Benutzer gestartet, so beginnt der Automat den Steuerungsgraphen zu interpretieren. Die Transformation des Arbeitsgraphen beginnt an einem Startsymbol, das pro Arbeitsgraph einmal vorhanden sein muss. Die Transformation hört an einem Stopsymbol auf. Transformationen im Steuerungseditor sind nicht atomar, denn scheitert eine Anweisung, wie zum Beispiel die Anwendung einer Regel durch den Aktionszustand, so geht die Transformation trotzdem weiter. Das Transaktionsmodell in PROGRES unterscheidet sich diesbezüglich durch eine atomare Transformation. Transaktionen in PROGRES spezifizieren die kontrollierte Kombination von Graphersetzungen. Eine Transaktion wird, wie in Datenbanken auch, entweder komplett oder gar nicht ausgeführt. Scheitert eine Transaktion, so wird der Ausgangszustand des Arbeitsgraphen wiederhergestellt. Die folgende Tabelle zeigt den Zusammenhang noch einmal:

	PROGRES	**Steuerungseditor**
Graphersetzungskontrolle	<u>Transaktion</u> • Kontrollierte Folge von kombinierten Grapher-setzungen.	<u>Transformation</u> • Beginnt an einem Startsymbol. • Hört bei einem Stopsymbol auf.
Atomarität	<u>Atomar</u> • Wird ganz durchgeführt oder gar nicht. • Beim Scheitern einer Anweisung, wird der Ausgangszustand wieder hergestellt.	<u>nicht atomar</u> • Wird bis zum Stopsymbol ausgeführt, auch wenn Anweisungen scheitern

Tabelle 3 Transformationen in PROGRES und im Steuerungseditor

Die nicht atomare Transformation spiegelt sich in der interpretativen Lösung des Automaten wieder. Neben einer interpretativen Lösung des Automaten ist auch eine kompilative Lösung vorstellbar. Das folgende Unterkapitel skizziert diese Lösung.

4.1 Die kompilative Lösung des Automaten

In Artikel [Taentzer] wird beschrieben, wie Graphtransformationen in objektorientierte Sprachen überführt werden können. Die folgenden Ausführungen greifen auf Erkenntnisse dieses Artikels zurück.

Zunächst soll die Interaktion des Benutzers mit einer Kontrollkomponente, die Graphersetzungen kompilativ steuert, beschrieben werden. Hat der Benutzer eine Graphgrammatik spezifiziert und möchte er dann eine gesteuerte Graphersetzung programmieren, so öffnet er den Steuerungseditor. Es öffnet sich ein Fenster, in welchem der

Benutzer (ähnlich wie bei PROGRES) Kontrollstrukturen textlich eingeben kann. Nachdem er die Kontrolle spezifiziert hat (in Form eines Java – Programms) wird dieser Code kompiliert. Bei erfolgreicher Kompilation kann die spezifizierte Graphersetzung durch den Computer ausgeführt werden. Der große Vorteil einer solchen Lösung wäre, dass die Kontrollstrukturen der Programmiersprache verwendet werden können, auf welcher der komplilative Automat basiert (bei AGG wäre es Java). Der Nachteil ist, dass eine visuelle Darstellung von Kontrollstrukturen nicht mehr gegeben ist und die Programmierung von Steuerungsanweisungen weniger intuitiv wird. Außerdem muss eine zusätzliche Komponente implementiert werden, die den im kompilativen Steuerungseditor spezifizierten Javacode auf die Syntax hin überprüft, so dass bei Kompilationsfehlern die Zeile markiert wird, die fehlerhaft ist.

Wie kann man nun eine Schnittstelle zwischen dem Graphersetzungssystem AGG definieren, um so Graphersetzungen zu kontrollieren?

Zunächst ist die Plattform, auf welcher der kompilative Steuerungseditor basieren soll, durch AGG vorgegeben, das in Java implementiert wurde.

Folgende Schritte skizzieren die kompilative Lösung des Automaten und den entsprechenden Steuerungseditor:

1. Der Steuerungseditor erhält wie in der interpretativen Lösung des Automaten eine vollständige Graphgrammatik. Dies kann dadurch geschehen, dass der Benutzer über einen Menüeintrag eine Graphgrammatik auswählen kann.
2. Die übergebene Graphgrammatik wird intern vom Steuerungseditor in eine Javaklasse übersetzt. Dies geschieht dadurch, dass die Graphgrammatik rekursiv abgearbeitet wird. Dabei werden Startgraph und Regeln in eine Javaklasse überführt, die im Konstruktor die übergebene Graphgrammatik neu erzeugt (siehe Anhang 6.3). Im einzelnen werden dabei get – Methoden auf die übergebene Graphgrammatik angewendet, um so die Datenstrukturen der Graphgrammatik zu traversieren und die entsprechenden set - Methoden auf die Instanz der Graphgrammatik in der Javaklasse anzuwenden, um eine Kopie der übergebenen Graphgrammatik zu erhalten. Als Ergebnis erhält der Steuerungseditor die Javacoderepräsentation der übergebenen Graphgrammatik, wohlgemerkt in textlicher Notation.
3. Die erzeugte Javaklasse aus 2. hat als Attribute die Regelklassen der Graphgrammatik und die Graphgrammatikklasse selbst. Die Graphgrammatikklasse implementiert die Funktionen *führeRegelAus()* und *istRegelAnwendbar(Regelname)*. Diese zwei Methoden werden jetzt in der Javaklasse für alle Graphgrammatikregeln, die als Attribute vorhanden sind, neu implementiert. Als Ergebnis entstehen zum Beispiel die Methoden *istRegel1Anwendbar()* oder *führeRegel2Aus()*.
4. Nun muss die Javaklasse noch um eine Methode erweitert werden, die der Benutzer selbst programmiert. Da die Ausführung dieser Methode parallel zur Ausführung des Steuerungseditors geschehen muss, kann die *run*() –

Methode aus der Schnittstelle *Runnable* in Java verwendet werden. Wird eine Klasse, die diese Schnittstelle (in Java *interface genannt*) implementiert, der Klasse *Thread* übergeben, so laufen die Anweisungen der Methode *run*() nebenläufig ab. Deshalb wird die Javaklassse, die generiert wurde, mit der Schnittstellenprogrammierung der Schnittstelle *Runnable* erweitert.

5. Der Steuerungseditor öffnet einen Editierbereich, in dem der Benutzer seine Anweisungen spezifizieren kann. Diese Anweisungen werden der *run*() – Methode durch den Editor zugewiesen.
6. Der Benutzer veranlasst die Kompilierung der Javaklasse. Die *run*() – Methode wurde von ihm selbst spezifiziert. Darin werden Graphersetzungen gesteuert.
7. Der Steuerungseditor veranlasst die Ausführung der *run*() – Methode durch die Erzeugung eines neuen *Threads*. Dieser *Thread* entspricht dem Automaten, der Graphersetzungen steuert. Da die Anweisungen in der *run*() – Methode des *Threads* zunächst kompiliert werden müssen, bevor sie ausgeführt werden können, wird dieser Atutomat als kompilativ bezeichnet.

Im Folgenden wird ein Beispiel für die Steuerung der Graphersetzung gegeben, indem die run() – Methode überschrieben wird:

```
public void run() {
        if ( isRule1Applicable() && isRule2Applicable () ) {
                while ( isRule2Applicable() ) execRule2() ;
                execRule1() ;
        } else  execRule4() ;

        int INDEX = 0 ;
        for ( INDEX = 0 ; INDEX < 10 && isRule3Applicable() ; INDEX ++)
            execRule3() ;
} //end run()
```

Abbildung 48 Kompilative Kontrollstruktur

Die Abarbeitung der Anweisungen in der Abbildung 48 geschieht sequenziell. Als erstes wird die *if-then-else* – Anweisung ausgeführt und danach die *for* – Anweisung. Ist eine der Regeln Rule1 oder *Rule2* nicht anwendbar (das heißt es gibt keinen Ansatz für diese Regeln) wird die *else* – Anweisung ausgeführt, ansonsten wird die *while* – Schleife und die Anwendung der *Regel1* auf den Startgraphen sequenziell ausgeführt. Die *for* - Schleife wird zehnmal die *Regel3* anwenden, wenn sie anwendbar ist, was durch den *&&* (UND) – Operator in der *for* – Anweisung sichergestellt wird.

Bedingte Verzweigungen, sequenzielle Anweisungen und WHILE – Schleifen kennt der interpretative Automat auch, jedoch ist eine *for* – Schleife aus dem kompilativen Automaten nicht bekannt. Der kompilative Automat hingegen erbt alle Steuerungsanweisungen und Vergleichsoperatoren aus Java, die Steuerungsmöglichkeiten enorm erhöhen. Der Preis dafür ist, dass der Benutzer Java – Kenntnisse besitzen muss und keine intuitive Steuerung der Graphersetzung mehr möglich ist.

Die Transformation bei der kompilativen Lösung beginnt mit dem Programmierblock der *run*() – Methode und hört mit der letzten Anweisung auf. Sie ist nicht atomar und kann jederzeit durch den Editor angehalten oder gestoppt werden.

Die kompilative Lösung des Automaten ist nicht realisiert. Es existieren lediglich die Implementierungen zur Überführung einer AGG – Graphgrammatik in eine Javaklasse.

Die interpretative Lösung stellt eine fest implementierte Interpretation des Steuerungsgraphen dar, wie folgender Abschnitt zeigt.

4.2 Die interpretative Lösung des Automaten

Damit der Automat eine Graphersetzung steuern kann, muss ihm zunächst der Steuerungsgraph vom Steuerungseditor übergeben werden. Der Steuerungseditor gibt dem Benutzer die Möglichkeit, die Transformation zu starten, indem durch das Kontextmenü des Startsymbols die Option *'start'* angeboten wird. Wird diese Option gewählt, beginnt der Automat damit, den Steuerungsgraphen zu interpretieren und Graphersetzungen durch das Graphersetzungssystem AGG zu veranlassen. Die Interpretation des Steuerungsgraphen geschieht nebenläufig zur Ausführung des Steuerungseditors. Daher kann die Transformation durch den Benutzer jederzeit gestoppt werden (durch die Option *'stop'*). Implementierungstechnisch handelt es sich beim Automaten um eine Java – Klasse, die das Interface *Runnable* implementiert. In der *run*() – Methode dieses Interfaces wird die Interpretierung des Steuerungsgraphen implementiert. Der Steuerungseditor erzeugt ein *Thread* (ein nebenläufig ablaufendes Programm in Java), indem das *Runnable* (die Automatenklasse) übergeben wird. Bei Transformation wird dann die *run*() – Methode ausgeführt.

Im Kapitel 2.1.2 wurde die Transformationsschleife, die in AGG zur Zeit realisiert ist, vorgestellt. Die folgende Abbildung zeigt, dass der Automat auf die spezifizierten (siehe Kapitel 3.1.1) Steuerungssymbole reagiert und in Abhängigkeit des zu interpretierenden Steuerungssymbols verzweigt:

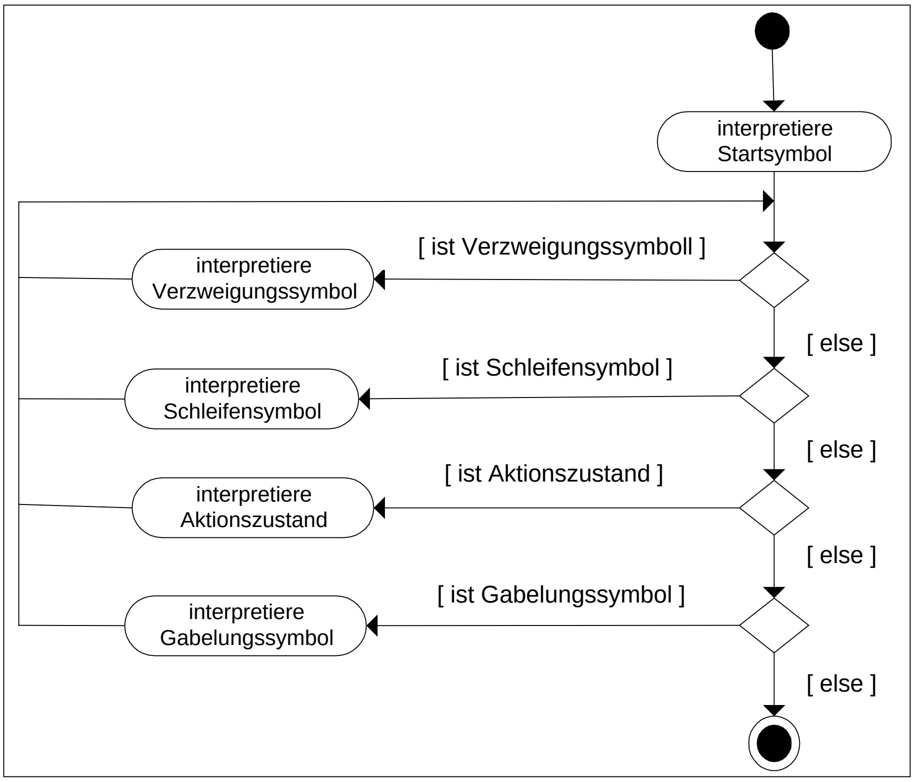


Abbildung 49 Transformationsschleife im Automaten

Die Interpretation des Steuerungsgraphen beginnt damit, dass das Startsymbol interpretiert wird. Das Startsymbol hat immer nur eine ausgehende Kannte; Daher ist die Interpretation ganz einfach, nämlich ein Zustandsübergang wird ausgelöst, der auf das nächste Steuerungssymbol zeigt. Ist das nächste Steuerungssymbol ein Stopsymbol, so scheitern alle Verzweigungen und der Automat terminiert. Andernfalls handelt es sich um eines der anderen Steuerungssymbole, die unterschiedlich interpretiert werden:

- *Das Verzweigungssymbol*: Das Verzweigungssymbol hat zwei ausgehende Kanten. Daher sind Wächterbedingungen zwingend, um deterministisch entscheiden zu können, welcher Zustandsübergang ausgelöst werden soll. Eine der Wächterbedingungen ist der *guard*. Ist dieser als *true* auszuwerten, wird der Zustandsübergang, zu dem *guard* gehört, ausgelöst, sonst wird der else – Zustandsübergang ausgelöst.
- *Das Schleifensymbol:* Die Entscheidung bezüglich des auszulösenden Zustandsübergangs geschieht nach demselben Muster wie beim Verzweigungssymbol.
- *Das Gabelungssymbol:* Dieses Symbol hat mehrere ausgehende Kannten, die in einer ODER – Beziehung miteinander stehen. Eine der Kanten wird nicht deterministisch ausgewählt. Die ausgewählte Kante entspricht einem Zustandsübergang, der dann ausgelöst wird.
- *Der Aktionszustand:* Als erstes wird die Aktion ausgeführt, nämlich die Anwendung einer Produktionsregel auf den Startgraphen. Danach muss erst anhand der ausgehenden Kanten entschieden werden, ob es sich um einen

booleschen oder impliziten Aktionszustand handelt. Handelt es sich um einen impliziten Aktionszustand, so wird die ausgehende Kante sofort ausgelöst. Beim booleschen Aktionszustand wird anhand des Rückgabeparameters der Aktion die Kantenwahl getroffen. Ist die Aktion positiv gewesen, das heißt es fand eine Graphersetzung im Arbeitsgraphen statt, wird die Kante mit dem *true* – Wächter ausgelöst, sonst die mit dem *false – Wächter.*

- *Der Zustandsübergang:* Ein Zustandsübergang bedeutet, dass dem Automaten das nächste zu bearbeitende Steuerungssymbol übergeben wird. War der Zustandsübergang implizit, so wurde er automatisch ausgelöst, andernfalls hatte er einen Wächter, von dessen Auswertung es abhängig gemacht wurde, ob diese Kante ausgelöst wird oder nicht. Findet ein Zustandsübergang statt wird das Folgesymbol mit einer erhellten Hintergrundfarbe dargestellt (highlighting).

Es ist festzustellen, dass der Automat eine Schleife durchläuft, die permanent ein Steuerungssymbol interpretiert. Trifft keine der Verzweigungen im Automaten zu, so war es ein Stopsymbol und der Automat terminiert. Der Steuerungseditor sorgt dafür, dass nur genau ein Startsymbol vorhanden ist, daher braucht auch nicht auf dieses Steuerungssymbol eingegangen zu werden (durch eine weitere Verzweigung).

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Steuerungseditor für die attributierte Graphersetzung in AGG entworfen, der das Graphersetzungssystem AGG um eine Kontrollkomponente erweitert. Das Hauptmerkmal dieser Kontrollkomponente ist die Spezifizierung von Steuerungsgraphen, die, durch einen Automaten interpretiert, Graphersetzungen in AGG veranlasst. Die Akteure des Steuerungsgraphen sind Steuerungsobjekte, die unterschiedliche Kontrollstrukturen repräsentieren.

Der Steuerungseditor unterstützt den Benutzer in der Spezifikation des Steuerungsgraphen in dreierlei Hinsicht: Erstens können nur syntaxgerechte Steuerungsgraphen erzeugt werden. Zweitens werden die Steuerungsobjekte im Sinne besseren Layouts des Steuerungsgraphen durch den Editor vorplatziert. Drittens kann durch die Kontextmenüunterstützung des Steuerungseditors ein flüssiges Interagieren mit dem Steuerungseditor erreicht werden.

Der Entwurf des Steuerungseditors spiegelt eine Synthese wieder, die Kontrollstrukturen der Programmschemata mit der Semantik der Aktivitätsdiagramme vereint. Bei der grafischen Notation wurde an das Aktivitätsdiagramm angelehnt, weil es allgemein bekannt ist und hohen Zuspruch in der Systemmodellierung genießt.

Es ist festgestellt worden, dass andere Graphersetzungssysteme, wie Grrr [King/Rodgers] oder PROGRES [Schürr], ebenfalls Kontrollkomponenten realisieren. Der Steuerungseditor in AGG ordnet sich zwischen die Kontrollkomponenten dieser beiden Graphersetzungssysteme ein, denn er erlaubt es, Kontrollstrukturen in Form von Steuerungsgraphen grafisch zu spezifizieren (wie in Grrr), und er bietet explizite Steuerungsanweisungen an (wie in PROGRES).

Hinsichtlich der Weiterentwicklung des Steuerungseditors kann gesagt werden, dass hierarchische Kontrollstrukturen, wie in Grrr realisiert, ein interessanter Ansatz sind, um die Funktionalität des Steuerungsgraphen zu erweitern. Außerdem ist festzustellen, dass die Steuerung der Graphersetzung durch den Automaten, nicht von den Attributen der Graphobjekte im Arbeitsgraphen abhängig gemacht werden kann. An dieser Stelle müssen neue Steuerungsobjekte entworfen werden, die auf Attribute, der zum Beispiel an einem Ansatz beteiligten Graphobjekte, reagieren können, um so bedingte Verzweigungen zuzulassen. Dabei darf jedoch die grafische Notation des Steuerungsgraphen nicht durchbrochen werden, um die erwähnte Synthese nicht zu zerstören.

Durch die Erweiterung der Kontrollkomponente erhöht sich die Funktionalität des Graphersetzungssystems AGG um einen bedeutenden Punkt.

6 Anhang

Im Anhang befinden sich komplettierende Informationen zu den einzelnen Kapiteln. Es werden die Kontextmenüspezifikationen der einzelnen Steuerungsobjekte vervollständigt. Die komplette Spezifikation des Steuerungsgraphen wird durch eine Graphgrammatik nachgeliefert und die API Dokumentation des Steuerungseditors angefügt.

6.1 Die Kontextmenüspezifikation der Steuerungsobjekte

Im Kapitel 3.1.4 wurde die Spezifikation des Kontextmenüs für das Aktionssymbol beschrieben. Es wurde darauf aufmerksam gemacht, dass die Kontextmenüs der einzelnen Steuerungsobjekte sich sehr ähnlich sind. Die Kontextmenüs unterscheiden sich nur im Abschnitt ‚privates' voneinander, denn dort befinden sich die steuerungsspezifischen Optionen der einzelnen Steuerungsobjekte. In den folgenden Unterkapiteln finden also die Beschreibungen der privaten Optionen statt.

6.1.1 Kontextmenü des Stopsymbols

Das Stopsymbol besitzt nur den ‚basics' – Abschnitt des Kontextmenüs, wie folgende Abbildung zeigt:

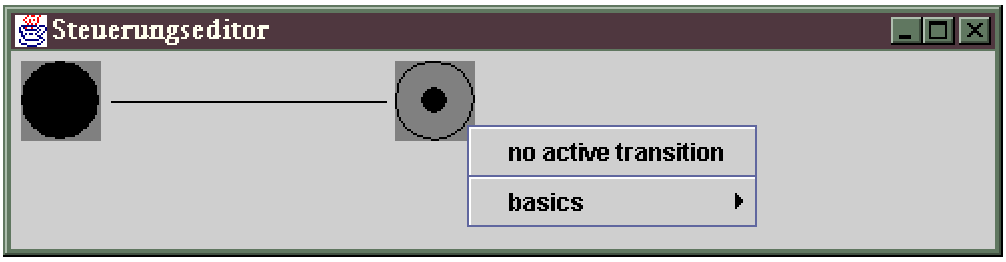


Abbildung 50 Kontextmenü des Stopsymbol

6.1.2 Private Optionen des Startsymbols

Die folgende Abbildung zeigt, dass das Startsymbol drei Optionen anbietet, nämlich die Optionen ‚start', ‚stop' und ‚init':

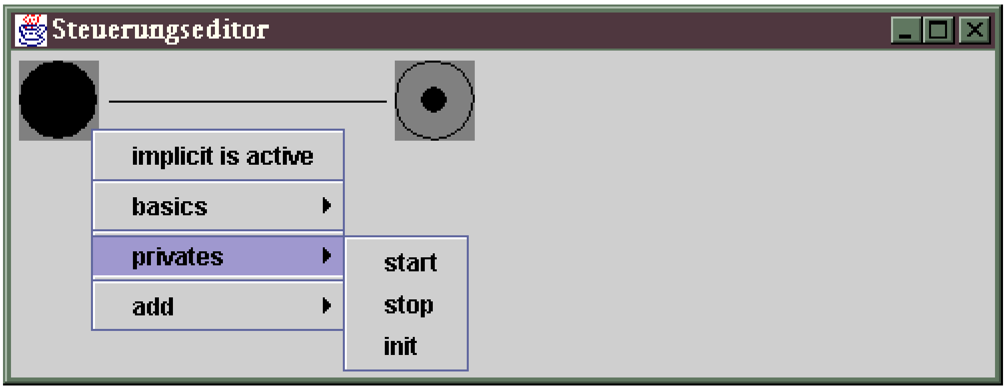


Abbildung 51 Kontextmenü des Startsymbols

Beschreibung der Optionen:

1. ‚start': Mit dieser Option kann der Benutzer eine Graphersetzung beginnen, indem der Automat gestartet wird. Dieser bekommt als Bearbeitungsgrundlage das Startsymbol, das den Eintrittspunkt eines Steuerungsgraphen repräsentiert. Dann beginnt der Automat den Steuerungsgraphen zu bearbeiten. Trifft er auf einen Aktionszustand, versucht er die Aktion (also eine Graphersetzung) durchzuführen.
2. ‚stop': Stoppt den Automaten. Danach initialisiert sich der Automat, das heißt die Graphersetzung beginnt beim nächsten Start des Automaten wieder beim Startsymbol.
3. ‚init': Initialisiert den Automaten. Beim nächsten Start des Automaten beginnt die Bearbeitung des Steuerungsgraphen wieder beim Startsymbol..

6.1.3 Private Optionen des Verzweigungs- und Schleifensymbols

Beide Steuerungssymbole kennen nur eine private Option, nämlich die Option ‚select guard'. Wird diese Option gewählt, so können über einen Dialog eine oder mehrere Regelnamen selektiert werden. Zum jetzigen Zeitpunkt der Implementierung des Steuerungseditors werden diese Regelnamen mit einer Oderbeziehung verknüpft. Die Forderung, komplette boolesche Ausdrücke zuzulassen, ist nicht implementiert.

Folgende Abbildung zeigt das Kontextmenü des Verzweigungssymbols und ein Schleifensymbol (WHILE – Schleife):

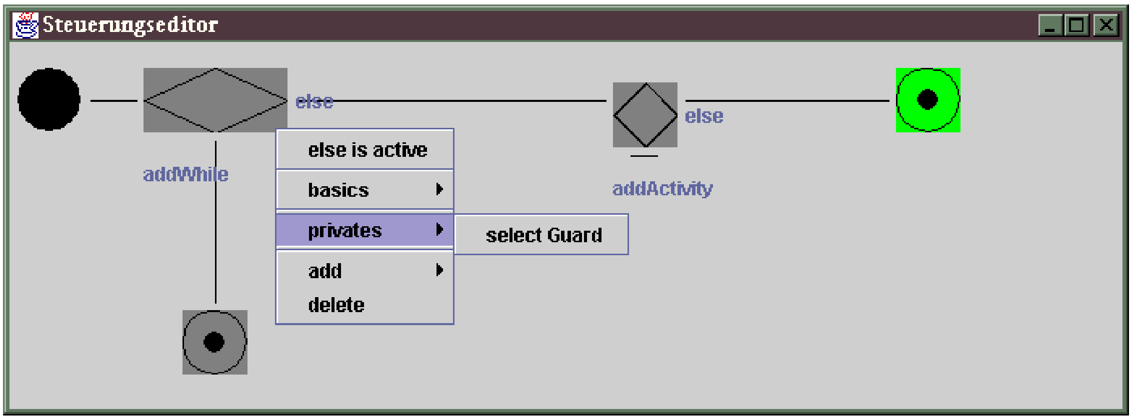


Abbildung 52 Kontextmenü des Verzweigungssymbols

Wird ein Verzeigungs- oder Schleifensymbol erzeugt, so muss erst aus dem eben erwähnten Dialog mindestens eine Regel gewählt werden. Dieser Dialog sieht genau so aus wie in Abbildung 35 gezeigt.

6.1.4 Kontextmenü des Gabelungs- und Synchronisationssymbols

Diese Objekte sind zu diesem Zeitpunkt der Diplomarbeit nicht implementiert. Daher kann kein Screenshot vorgestellt werden. Aber es kann zumindest beschrieben werden, wie die privaten Optionen, sofern solche vorhanden sind, aussehen. Um es vorwegzunehmen, das Synchronisationssymbol hat keine privaten Optionen, während das Gabelungssymbol zwei Optionen kennt, die das Einfügen und Löschen einer Gabelung regeln:

1. ‚add line': Mit dieser Option können neue Gabelungen, die aus dem Gabelungssymbol austreten, erzeugt werden. Alle Gabelungen werden sofort mit dem korrespondierenden Synchronisationssymbol verbunden.
2. ‚delete line': Diese Option wird zum Löschen einer Gabelung gewählt. Ein Gabelungssymbol kann immer nur bis zu den letzten zwei Gabelungen verkleinert werden.

6.2 Graphgrammatische Spezifikation des Steuerungsgraphen

Hier sollen alle Graphgrammatikregeln vorgestellt werden, die nötig sind, um den Steuerungsgraphen zu spezifizieren. Die Definition der Metatypen und die Interpretation dieser Regeln sind im Kapitel 3.2 erklärt. Der Startgraph, auf den diese Grammatik definiert sein soll, sieht folgendermaßen aus:

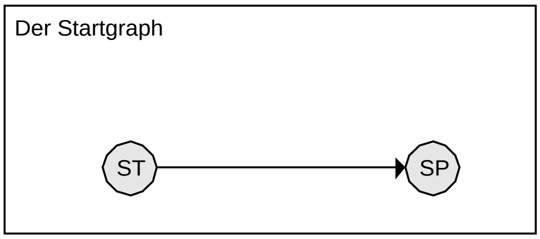
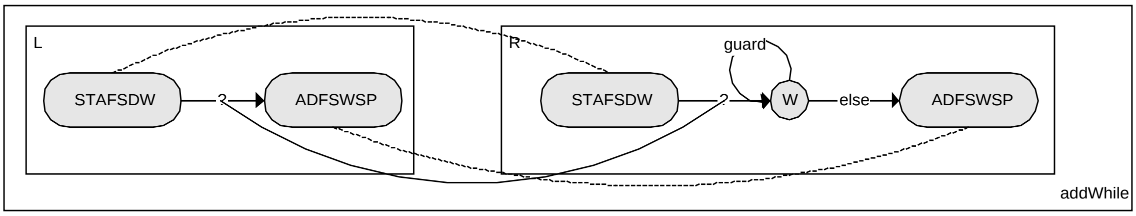


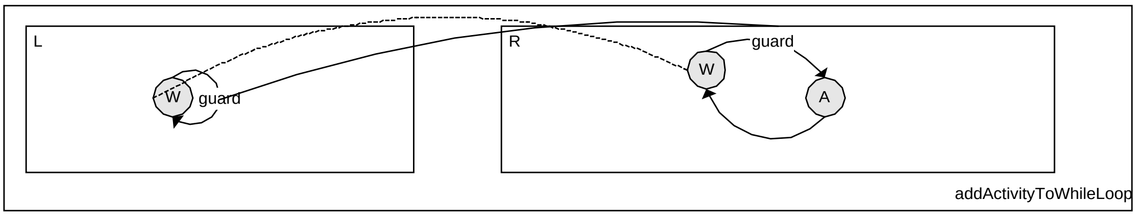
Abbildung 53 Der Startgraph

Der Initiale Startgraph ist ein auf das Stopsymbol zeigendes Startsymbol. Dies war eine der Voraussetzungen für die Syntaxsteuerung durch den Steuerungseditor, denn es dürfen neue Steuerungsobjekte immer nur zwischen zwei Steuerungsobjekten, die durch eine Kante verbundenen sind, eingefügt werden.

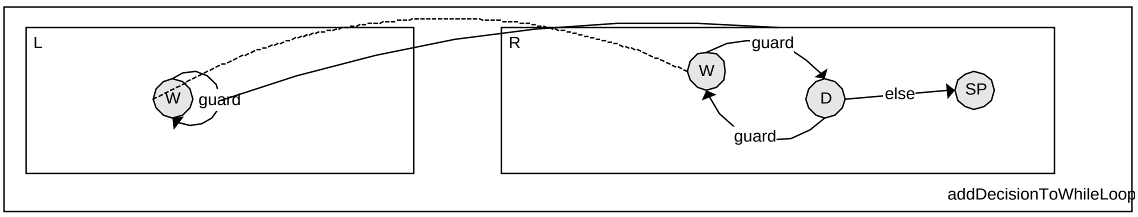
Folgende Tabelle listet alle Graphersetzungsregeln auf:



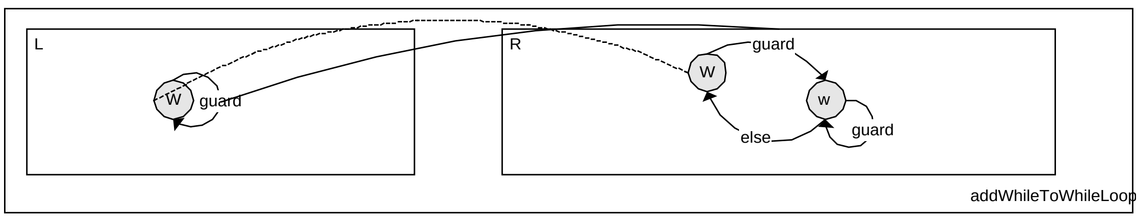
Einfügen eines Schleifensymbols



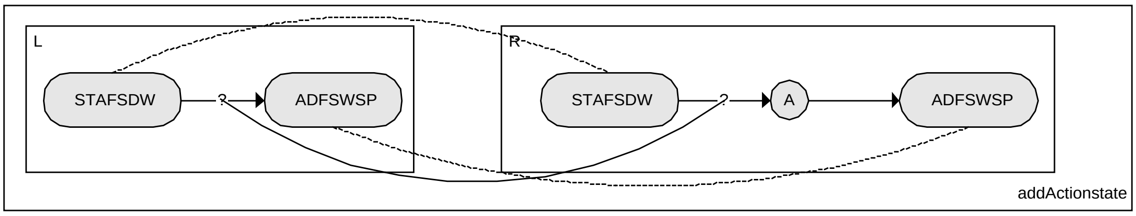
Einfügen eines impliziten Aktionszustandes in die Schleife des Schleifensymbols



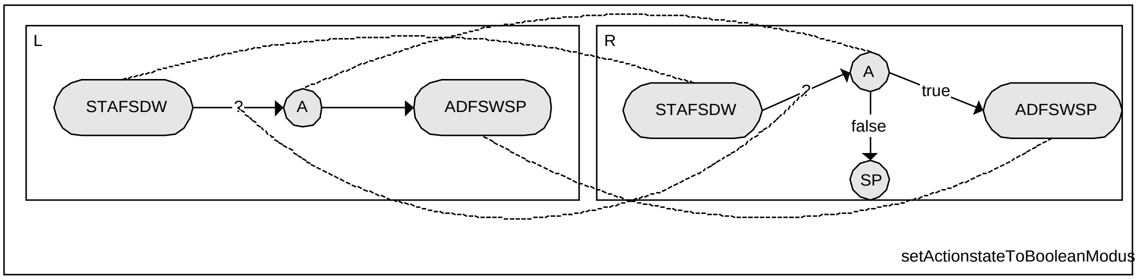
Einfügen eines Verzweigungssymbols in die Schleife des Schleifensymbols



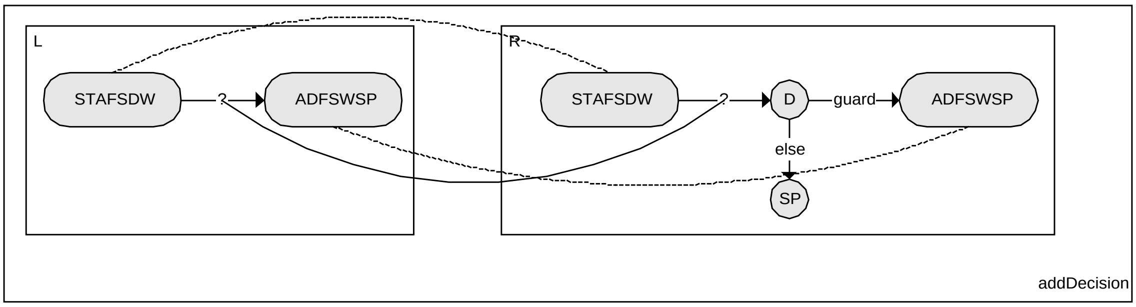
Einfügen eines Schleifensymbols in die Schleife des Schleifensymbols



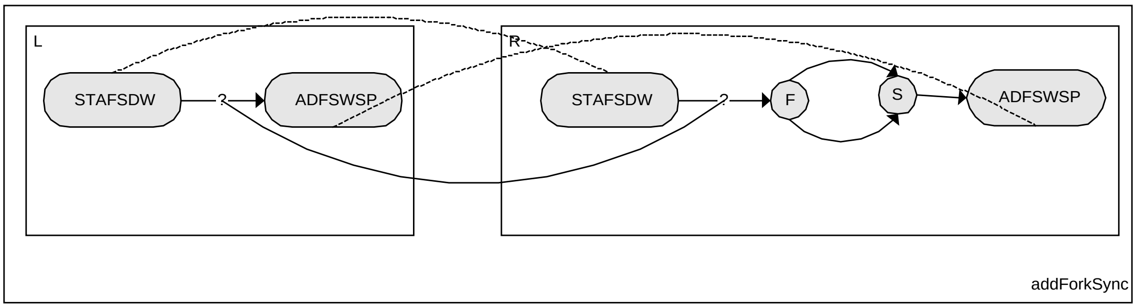
Einfügen eines impliziten Aktionszustandes



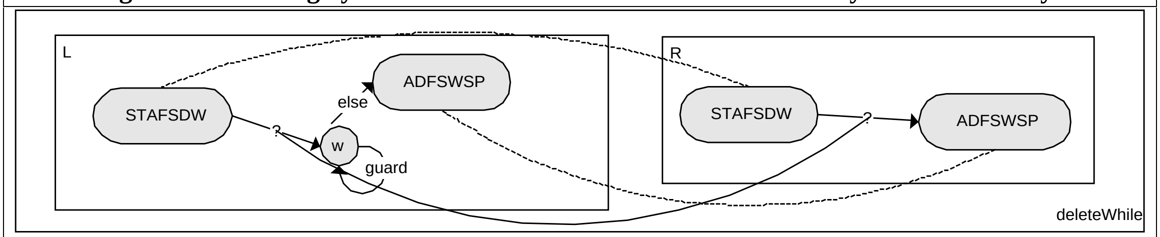
Überführen eines im impliziten Modus befindlichen Aktionszustandes in einen booleschen Modus



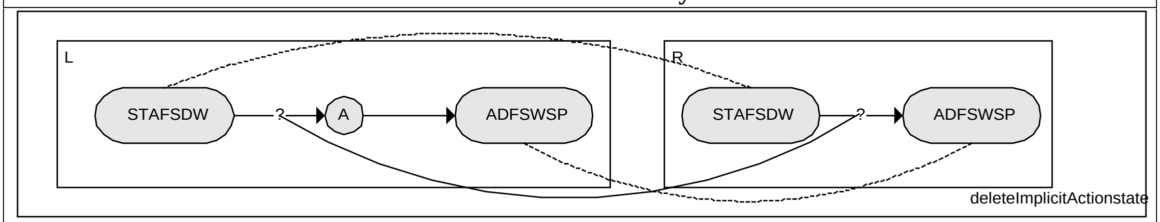
Einfügen eines Verzweigungssymbols



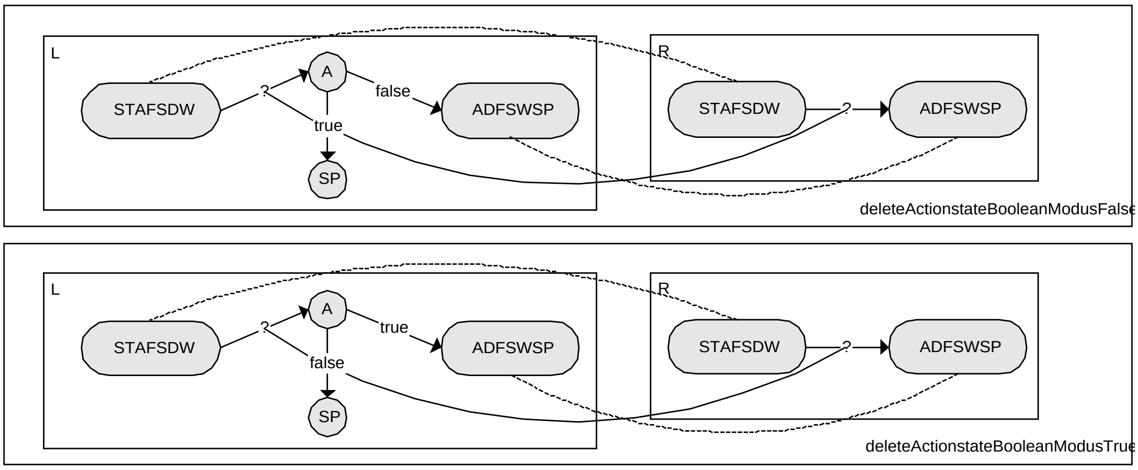
Einfügen des Gabelungssymbols zusammen mit dem abschließenden Synchronisationssymbol



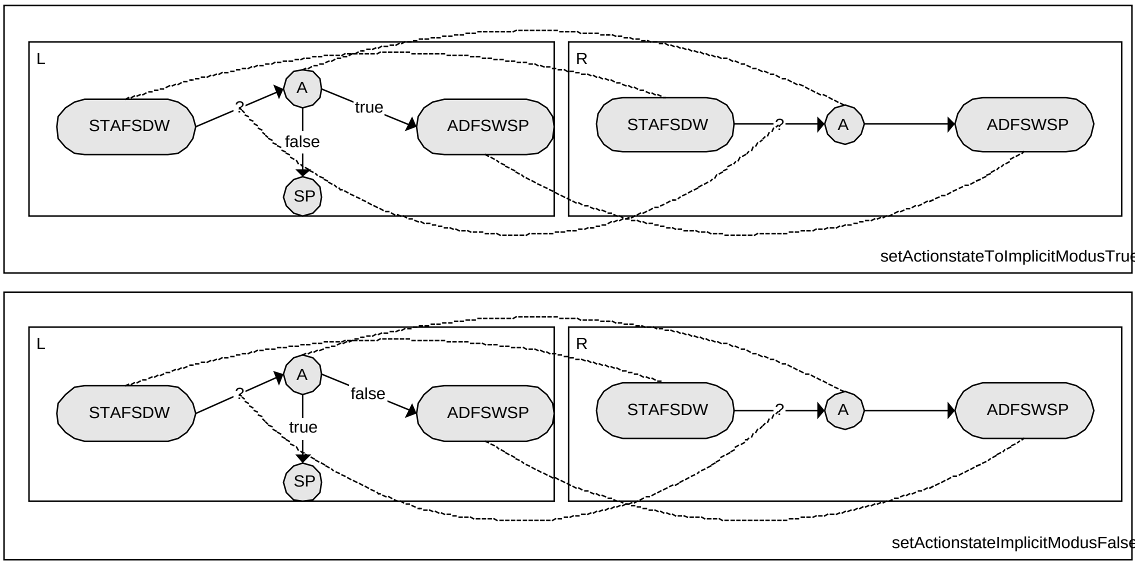
Löschen eines Schleifensymbols



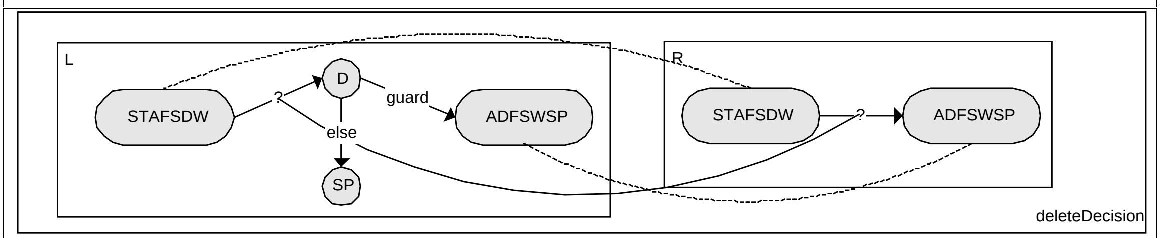
Löschen eines impliziten Aktionszustandes



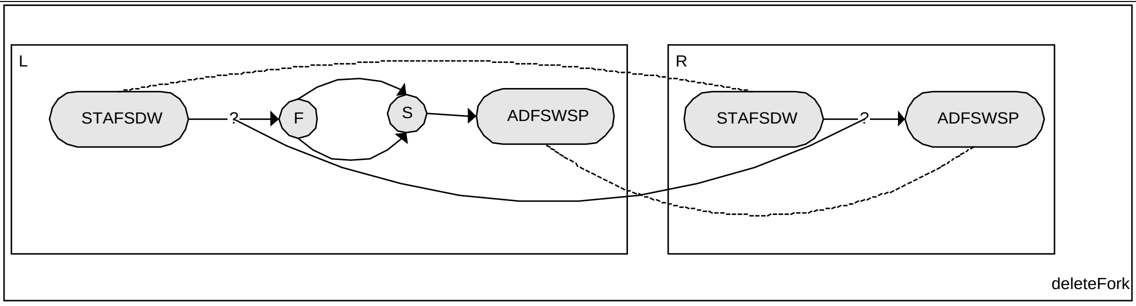
Löschen eines booleschen Aktionszustandes: 1. Fall: Die true – Kante wird gelöscht. 2. Fall: Die false – Kante wird gelöscht.



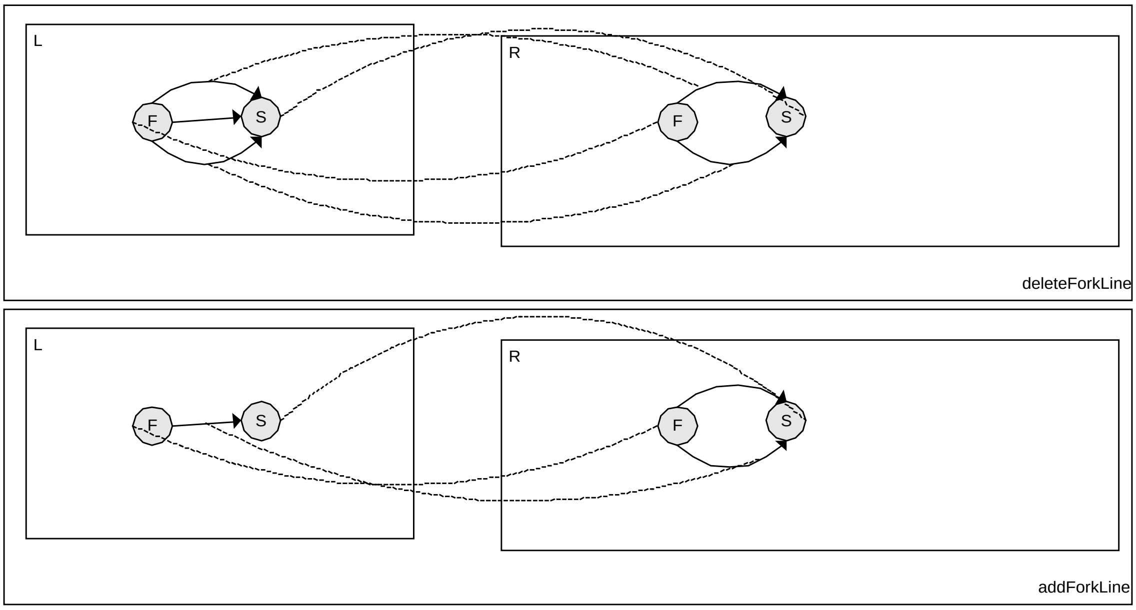
Überführen eines booleschen Aktionszustands in einen impliziten Modus: 1.Fall: Die false – Kante wird gelöscht. 2. Fall: Die true – Kante wird gelöscht.



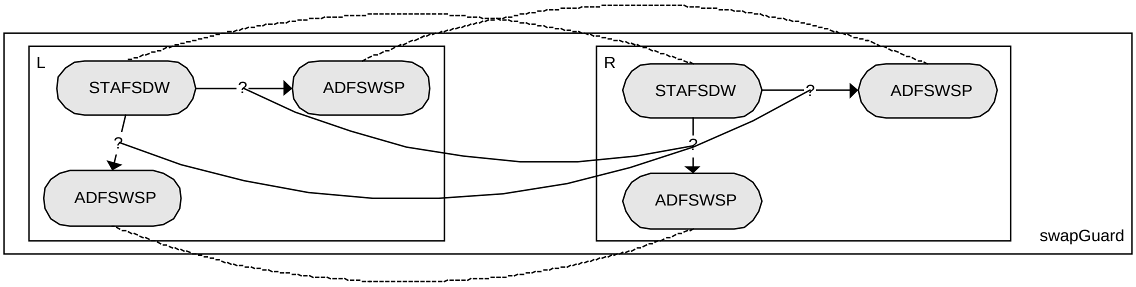
Löschen eines Verzweigungssymbols



Löschen des Gabelungssymbols zusammen mit dem abschließenden Synchronisationssymbol



Einfügen und Löschen von Gabelungen: Die Abbildung der linken Regelseite auf den Startgraphen muss für die Regel ‚deleteForkLine' injektiv sein, damit Verklebungen von Kanten ausgeschlossen werden.



Vertauschen der Wächterbedingung zweier ausgehender Kanten

6.3 Beispiel eines Javacodes zu Erzeugung einer Graphgrammatik

Der folgende dokumentierte Javacode zeigt, wie die Klasse aussieht, die eine kompilativ gesteuerte Kontrollkomponente intern erzeugt. Im Konstruktor dieser Klasse wird die an den Steuerungseditor übergebene Graphgrammatik über set- und get- Methoden neu erzeugt. Außerdem werden die Klassenmethoden für die Anwendbarkeit einer Regel (*isRuleApplicable*(*Rule*)) und für die Ausführung einer Regel (*executeRule*(*Rule*)) dem Benutzer vorimplementiert. Zur Kontrollierung der Graphersetzungen kann der Benutzer nun das *interface* Runnable implementieren, in dem er die Methode *run*() überschreibt.

Damit die Methode *run*() ausgeführt werden kann, muss ein nebenläufiges Programm implementiert werden, dem diese Klasse übergeben wird. Nebenläufige Programme leiten sich aus der Javaklasse *java.util.Thread* ab. Ihnen kann eine *interface* – Implementierung von *Runnable* übergeben werden. Dann führen sie die in der Methode *run*() spezifizierten Anweisungen aus.

In der Methode *run*() kann der Benutzer zur Steuerung von Graphersetzungen eine ganze Reihe von Anweisungen anwenden, wie WHILE – Schleifen oder bedingte Verzweigungen (if – then – else).

```
import java.util.Enumeration ;
import agg.xt_basis.Rule ;
import agg.xt_basis.Node ;
import agg.xt_basis.GraGra ;
import agg.xt_basis.Graph ;
import agg.xt_basis.Arc ;
import agg.xt_basis.BadMappingException ;
import agg.xt_basis.GraphObject ;
import agg.xt_basis.OrdinaryMorphism ;
import agg.editor.impl.EdGraGra ;
import agg.attribute.* ;
import agg.attribute.facade.* ;
import agg.attribute.handler.* ;
import agg.attribute.facade.impl.DefaultInformationFacade ;
import agg.alr.Type ;
import javacode.* ;
import aggjavacode.* ;


public class ControlRewrite implements Runnable
{
  GraGra   bgragra  = null ;
  EdGraGra edgragra = null ;
  Rule Rule0 = null ;
  Rule Rule1 = null ;
  Rule Rule2 = null ;

  //
  // Der Konstruktor
  //    hier wird durch set – Methoden die Graphgrammatik
  //    erzeugt.

  public ControlRewrite ()
  {
    bgragra = new GraGra() ;
    bgragra.setName("Benutzerdefinierter Name der Transformation") ;


    InformationFacade infoFacade = null ;
    AttrType tupleType = null ;
    AttrHandler javaHandler = null ;
    AttrInstance tupleInstance = null ;
    AttrInstanceMember instMember = null ;


    infoFacade = DefaultInformationFacade.self() ;
    javaHandler = infoFacade.getJavaHandler() ;


    // create Types
    Type Type_0 = bgragra.createType() ;
    Type_0.setStringRepr("") ;
    // attribute members
```

```
tupleType = Type_0.getAttrType() ;
Type Type_1 = bgragra.createType() ;
Type_1.setStringRepr("") ;
// attribute members
tupleType = Type_1.getAttrType() ;
Type Type_2 = bgragra.createType() ;
Type_2.setStringRepr("") ;
// attribute members
tupleType = Type_2.getAttrType() ;
Type Type_3 = bgragra.createType() ;
Type_3.setStringRepr("") ;
// attribute members
tupleType = Type_3.getAttrType() ;
Type Type_4 = bgragra.createType() ;
Type_4.setStringRepr("") ;
// attribute members
tupleType = Type_4.getAttrType() ;
Type Type_5 = bgragra.createType() ;
Type_5.setStringRepr("") ;
// attribute members
tupleType = Type_5.getAttrType() ;
Type Type_6 = bgragra.createType() ;
Type_6.setStringRepr("") ;
// attribute members
tupleType = Type_6.getAttrType() ;
Type Type_7 = bgragra.createType() ;
Type_7.setStringRepr("") ;
// attribute members
tupleType = Type_7.getAttrType() ;
Type Type_8 = bgragra.createType() ;
Type_8.setStringRepr("") ;
// attribute members
tupleType = Type_8.getAttrType() ;
Type Type_9 = bgragra.createType() ;
Type_9.setStringRepr("") ;
// attribute members
tupleType = Type_9.getAttrType() ;
Type Type_10 = bgragra.createType() ;
Type_10.setStringRepr("") ;
// attribute members
tupleType = Type_10.getAttrType() ;
Type Type_11 = bgragra.createType() ;
Type_11.setStringRepr("") ;
// attribute members
tupleType = Type_11.getAttrType() ;
Type Type_12 = bgragra.createType() ;
Type_12.setStringRepr("") ;
// attribute members
tupleType = Type_12.getAttrType() ;
Type Type_13 = bgragra.createType() ;
Type_13.setStringRepr("") ;
// attribute members
tupleType = Type_13.getAttrType() ;
Type Type_14 = bgragra.createType() ;
Type_14.setStringRepr("") ;
// attribute members
tupleType = Type_14.getAttrType() ;
Type Type_15 = bgragra.createType() ;
Type_15.setStringRepr("") ;
// attribute members
tupleType = Type_15.getAttrType() ;
Type Type_16 = bgragra.createType() ;
Type_16.setStringRepr("") ;
// attribute members
tupleType = Type_16.getAttrType() ;
Type Type_17 = bgragra.createType() ;
```

```
Type_17.setStringRepr("") ;
// attribute members
tupleType = Type_17.getAttrType() ;
Type Type_18 = bgragra.createType() ;
Type_18.setStringRepr("") ;
// attribute members
tupleType = Type_18.getAttrType() ;
Type Type_19 = bgragra.createType() ;
Type_19.setStringRepr("") ;
// attribute members
tupleType = Type_19.getAttrType() ;
Type Type_20 = bgragra.createType() ;
Type_20.setStringRepr("") ;
// attribute members
tupleType = Type_20.getAttrType() ;
Type Type_21 = bgragra.createType() ;
Type_21.setStringRepr("") ;
// attribute members
tupleType = Type_21.getAttrType() ;
Type Type_22 = bgragra.createType() ;
Type_22.setStringRepr("") ;
// attribute members
tupleType = Type_22.getAttrType() ;
Type Type_23 = bgragra.createType() ;
Type_23.setStringRepr("") ;
// attribute members
tupleType = Type_23.getAttrType() ;
Type Type_24 = bgragra.createType() ;
Type_24.setStringRepr("") ;
// attribute members
tupleType = Type_24.getAttrType() ;
Type Type_25 = bgragra.createType() ;
Type_25.setStringRepr("") ;
// attribute members
tupleType = Type_25.getAttrType() ;
Type Type_26 = bgragra.createType() ;
Type_26.setStringRepr("") ;
// attribute members
tupleType = Type_26.getAttrType() ;
Type Type_27 = bgragra.createType() ;
Type_27.setStringRepr("") ;
// attribute members
tupleType = Type_27.getAttrType() ;


// Create Graph: startgraph
Graph startgraph = bgragra.getGraph() ;
startgraph.setName("Graph") ;
// Create GraphObjects
Node startgraph_go_0 = startgraph.createNode(Type_0) ;
// attribute
tupleInstance = startgraph_go_0.getAttribute() ;
Node startgraph_go_1 = startgraph.createNode(Type_0) ;
// attribute
tupleInstance = startgraph_go_1.getAttribute() ;
Node startgraph_go_2 = startgraph.createNode(Type_0) ;
// attribute
tupleInstance = startgraph_go_2.getAttribute() ;
Node startgraph_go_3 = startgraph.createNode(Type_0) ;
// attribute
tupleInstance = startgraph_go_3.getAttribute() ;
  Arc startgraph_go_4 =
              startgraph.createArc(Type_1,startgraph_go_1,start
            graph_go_1) ;
// attribute
tupleInstance = startgraph_go_4.getAttribute() ;
```

```
Arc startgraph_go_5 =
                startgraph.createArc(Type_1,startgraph_go_3,start
              graph_go_3) ;
// attribute
tupleInstance = startgraph_go_5.getAttribute() ;
Node startgraph_go_6 = startgraph.createNode(Type_0) ;
// attribute
tupleInstance = startgraph_go_6.getAttribute() ;
Arc startgraph_go_7 =
       startgraph.createArc(Type_1,startgraph_go_6,startgraph_go_6
     ) ;
// attribute
tupleInstance = startgraph_go_7.getAttribute() ;
Arc startgraph_go_8 =
       startgraph.createArc(Type_1,startgraph_go_6,startgraph_go_1
     ) ;
// attribute
tupleInstance = startgraph_go_8.getAttribute() ;
Arc startgraph_go_9 =
         startgraph.createArc(Type_1,startgraph_go_1,startgraph_g
       o_3) ;
// attribute
tupleInstance = startgraph_go_9.getAttribute() ;
Arc startgraph_go_10 =
         startgraph.createArc(Type_1,startgraph_go_3,startgraph_g
       o_2) ;
// attribute
tupleInstance = startgraph_go_10.getAttribute() ;
Arc startgraph_go_11 =
         startgraph.createArc(Type_1,startgraph_go_0,startgraph_g
       o_2) ;
// attribute
tupleInstance = startgraph_go_11.getAttribute() ;


// Create Rule: Rule0
Rule0 = bgragra.createRule() ;
Rule0.setName("Rule0") ;
// Create Graph: Rule0_leftgraph
Graph Rule0_leftgraph = Rule0.getLeft() ;
Rule0_leftgraph.setName("Left.graph") ;
// Create GraphObjects
Node Rule0_leftgraph_go_0 = Rule0_leftgraph.createNode(Type_0) ;
// attribute
tupleInstance = Rule0_leftgraph_go_0.getAttribute() ;
// Create Graph: Rule0_rightgraph
Graph Rule0_rightgraph = Rule0.getRight() ;
Rule0_rightgraph.setName("Right.graph") ;
// Create GraphObjects
Node Rule0_rightgraph_go_0 = Rule0_rightgraph.createNode(Type_0) ;
// attribute
tupleInstance = Rule0_rightgraph_go_0.getAttribute() ;
// add mappings
try
{
}
catch(BadMappingException bmx) {}
// Create NAC: Rule0_nac0
OrdinaryMorphism Rule0_nac0 = Rule0.createNAC() ;
Rule0_nac0.setName("nac0") ;
// Create Graph: Rule0_nac0_imagegraph
Graph Rule0_nac0_imagegraph = Rule0_nac0.getImage() ;
Rule0_nac0_imagegraph.setName("Nac0") ;
// Create GraphObjects
Node Rule0_nac0_imagegraph_go_0 =
         Rule0_nac0_imagegraph.createNode(Type_0) ;
```

```
// attribute
tupleInstance = Rule0_nac0_imagegraph_go_0.getAttribute() ;
// add mappings
try
{
}
catch(BadMappingException bmx) {}
// Create NAC: Rule0_nac1
OrdinaryMorphism Rule0_nac1 = Rule0.createNAC() ;
Rule0_nac1.setName("nac1") ;
// Create Graph: Rule0_nac1_imagegraph
Graph Rule0_nac1_imagegraph = Rule0_nac1.getImage() ;
Rule0_nac1_imagegraph.setName("Nac1") ;
// Create GraphObjects
Node Rule0_nac1_imagegraph_go_0 =
            Rule0_nac1_imagegraph.createNode(Type_0) ;
// attribute
tupleInstance = Rule0_nac1_imagegraph_go_0.getAttribute() ;
// add mappings
try
{
}
catch(BadMappingException bmx) {}


// Create Rule: Rule1
Rule1 = bgragra.createRule() ;
Rule1.setName("Rule1") ;
// Create Graph: Rule1_leftgraph
Graph Rule1_leftgraph = Rule1.getLeft() ;
Rule1_leftgraph.setName("Left.graph") ;
// Create GraphObjects
Node Rule1_leftgraph_go_0 = Rule1_leftgraph.createNode(Type_0) ;
// attribute
tupleInstance = Rule1_leftgraph_go_0.getAttribute() ;
// Create Graph: Rule1_rightgraph
Graph Rule1_rightgraph = Rule1.getRight() ;
Rule1_rightgraph.setName("Right.graph") ;
// Create GraphObjects
Node Rule1_rightgraph_go_0 = Rule1_rightgraph.createNode(Type_0) ;
// attribute
tupleInstance = Rule1_rightgraph_go_0.getAttribute() ;
// add mappings
try
{
}
catch(BadMappingException bmx) {}
// Create NAC: Rule1_nac0
OrdinaryMorphism Rule1_nac0 = Rule1.createNAC() ;
Rule1_nac0.setName("nac0") ;
// Create Graph: Rule1_nac0_imagegraph
Graph Rule1_nac0_imagegraph = Rule1_nac0.getImage() ;
Rule1_nac0_imagegraph.setName("Nac0_") ;
// Create GraphObjects
Node Rule1_nac0_imagegraph_go_0 =
       Rule1_nac0_imagegraph.createNode(Type_0) ;
// attribute
tupleInstance = Rule1_nac0_imagegraph_go_0.getAttribute() ;
Node Rule1_nac0_imagegraph_go_1 =
       Rule1_nac0_imagegraph.createNode(Type_0) ;
// attribute
tupleInstance = Rule1_nac0_imagegraph_go_1.getAttribute() ;
Node Rule1_nac0_imagegraph_go_2 =
       Rule1_nac0_imagegraph.createNode(Type_0) ;
// attribute
tupleInstance = Rule1_nac0_imagegraph_go_2.getAttribute() ;
```

```
    // add mappings
    try
    {
    }
    catch(BadMappingException bmx) {}


    // Create Rule: Rule2
    Rule2 = bgragra.createRule() ;
    Rule2.setName("Rule2") ;
    // Create Graph: Rule2_leftgraph
    Graph Rule2_leftgraph = Rule2.getLeft() ;
    Rule2_leftgraph.setName("Left.graph") ;
    // Create GraphObjects
    Node Rule2_leftgraph_go_0 = Rule2_leftgraph.createNode(Type_0) ;
    // attribute
    tupleInstance = Rule2_leftgraph_go_0.getAttribute() ;
    // Create Graph: Rule2_rightgraph
    Graph Rule2_rightgraph = Rule2.getRight() ;
    Rule2_rightgraph.setName("Right.graph") ;
    // Create GraphObjects
    Node Rule2_rightgraph_go_0 = Rule2_rightgraph.createNode(Type_0) ;
    // attribute
    tupleInstance = Rule2_rightgraph_go_0.getAttribute() ;
    Node Rule2_rightgraph_go_1 = Rule2_rightgraph.createNode(Type_0) ;
    // attribute
    tupleInstance = Rule2_rightgraph_go_1.getAttribute() ;
    Node Rule2_rightgraph_go_2 = Rule2_rightgraph.createNode(Type_0) ;
    // attribute
    tupleInstance = Rule2_rightgraph_go_2.getAttribute() ;
    // add mappings

    edgragra = new EdGraGra() ;
    try
    {
    }
    catch(BadMappingException bmx) {}
  }

  //Die folgenden implementieren die Schnittstelle zur Graphersetzung.
  // Die Anwendbarkeit einer Methode überprüfen:
  // Eine Regel ausführen:
  //          boolean     executeRegelname() ;
  // Eine Regel auf Anwendbarkeit überprüfen
  //          boolean     isRegelnameApplicable() ;
  // Die Regeln dieser Graphgrammatik heißen Rule0, Rule1 und Rule2
  public boolean rule0IsApplicable()
  {
    return edgragra.isApplicable(Rule0) ;
  }


  public void execRule0()
  {
    return edgragra.executeRule(Rule0) ;
  }


  public boolean rule1IsApplicable()
  {
    return edgragra.isApplicable(Rule1) ;
  }


  public void execRule1()
  {
```

```
    return edgragra.executeRule(Rule1) ;
  }


  public boolean rule2IsApplicable()
  {
    return edgragra.isApplicable(Rule2) ;
  }


  public void execRule2()
  {
    return edgragra.executeRule(Rule2) ;
  }

  // Implementierung der Methode run() aus dem Interface
  //  Runnable. Hier findet die Graphersetungskontrolle durch
  //  den Benutzer statt.

  public void run()
  {
    // Implementierung einer Bedingten Verzweigung

    if ( rule2IsApplicable )
      {
        execRule2() ;
      } else
        {
            // Implementierung einer WHILE - Schleife
            while ( rule0IsApplicable && rule1IsApplicable())
                {
                    execRule1() ;
                    execRule0() ;
                }
        } // ende else

  } // ende run()
}// ende Klasse Graphgrammatik
```

6.4 Api der Kontrollkomponente

Im folgenden wird die API – Dokumentation der Kontrollkomponente vorgelegt. Auf die Einbringung der Klassen *JActivity*, *JStart*, *JStop*, *JWhile* und JDecision wird verzichtet, da sie lediglich die *paintComponent*(*Graphics g*) – Methode überschreiben.
Zuerst werden die Klassen der Bibliothek *controlshapes* aufgelistet, danach die Klasse *Automat* aus der Bibliothek *automat*.

6.4.1 Controlshapes

Die folgenden Klassen findet man in der Bibliothek *controlshapes*.

6.4.1.1 AccessPoint

public class **AccessPoint**
extends java.lang.Object

Diese Klasse repräsentiert einen Zugangspunkt. Kanten definieren sich durch zwei Zugangspunkte (Quell- und Zielpunkt), die von Zugangscontainern (controlshapes.CoastContainer) vergeben oder angefordert werden können.

See Also:
`CoastContainer`

Constructor Detail

AccessPoint

`public AccessPoint(CoastContainer coastContainer)`

Erzeugung eines Zugangspunktes durch Übergabe des ihm Zugewiesenen Zugangscontainers.
Parameters:
`CoastContainer` - Der Zugangscontainer, der diesen Zugangspunkt freigeben darf.

Method Detail

getX

`public int getX()`

Die x - Komponente eines Zugangspunktes wird zurückgegeben.
Returns:
int Die x - Koordinate eines Zugangspunktes.

---

getY

`public int getY()`

Die y - Komponente eines Zugangspunktes wird zurückgegeben.
Returns:
int Die y - Koordinate eines Zugangspunktes.

---

getCoastContainer

`public CoastContainer getCoastContainer()`

Der Zugangscontainer, zu der dieser Zugangspunkt gehört, wird zurückgegeben.
Returns:
CoastContainer - Zugangsbehälter

---

use

`public void use()`

Setzt einen Zugangspunkt in den Status "wird benutzt".

---

retrieve

`public void retrieve()`

Freigabe dieses Zugangspunktes.

---

isInUse

`public boolean isInUse()`

Testet ob dieser Zugangspunkt vergeben wurde oder nicht.
Returns:
true - Zugangspunkt ist vergeben.

6.4.1.2 CoastContainer

controlshapes
Class CoastContainer

```
java.lang.Object
  |
```

```
  +--controlshapes.CoastContainer
```

---

public class **CoastContainer**
extends java.lang.Object

Diese Klasse implementiert einen Zugangscontainer (o. auch Zugangsbehälter), der die Anforderung u. Freigabe von Zugangspunkten realisiert. Es können Zugangspunkte angefordert werden. Wird ein Zugangspunkt nicht mehr verwendet, so wird er durch diese Klasse wieder freigegeben.

Constructor Detail

CoastContainer

```
public CoastContainer(BaseShape bs,
                      int size,
                      java.lang.String direction)
```

Zugangsbehälter werden mit der Anzahl der Zugangspunkte, die sie freigeben können instanziiert. Außerdem werden Editierrichtung und die Referenz der Klasse übergeben, zu der dieser Zugangscontainer gehört.

Parameters:
`bs` - Besitzer dieses Zugangscontainers.
`size` - Anzahl der anforderbaren Zugangspunkte.
`direction` - Editierrichtung.

Method Detail

isUnused

```
public boolean isUnused()
```

Hat ein Zugangscontainer mindestens einen Zugangspunkt vergeben, so wird er benutzt.

Returns:
true -> Zugangsbehälter wird benutzt.

false -> Zugangsbehälter wird nicht benutzt.

---

isAccessPointAvailable

```
public boolean isAccessPointAvailable()
```

Stellt fest, ob dieser Zugangsbehälter noch einen freien Zugangspunkt hat.

Returns:
true Mindestens ein Zugangspunkt kann vergeben werden.

false Alle Zugangspunkte sind vergeben.

---

getAccessPoint

```
public AccessPoint getAccessPoint()
```

Vergibt einen Zugangspunkt.

Returns:
AccessPoint Zugangspunkt, der vergeben wird.

---

retrieveAccessPoint

```
public void retrieveAccessPoint(AccessPoint accessPoint)
```

Meldet einen Zugangspunkt wieder an.

Parameters:
`accessPoint` - Zugangspunkt.

---

retrieveAllAccessPoints

`public void **retrieveAllAccessPoints**()`

Alle Zugangspunkte wieder zurückrufen.

---

getBaseShape

`public BaseShape **getBaseShape**()`

Gibt den Besitzer dieses Zugangscontainers zurück.

---

getDirection

`public java.lang.String **getDirection**()`

Gibt die Editierrichtung zurück.

---

getSize

`public int **getSize**()`

Gibt die Anzahl der Zugangspunkte zurück.

---

getIndexOfAccessPoint

`protected int **getIndexOfAccessPoint**(AccessPoint ap)`

Gibt den Index des Zugangspunktes in der internen Liste dieser Klasse zurück.

6.4.1.3 DiagramFeature

---

controlshapes

Class DiagramFeature

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPanel
                          |
                          +--controlshapes.DiagramFeature
```

---

public class **DiagramFeature**
extends javax.swing.JPanel

Ein DiagramFeature realisiert die Vergrößerungsfunktion von Steuerungsobjekten. Per Maus kann die dargestellte Größe verändert werden. Dazu wird ein Rahmen angeboten (ähnlich wie bei der Fensterdarstellung von vielen Betriebssystemen).

Der Steuerungseditor erhält diese Klasse, um Steuerungsobjekte darzustellen. Die grafische Repräsentation von Steuerungsobjekten übernehmen Klassen, die sich von BaseJComponent ableiten. Der DiagramFeature umhüllt diese Komponenten.

See Also:
Serialized Form

Field Detail

x

`public int` **`x`**

X - Koordinate dieser Komponente.

---

y

`public int` **`y`**

Y - Koordinate dieser Komponente.

---

width

`public int` **`width`**

Breite dieser Komponente.

---

height

`public int` **`height`**

Höhe dieser Komponente

Constructor Detail

DiagramFeature

```
public DiagramFeature(BaseShape bs,
                      int x,
                      int y,
                      int width,
                      int height)
```

Erzeugung eines DiagramFeature's.
Parameters:
`bs` - Die Refernz des Steuerungsobjekts, das dieses DiagramFeature besitzt.
`x` - X - Koordinate dieser Komponente.
`y` - Y - Koordinate dieser Komponente.
`width` - Breite dieser Komponente.
`height` - Höhe dieser Komponente.

DiagramFeature

```
public DiagramFeature(BaseShape bs)
```

Erzeugung eines DiagramFeature's.
Parameters:
`bs` - Die Refernz des Steuerungsobjekts das dieses DiagramFeature besitzt.

Method Detail

paintComponent

```
public void paintComponent(java.awt.Graphics g)
```

Zeichnet einen Vergrößerungsrahmen.
Overrides:
paintComponent in class javax.swing.JComponent

---

getInsets

`public java.awt.Insets` **`getInsets`**`()`

Gibt die Ausrichtung eines DiagramFeature's zurück.
Overrides:
getInsets in class javax.swing.JComponent
Returns:
Insets Die grafische Ausrichtung dieses DiagramFeatures.

---

getFrameWidth

`public int` **`getFrameWidth`**`()`

Gibt die Breite des Vergrößerungsrahmen zurück.
Returns:
int Breite des Vergrößerungsrahmen.

---

getCornerWidth

`public int` **`getCornerWidth`**`()`

Gibt die Seitenbreite des Quadrates zurück, in dem das DiagramFeature diagonal verkleinert oder vergrößert werden kann.
Returns:
int Seitenbreite des Quadrates.

---

setResizeFrame

`public void` **`setResizeFrame`**`()`

Zeigt den Vergrößerungsrahmen an.

---

setResizeFrame

`public void` **`setResizeFrame`**`(java.awt.Color color)`

Die Farbe des Vergrößerungsrahmen wird neu gesetzt.
Parameters:
`color` - Die Farbe des Vergrößerungsrahmen.

---

removeResizeFrame

`public void` **`removeResizeFrame`**`()`

Versteckt den Vergrößerungsrahmen.

---

isResizeFrameVisible

`public boolean` **`isResizeFrameVisible`**`()`

Der Test, ob der Vergrößerungsrahmen zu sehen ist.
Returns:
boolean - true --> Vergrößerungsrahmen ist zu sehen.

false --> Vergrößerungsrahmen ist nicht zu sehen.

---

getResizeListener

`public ResizeListener` **`getResizeListener`**`()`

Rückgabe der Referenzklasse, die Mausevents implementiert.
Returns:
ResizeListener Referenzklasse für Mausevents

---

resizeFeature

```
public void resizeFeature(int width,
                          int height)
```

Setzt eine neu Ausdehnung des DiagramFeature's fest.
Parameters:
`width` - Die neue Breite.
`height` - Die neue Höhe.

6.4.1.4 BaseJComponent

controlshapes
Class BaseJComponent

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPanel
                          |
                          +--controlshapes.BaseJComponent
```

Direct Known Subclasses:
JActivity, JDecision, JStart, JStop, JTransitionLabel, JWhile

---

public class **BaseJComponent**
extends javax.swing.JPanel

Diese Klasse implementiert das grafische Aussehen der Steuerungsobjekte. Alle Steuerungsobjekte, die sich von dieser Klasse ableiten, erhalten einen *default*- und *highlight* - Hintergrund. Der *default* - Hintergrund zeigt an, dass ein Steuerungsobjekt weder selektiert ist noch vom Automaten interpretiert wird.

Wird ein Steuerungsobjekt durch den Benutzer selektiert, so wird dieser im *highlight* Modus dargestellt. Das selbe gilt für die Steuerungsobjekte, die gerade vom Automaten interpretiert werden. Ist die Interpretation eines Steuerungsobjekts beendet wird es wieder im *default* Modus *dargestell*.

Für den Fall, dass der Hintergrund nicht dargestellt werden soll, ist die Möglichkeit berücksichtigt worden, dass Steuerungsobjekte opaque dargestellt werden. In diesem Fall werden dann nur noch die durch die Methode paintComponente(Graphics g), der JComponent Basisklasse, implementierten Grafikkontexte dargestellt. Diese Klasse implementiert die paintComponent(Graphics g) Methode nicht. Alle Steuerungsobjekte besitzen jedoch Komponenten, die sich aus dieser Klasse ableiten und die paintComponent(Graphics g) Methode implementieren.

Constructor Detail

BaseJComponent

```
public BaseJComponent(BaseShape bs)
```

Durch die Übergabe der Basisklasse für Steuerungsobjete (BaseShape) werden Zugehörigkeiten der dargestellten Grafikobjekte leicht erkennbar.
Parameters:
`BaseShape` - - Beitzer dieser Komponente.

Method Detail

paintComponent

`public void paintComponent(java.awt.Graphics g)`

Overrides:
paintComponent in class javax.swing.JComponent

---

setDefaultModus

`public void setDefaultModus()`

Stellt ein Steuerungsobjekt in der Standardhintergrundfarbe dar.

---

setHighlightModus

`public void setHighlightModus()`

Stellt ein Steuerungsobjekt in der "selektiert" - Hintergrundfarbe dar.

---

setDefaultModus

`public void setDefaultModus(java.awt.Color color)`

Setzt eine andere Farbe als standard Hintergrundgfarbe fest.
Parameters:
`Color` - Neue standard Hintergrundfarbe.

---

setHighlightModus

`public void setHighlightModus(java.awt.Color color)`

Setzt eine andere Farbe für den "selektiert" - Modus fest.
Parameters:
`Color` - Neue Hintergrundfarbe.

---

hideBorder

`public void hideBorder()`

Das Steuerungsobjekt wird opaque dargestell.

---

showBorder

`public void showBorder()`

Macht den Hintergrund wieder sichtbar.

---

isBorderVisible

`public boolean isBorderVisible()`

Gibt zurück, ob der Hintergrund sichtbar ist oder nicht.
Returns:
boolean - true --> Der Hintergrund ist sichtbar.

false --> Der Hintergrund ist nicht sichtbar.

---

isHighlighted

`public boolean isHighlighted()`

Gibt zurück, ob das Steuerungsobjekt im selektierten Modus dargestellt wird oder nicht.
Returns:
boolean - true --> Das Steuerungsobjekt ist selektiert.

false --> Das Steuerungsobjekt ist nicht selektiert.

---

select

`public void` **`select`**`()`

Standardfunktion zum Wechseln vom default- in den highlight- Modus und und wieder zurück.

---

getBaseShape

`public BaseShape` **`getBaseShape`**`()`

Gibt das Steuerungsobjekt wieder zurück, dass diese Komponente besitzt.
Returns:
BaseShape - Beitzer dieser Komponente.

---

getDiagramFeature

`public DiagramFeature` **`getDiagramFeature`**`()`

Diese Methode gibt den Container zurück, zu der diese Klasse gehört.
Returns:
DiagramFeature - Der grafische Container zu dieser Komponente.

---

setDiagramFeature

`public DiagramFeature` **`setDiagramFeature`**`(DiagramFeature df)`

Fügt diese Komponente in einen grafischen Container ein.
Returns:
DiagramFeature Der grafische Container zu dieser Komponente.

6.4.1.5 DiagramContainer

Package **Class** **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS   FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD   DETAIL: FIELD | CONSTR | METHOD

---

controlshapes
Class DiagramContainer

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPanel
                          |
                          +--controlshapes.DiagramContainer
```

---

public class **DiagramContainer**
extends javax.swing.JPanel

Diese Klasse verwaltet die grafische Darstellung des Steuerungsgraphen. Es können nur Kanten eingefügt werden. Dabei muss das Quellobjekt dieser Kante schon in diesem Container verankert sein. Das Zielobjekt wird dann neu eingefügt, sofern es nicht schon verankert ist.

Als Schnittstelle zu AGG ist die Klasse GraGraEditor gewählt worden, von der die Namen der Produktionsregeln extrahiert werden. Braucht eines der Steuerungsobjekte diese Schnittstelle, so kann es mit der Methode getGraGraEditor() von jedem Steuerungsobjekt angefordert werden.

Aufgrund dieser Eigenschaften kann diese Klasse als Steuerungseditor bezeichnet werden. Grafisch gesehen ist der Steuerungseditor lediglich eine Swingkomponente.

Field Detail

components

`protected java.util.Vector` **`components`**

Die Liste der grafischen Komponenten der dargestellten Steuerungsobjekte

Constructor Detail

DiagramContainer

```
public DiagramContainer(java.awt.Frame frame,
                        agg.gui.GraGraEditor editor)
```

Ein DiagramContainer wird erzeugt, indem das Fenster, das ihn darstellen soll, und die Schnittstelle zu AGG übergeben wird.

Parameters:

`frame` - Das diesen Container darstellende Fenster.

`editor` - Die Referenz zum AGG Hauptfenster.

Method Detail

getContentEdGraGra

`public agg.editor.impl.EdGraGra` **`getContentEdGraGra`**`()`

Gibt die Graphgrammatik zurück, die der Steuerungsgraph als Basis verwendet.

Returns:

EdGraGra - Graphgrammatikreferenz dieses Steuerungseditors.

---

getFrame

`public java.awt.Frame` **`getFrame`**`()`

Gibt das Fenster zurück, das diesen Container darstellt.

Returns:

Frame Das diesen Container darstellende Fenster.

---

paintComponent

`public void` **`paintComponent`**`(java.awt.Graphics g)`

Hier werden die Kanten gezeichnet. Ziel- und Quellpunkte werden durch die Zugangspunkte, der in diesem Container enthaltenen Steuerungsobjekte, definiert.

Overrides:

paintComponent in class javax.swing.JComponent

Parameters:

`g` - Der Grafikkontext dieses Containers.

---

updatePositionOfLabels

`protected void` **`updatePositionOfLabels`**`()`

Sorgt dafür, dass die Wächter neu positioniert werden.

---

setStartShape

`public void **setStartShape**(Start start)`

Setzt den Beginn des Steuerungsgraphen fest.
Parameters:
`start` - Das Startsymbol.

---

getStartShape

`public Start **getStartShape**()`

---

addFeature

`public void **addFeature**(Transition transition)`

Fügt ein neues Steuerungsobjekt hinzu, in dem es die eingehende Kante dieses Steuerungsobjekts erhält.
Parameters:
`transition` - Die Kante, die das neue Steuerungsobjekt als Ziel enthält.

---

deleteFeature

`protected void **deleteFeature**(BaseShape bs)`

Löscht ein Steuerungsobjekt. Ein- und ausgehende Kanten müssen vorher in der Klasse BaseShape gelöscht worden sein.
Parameters:
`Baseshape` - Das zu löschende Objekt.

6.4.1.6 BaseInterface

controlshapes

Interface BaseInterface
All Known Implementing Classes:
BaseShape

---

public interface **BaseInterface**

Die Grundfunktionen, die ein Steuerungsobjekt erfüllen muss, wird von dieser Schnittstelle spezifiziert. Neben der Implementierung der Einfüge- und Löschoperation wird durch diese Schnittstelle das Modell der aktivierten Kanten protokollarisch festgelegt.

Alle Steuerungsobjekte erben diese Schnittstelle. Somit wird sichergestellt, dass der Steuerungsgraph nur aus Steuerungsobjekten besteht, die ein gemeinsames Einfüge- und Löschprotokoll einhalten.

Method Detail

addObject

`public void **addObject**(java.lang.Object obj)`

Fügt das übergebene Objekt in die aktivierte Kante des Steuerungsobjektes ein.
Parameters:
`Objekt` - Das einzufügende Steuerungsobjekt.

---

deleteObject

`public void deleteObject()`

Wird aufgerufen, um das Steuerungsobjekt zu löschen.

---

getActiveTransition

`public Transition getActiveTransition()`

Rückgabe der Kante, die aktiviert wurde. Sollte ein Steuerungsobjekt eingefügt werden, so wird das zwischen Quell- und Zielobjekt dieser Kante eingefügt.

Returns:

Transition - Die aktivierte Kante.

---

setActiveTransition

`public void setActiveTransition(Transition transition)`

Aktiviert eine ausgehende Kante für die Löschoperation.

Parameters:

`Transition` - Die zu aktivierende Kante.

---

updateDialog

`public void updateDialog()`

Alle Steuerungsobjekte besitzen Kontextmenüs, die durch diese Operation aktualiesiert werden sollen.

6.4.1.7 BaseShape

controlshapes

Class BaseShape

```
java.lang.Object
  |
  +--controlshapes.BaseShape
```

Direct Known Subclasses:

Activity, Decision, Start, Stop, Transition, While

---

public class **BaseShape**
extends java.lang.Object
implements BaseInterface

Diese Klasse ist als Basisklasse aller Steuerungsobjekte zu verstehen. Hier werden Methoden implementiert, die für alle bisher realiesierten Steuerungsobjekte verwendet werden können. Als Protokoll wird die Schnittstelle BaseInterface implementiert.

Außerdem werden Methoden bereitgestellt, mit denen man die Anzahl der ein- und ausgehenden Kanten bestimmen kann.

Steuerungsobjekte verwalten lediglich ein- und ausgehende Kanten. Ziel- und Quellknoten hingegen werden beim Erzeugen einer Kante verwaltet.

Field Detail

type

`public static java.lang.String type`

Der Typname dieses Steuerungsobjekts

---

bjc

`protected BaseJComponent` **`bjc`**

Die grafische Komponente dieses Steuerungsobjekts

---

activeTransitionsGuard

`protected java.lang.String` **`activeTransitionsGuard`**

Die zur Zeit aktivierte Kante

---

jpopupMenu

`public javax.swing.JPopupMenu` **`jpopupMenu`**

Kontextmenü eines Steuerungsobjekts.

---

basicsJMenu

`public javax.swing.JMenu` **`basicsJMenu`**

Der basics - Abschnitt des Kontextmenüs.

---

privatesJMenu

`public javax.swing.JMenu` **`privatesJMenu`**

Der private Abschnitt des Kontextmenüs.

---

addJMenu

`public javax.swing.JMenu` **`addJMenu`**

Das Einfügemenü des Kontextmenüs.

---

deleteJItem

`public javax.swing.JMenuItem` **`deleteJItem`**

Die Löschoption des Kontextmenüs.

---

north

`public CoastContainer` **`north`**

Der nördliche Zugangsbehälter.

---

south

`public CoastContainer` **`south`**

Der südliche Zugangsbehälter.

---

west

`public CoastContainer` **`west`**

Der westliche Zugangsbehälter.

---

east

`public CoastContainer` **`east`**

Der östliche Zugangsbehälter.

Constructor Detail

BaseShape

`public BaseShape(DiagramContainer controller)`

Ein Steuerungsobjekt wird mit der Referenz des Containers erzeugt, der den Steuerungsgraphen verwaltet, zu der dieses Steuerungsobjekt gehört.

Parameters:

`controller` - Der Steuerungseditor zu diesem Steuerungsobjekt.

Method Detail

updateDialog

`public void updateDialog()`

implementing of interface 'ShapeBase'

Specified by:

updateDialog in interface BaseInterface

---

addObject

`public void addObject(java.lang.Object obj)`

Description copied from interface: BaseInterface

Fügt das übergebene Objekt in die aktivierte Kante des Steuerungsobjektes ein.

Specified by:

addObject in interface BaseInterface

Tags copied from interface: BaseInterface

Parameters:

`Objekt` - Das einzufügende Steuerungsobjekt.

---

deleteObject

`public void deleteObject()`

Description copied from interface: BaseInterface

Wird aufgerufen, um das Steuerungsobjekt zu Löschen.

Specified by:

deleteObject in interface BaseInterface

---

getActiveTransition

`public Transition getActiveTransition()`

Description copied from interface: BaseInterface

Rückgabe der Kante, die aktiviert wurde. Sollte ein Steuerungsobjekt eingefügt werden, so wird das zwischen Quell- und Zielobjekt dieser Kante eingefügt.

Specified by:

getActiveTransition in interface BaseInterface

Tags copied from interface: BaseInterface

Returns:

Transition - Die aktivierte Kante.

---

setActiveTransition

`public void setActiveTransition(Transition transition)`

Description copied from interface: BaseInterface

Aktiviert eine ausgehende Kante für die Löschoperation.

Specified by:

setActiveTransition in interface BaseInterface

Tags copied from interface: BaseInterface

Parameters:
`Transition` - Die zu aktivierende Kante.

---

getAccessPoint

`public AccessPoint **getAccessPoint**(java.lang.String direction)`

Fordert von einem Zugangsbehälter dieser Klasse einen Zugangspunkt an.
Parameters:
`direction` - Name des Zugangsbehälters, das einen Zugangspunkt vergeben soll.
Returns:
AccessPoint - Der vergebene Zugangspunkt.

---

retrieveAccessPoint

```
public void retrieveAccessPoint(java.lang.String direction,
                                AccessPoint ap)
```

Freigabe eines zuvor vergebenen Zugangspunkts
Parameters:
`direction` - Name des Zugangsbehälters, das ein Zugangspunkt freigegeben soll.
`ap` – Zugangspunkt, der freigegeben wird.

---

getCoastContainer

```
public CoastContainer getCoastContainer(java.lang.String direction)
```

---

getFirstUnusedDirection

```
public java.lang.String getFirstUnusedDirection()
```

Gibt den ersten nicht verwendeten Zugangsbehälternamen dieses Steuerungsobjekts zurück. Dabei wird zuerst der östliche, danach der südliche und nördliche und zum Schluss der westliche Zugangsbehälter überprüft.
Returns:
String - Der Zugangsbehältername.

---

getAllUnusedDirections

```
public java.util.Enumeration getAllUnusedDirections()
```

Gibt alle nicht verwendeten Zugangsbehälter in Form einer Aufzählung zurück. Dabei wird zuerst der östliche, danach der südliche und nördliche und zum Schluss der westliche Zugangsbehälter in die Aufzählung eingefügt, sofern sie nicht verwendet werden.
Returns:
Enumeration - Aufzählung der Zugangsbehälter.

---

getAllPossibleDirections

```
public java.util.Enumeration getAllPossibleDirections()
```

Gibt alle Zugangsbehälter in Form einer Aufzählung zurück, die noch freie Zugangpunkte haben. Dabei wird zuerst der östliche, danach der südliche und nördliche und zum Schluss der westliche Zugangsbehälter in die Aufzählung eingefügt, sofern sie noch freie Zugangspunkte haben.
Returns:
Enumeration - Aufzählung der Zugangsbehälter.

---

setListener

```
protected void setListener()
```

Wird nach Erzeugung eines Steuerungsobjekts aufgerufen, um Mausevents zu aktivieren.

---

getBaseListener

`protected controlshapes.BaseListener` **`getBaseListener`**`()`

Gibt die Referenz zurück, die die Mausevents implemtiert.

Returns:

BaseListener - Mausevents werden von dieser Referenz implementiert.

---

fillBasicsJMenu

`public void` **`fillBasicsJMenu`**`(javax.swing.JMenu jmenu)`

Implementation des basics - Abschnitts des Kontextmenüs.

Parameters:

`jmenu` - Die Referenz des basics - Menüs.

---

fillAddJMenu

`public void` **`fillAddJMenu`**`(javax.swing.JMenu jmenu)`

Implementation des Einfügemenüs.

Parameters:

`jmenu` - Die Referenz des Einfügemenüs.

---

fillDeleteJItem

`public void` **`fillDeleteJItem`**`(javax.swing.JMenuItem jitem)`

Implementation der Löschoption.

Parameters:

`jitem` - Die Referenz der Löschoption.

---

fillPrivatesJMenu

`public void` **`fillPrivatesJMenu`**`(javax.swing.JMenu jmenu)`

Implementation des privates - Abschnitts des Kontextmenüs.

Parameters:

`jmenu` - Die Referenz des privates - Menüs.

---

updatePopup

`public void` **`updatePopup`**`()`

---

getName

`public java.lang.String` **`getName`**`()`

Gibt den Namen dieses Steuerungsobjekts zurück.

Returns:

String - Der Name des Steuerungsobjekts.

---

setName

`public void` **`setName`**`(java.lang.String name)`

Setzt den Typ des Steuerungsobjekts.

Parameters:

`name` - - Der Typ des Steuerungsobjekts.

---

getId

`public java.lang.String` **`getId`**`()`

Gibt die ID dieses Steuerungsobjekts zurück.

Returns:
String - Die ID des Steuerungsobjekts.

---

setId

`public void` **`setId`**`(java.lang.String id)`
Setzt die ID des Steuerungsobjekts.
Parameters:
`id` - - Die ID des Steuerungsobjekts.

---

setShapeType

`protected void` **`setShapeType`**`(java.lang.String shapeType)`
Setzt den Typ des Steuerungsobjekts.
Parameters:
`shapeType` - - Der Typ des Steuerungsobjekts.

---

getShapeType

`protected java.lang.String` **`getShapeType`**`()`
Gibt den Typ dieses Steuerungsobjekts zurück.
Returns:
String - Der Typ des Steuerungsobjekts.

---

getNumberOfOutgoingTransitions

`public int` **`getNumberOfOutgoingTransitions`**`()`
Gibt die Anzahl der ausgehenden Kanten dieses Steuerungsobjekts zurück.
Returns:
int - Anzahl der ausgehenden Kanten.

---

getNumberOfIncomingTransitions

`public int` **`getNumberOfIncomingTransitions`**`()`
Gibt die Anzahl der eingehenden Kanten dieses Steuerungsobjekts zurück.
Returns:
int - Anzahl der eingehenden Kanten.

---

getIncomingTransitions

`public java.util.Enumeration` **`getIncomingTransitions`**`()`
Gibt alle eingehenden Kanten dieses Steuerungsobjekts, in Form einer Aufzählung, zurück.
Returns:
Enumeration - Aufzählung der eingehenden Kanten.

---

getOutgoingTransitions

`public java.util.Enumeration` **`getOutgoingTransitions`**`()`
Gibt alle ausgehenden Kanten diese Steuerungsobjekts, in Form einer Aufzählung, zurück.
Returns:
Enumeration - Aufzählung der ausgehenden Kanten.

---

getTransitions

`public java.util.Enumeration` **`getTransitions`**`()`
Gibt alle eingehenden und ausgehenden Kanten dieses Steuerungsobjekts, in Form einer Aufzählung, zurück.
Returns:

Enumeration - Aufzählung aller Kanten.

---

addOutgoingTransition

`protected boolean` **`addOutgoingTransition`**`(java.lang.Object obj)`

Fügt eine neue ausgehende Kante zum Steuerungsobjekten hinzu.

Parameters:

`obj` - Die neue ausgehende Kante.

Returns:

boolean - true Erfolgreich eingefügt.

false Nicht erfolgreich eingefügt.

---

addIncomingTransition

`protected boolean` **`addIncomingTransition`**`(java.lang.Object obj)`

Fügt eine neue eingehende Kante zum Steuerungsobjekten hinzu.

Parameters:

`obj` - Die neue eingehende Kante.

Returns:

boolean - true Erfolgreich hinzugefügt.

false Nicht erfolgreich hinzugefügt.

---

deleteOutgoingTransition

`protected boolean` **`deleteOutgoingTransition`**`(java.lang.Object obj)`

Entfernt eine ausgehende Kante vom Steuerungsobjekte.

Parameters:

`obj` - Die zu entfernende ausgehende Kante.

Returns:

boolean - true Erfolgreich gelöscht.

false Nicht erfolgreich gelöscht.

---

deleteIncomingTransition

`protected boolean` **`deleteIncomingTransition`**`(java.lang.Object obj)`

Entfernt eine eingehende Kante vom Steuerungsobjekten.

Parameters:

`obj` - Die zu entfernende eingehende Kante.

Returns:

boolean - true Erfolgreich gelöscht.

false Nicht erfolgreich gelöscht.

---

deleteTransition

`protected boolean` **`deleteTransition`**`(java.lang.Object obj)`

Entfernt eine Kante vom Steuerungsobjekte. Es wird automatisch bestimmt, ob die Kante eine eingehende oder ausgehende Kante war.

Parameters:

`obj` - Die zu entfernende Kante.

Returns:

boolean - true Erfolgreich gelöscht.

false Nicht erfolgreich gelöscht.

---

swapBooleanGuard

`public void swapBooleanGuard()`

Vertauscht die Wächterbedingung von zwei ausgehenden Kanten. Voraussetzung:

Es gibt genau zwei ausgehende Kanten.

---

getBaseJComponent

`public BaseJComponent getBaseJComponent()`

Gibt die grafische Komponente dieses Steuerungsobjekten zurück.
Returns:
BaseJComponent - Die grafische Komponente dieses Steuerungsobjekten.

---

setBaseJComponent

`public void setBaseJComponent(BaseJComponent bjc)`

Setzt die grafische Komponente dieses Steuerungsobjekten fest.
Parameters:
`BaseJComponent` - - Die grafische Komponente dieses Steuerungsobjekten.

---

getSuccessorBaseShapes

`public java.util.Enumeration getSuccessorBaseShapes()`

Alle Folgesymbole dieses Steuerungsobjekts werden, in Form einer Aufzählung, zurückgegeben.

---

setController

`public void setController(DiagramContainer controller)`

Der Kontainer, der dieses Steuerungsobjekt verwaltet, wird festgelegt.
Parameters:
`controller` - Der Steuerungseditor zu diesem Steuerungsobjekt.

---

getController

`public DiagramContainer getController()`

Der Kontainer, der dieses Steuerungsobjekt verwaltet, wird zurückgegeben.
Returns:
DiagramContainer Der Steuerungseditor zu diesem Steuerungsobjekt.

---

getBaseShapeToString

`public java.lang.String getBaseShapeToString()`

Textliche Darstellung dieses Steuerungsobjekts. Wird von abgeleiteten Objekten überschrieben.
Returns:
String Stringrepräsentation dieses Steuerungsobjekts.

---

toString

`public java.lang.String toString()`

Textliche Darstellung dieses Steuerungsobjekts. Ruft getBaseShapeToString() auf.
Overrides:
toString in class java.lang.Object
Returns:
String Stringrepräsentation dieses Steuerungsobjekts.

6.4.1.8 Start

controlshapes

Class Start

```
java.lang.Object
  |
  +--controlshapes.BaseShape
        |
        +--controlshapes.Start
```

public class **Start**
extends BaseShape

Die Implementierung des Startsymbols.

Field Detail

type

`public static java.lang.String` **`type`**

Typname des Startsymbols

automat

`public automat.Automat` **`automat`**

Die Referenz des Automaten

Constructor Detail

Start

`public` **`Start`**`(DiagramContainer controller)`

Zur Erzeugung des Startsymbols wird die Referenz des Steuerungseditors übergeben.

Parameters:

`controller` - Die Referenz des Steuerungseditors.

Method Detail

fillPrivatesJMenu

`public void` **`fillPrivatesJMenu`**`(javax.swing.JMenu jmenu)`

Implementation des privates - Abschnitts des Kontextmenüs für das Startsymbol.

Overrides:

fillPrivatesJMenu in class BaseShape

Parameters:

`jmenu` - Die Referenz des privates - Menüs.

start

`public void` **`start`**`()`

Startet die Transformation.

stop

`public void` **`stop`**`()`

Stopt die Transformation.

init_

`public void init_()`

Initialisiert den Automaten.

6.4.1.9 Stop

controlshapes

Class Stop

```
java.lang.Object
  |
  +--controlshapes.BaseShape
        |
        +--controlshapes.Stop
```

---

public class **Stop**
extends BaseShape

Implementation des Stopsymbols.

Field Detail

type

`public static java.lang.String type`

Typname des Stopsymbols.

Constructor Detail

Stop

`public Stop(DiagramContainer controller)`

Zur Erzeugung des Stopsymbols wird die Referenz des Steuerungseditors übergeben.
Parameters:
`controller` - Die Referenz des Steuerungseditors.

6.4.1.10 Activity

controlshapes

Class Activity

```
java.lang.Object
  |
  +--controlshapes.BaseShape
        |
        +--controlshapes.Activity
```

---

public class **Activity**
extends BaseShape

Die Implementierung des Aktionszustandes.

Field Detail

type

`public static java.lang.String type`

Typname des Startsymbols

IMPLICIT

`public static java.lang.String` **`IMPLICIT`**

Stringrepräsentation des impliziten Aktionszustands

BOOLEAN

`public static java.lang.String` **`BOOLEAN`**

Stringrepräsentation des booleschen Aktionszustands

---

guardNameTrue

`public static java.lang.String` **`guardNameTrue`**

Stringrepräsentation des true - Wächters

---

guardNameFalse

`public static java.lang.String` **`guardNameFalse`**

Stringrepräsentation des false - Wächters

---

rsd

`protected RuleSelectionDialog` **`rsd`**

Referenzklasse, die den Dialog, zum Wählen einer Aktion, realisiert.

Constructor Detail

Activity

`public` **`Activity`**`(DiagramContainer controller)`

Zur Erzeugung des Aktionszustands wird die Referenz des Steuerungseditors übergeben.
Parameters:
`controller` - Die Referenz des Steuerungseditors.

Method Detail

selectActivity

`public void` **`selectActivity`**`()`

Wählt eine Aktion aus, in dem ein Dialog geöffnet wird und der Benutzer per Maus einen Regelnamen wählen muss.

---

updateDialog

`public void` **`updateDialog`**`()`

Aktualisiert die grafische Darstellung des Aktionszustandes nach Erhalt einer Aktion über das Dialogfenster.
Overrides:
updateDialog in class BaseShape

---

fillPrivatesJMenu

`public void` **`fillPrivatesJMenu`**`(javax.swing.JMenu jmenu)`

Implementation des privates - Abschnitts des Kontextmenüs.
Overrides:
fillPrivatesJMenu in class BaseShape
Parameters:
`jmenu` - Die Referenz des privates - Menüs.

---

fillDeleteJItem

`public void` **`fillDeleteJItem`**`(javax.swing.JMenuItem jitem)`

Implementation der Löschoption.
Overrides:
fillDeleteJItem in class BaseShape
Parameters:
`jitem` - Die Referenz der Löschoption.

---

setExpression

`public void` **`setExpression`**`(java.lang.String expression)`

Setzt den Regelnamen als Aktion fest.
Parameters:
`expression` - Die Aktion des Aktionszustands.

---

getExpression

`public java.lang.String` **`getExpression`**`()`

Gibt die Aktion des Aktionszustands zurück.
Returns:
String - Die Aktion.

---

setImplicitModus

`public void` **`setImplicitModus`**`(boolean saveTransition)`

Überführt einen booleschen Aktionszustand in einen impliziten Aktionszustand.
Parameters:
`saveTransition` - Wenn dieser Parameter true ist, wird die Kante mit dem true - Wächter gerettet, ansonsten mit false - Wächter.

---

setBooleanModus

`public void` **`setBooleanModus`**`(boolean changeImplicitTransitionTo)`

Überführt einen impliziten Aktionszustand in einen booleschen Aktionszustand.
Parameters:
`changeImplicitTransition` - Wenn dieser Parameter true ist, erhält die neu erzeugte Kante einen true - Wächter, ansonsten einen false - Wächter.

---

getModus

`public java.lang.String` **`getModus`**`()`

Gibt den Modus dieses Akionszustands zurück.
Returns:
String - IMPLIZIT oder BOOLEAN

---

toString

`public java.lang.String` **`toString`**`()`

Implementierung des toString()'s.
Overrides:
toString in class BaseShape
Tags copied from class: BaseShape
Returns:
String Stringrepräsentation dieses Steuerungsobjekts.

6.4.1.11 Decision

controlshapes

Class Decision

```
java.lang.Object
  |
  +--controlshapes.BaseShape
        |
        +--controlshapes.Decision
```

---

public class **Decision**
extends BaseShape

Die Implementierung des Verzweigungssymbols.

Field Detail

type

`public static java.lang.String` **`type`**

Typname des Verzweigunssymbols.

---

rsd

`protected RuleSelectionDialog` **`rsd`**

Referenzklasse, die den Dialog, zum Wählen einer Wächterbedingung, realisiert.

Constructor Detail

Decision

`public` **`Decision`**`(DiagramContainer controller)`

Zur Erzeugung des Verzweigungssymbols wird die Referenz des Steuerungseditors übergeben.
Parameters:
`controller` - Die Referenz des Steuerungseditors.

Method Detail

updateDialog

`public void` **`updateDialog`**`()`

Aktualisiert die grafische Darstellung des Verzweigungssymbols nach Erhalt eines guard's über das Dialogfenster.
Overrides:
updateDialog in class BaseShape

---

selectGuard

`public void` **`selectGuard`**`()`

Wählt einen guard aus, indem ein Dialog geöffnet wird und der Benutzer per Maus einen Ausdruck, der aus mehreren Regelnamen bestehen kann, auswählt. Diese Regelnamen werden mit dem ODER Operator verknüpft.

---

fillPrivatesJMenu

`public void` **`fillPrivatesJMenu`**`(javax.swing.JMenu jmenu)`

Implementation des privates - Abschnitts des Kontextmenüs.
Overrides:
fillPrivatesJMenu in class BaseShape

Parameters:
`jmenu` - Die Referenz des privates - Menüs.

---

fillDeleteJItem

`public void **fillDeleteJItem**(javax.swing.JMenuItem jitem)`

Implementation der Löschoption.
Overrides:
fillDeleteJItem in class BaseShape
Parameters:
`jitem` - Die Referenz der Löschoption.

6.4.1.12 While

controlshapes
Class While

```
java.lang.Object
  |
  +--controlshapes.BaseShape
        |
        +--controlshapes.While
```

---

public class **While**
extends BaseShape

Die Implementierung des Schleifensymbols.

Field Detail

type

`public static java.lang.String **type**`

Typname des Schleifensymbols

---

elseName

`public java.lang.String **elseName**`

---

rsd

`protected RuleSelectionDialog **rsd**`

Referenzklasse, die den Dialog, zum Wählen einer Wächterbedingung, realisiert.

Constructor Detail

While

`public **While**(DiagramContainer controller)`

Zur Erzeugung des Schleifensymbols wird die Referenz des Steuerungseditors übergeben.
Parameters:
`controller` - Die Referenz des Steuerungseditors.

While

```
public While(DiagramContainer controller,
             java.lang.String id)
```

Method Detail

updateDialog

`public void **updateDialog**()`

Aktualisiert die grafische Darstellung des Schleifensymbols nach Erhalt eines guard's über das Dialogfenster.
Overrides:
updateDialog in class BaseShape

---

selectGuard

`public void **selectGuard**()`

Wählt einen guard aus, indem ein Dialog geöffnet wird und der Benutzer per Maus einen Ausdruck, der aus mehreren Regelnamen bestehen kann, auswählt. Diese Regelnamen werden mit dem ODER – Operator verknüpft.

---

fillPrivatesJMenu

`public void **fillPrivatesJMenu**(javax.swing.JMenu jmenu)`

Implementation des privates - Abschnitts des Kontextmenüs.
Overrides:
fillPrivatesJMenu in class BaseShape
Parameters:
`jmenu` - Die Referenz des privates - Menüs.

---

fillDeleteJItem

`public void **fillDeleteJItem**(javax.swing.JMenuItem jitem)`

Implementation der Löschoption.
Overrides:
fillDeleteJItem in class BaseShape
Parameters:
`jitem` - Die Referenz der Löschoption.

6.4.1.13 Transition

controlshapes
Class Transition

```
java.lang.Object
  |
  +--controlshapes.BaseShape
        |
        +--controlshapes.Transition
```

---

public class **Transition**
extends BaseShape

Die Implementierung des Zustandsübergangs.

Field Detail

type

`public static java.lang.String **type**`

Typname des Zustandsübergangs

---

north

```
public static java.lang.String north
```
Stringrepräsentation für Norden

---

south

```
public static java.lang.String south
```
Stringrepräsentation für Süden

---

west

```
public static java.lang.String west
```
Stringrepräsentation für Westen

---

east

```
public static java.lang.String east
```
Stringrepräsentation für Osten

Constructor Detail

Transition

```
public Transition(DiagramContainer controller,
                  java.lang.Object sourceObject,
                  java.lang.Object targetObject)
```
Zur Erzeugung des Aktionszustands wird die Referenz des Steuerungseditors übergeben, sowie das Ziel- und Quellobjekt.

Parameters:

`controller` - Die Referenz des Steuerungseditors.

`sourceObject` - Das Quellobjekt.

`targetObject` - Das Zielobjekt.

Transition

```
public Transition(DiagramContainer controller,
                  java.lang.Object sourceObject,
                  java.lang.Object targetObject,
                  java.lang.String guard)
```
Zur Erzeugung des Aktionszustands wird die Referenz des Steuerungseditors übergeben, sowie das Ziel- und Quellobjekt.

Parameters:

`controller` - Die Referenz des Steuerungseditors.

`sourceObject` - Das Quellobjekt.

`targetObject` - Das Zielobjekt.

`guard` - Die Wächterbedingung des Zustandsüberganges.

Method Detail

deleteObject

```
public void deleteObject()
```
Löscht seinen Guard, der Komponente darstellt, aus dem DiagramContainer.

Overrides:

deleteObject in class BaseShape

---

setTargetObject

```
public void setTargetObject(java.lang.Object obj)
```

Setzt das Zielobjekt dieser Kante.
Parameters:
`obj` - Das Zielobjekt dieser Kante.

---

setSourceObject

`public void **setSourceObject**(java.lang.Object obj)`

Setzt das Quellobjekt dieser Kante.
Parameters:
`obj` - Das Zielobjekt dieser Kante.

---

getSourceObject

`public java.lang.Object **getSourceObject**()`

Gibt das Quellobjekt dieser Kante zurück.
Returns:
Object Das Quellobjekt dieser Kante.

---

getTargetObject

`public java.lang.Object **getTargetObject**()`

Gibt das Zielobjekt dieser Kante zurück
Returns:
Object Das Zielobjekt dieser Kante.

---

isDeadEnd

`public boolean **isDeadEnd**()`

Stellt fest, ob das Zielobjekt dieser Kante ein Stopsymbol ist oder nicht.
Returns:
boolean - true --> Das Zielobjekt ist ein Stopsymbol.

false --> Das Zielobjekt ist kein Stopsymbol.

---

getDirection

`public java.lang.String **getDirection**(java.lang.String direction)`

Gibt die gegenüberliegende Küste zurück.
Parameters:
`direction` - Die Küste, von der die gegenüberliegende Küste bestimmt werden soll.
Returns:
String - Die gegenüberliegende Küste.

---

getSrcDirection

`public java.lang.String **getSrcDirection**()`

Gibt die Küste zurück, aus der diese Kante heraustretet.
Returns:
String Die Quellküste dieser Kante.

---

getTrgDirection

`public java.lang.String **getTrgDirection**()`

Gibt die Küste zurück, in der diese Kante eintretet.
Returns:
String Die Zielküste dieser Kante.

---

setSrcDirection

`public void` **`setSrcDirection`**`(java.lang.String direction)`

Teilt dieser Kante eine neue Küste zu, aus der sie austreten soll.

Parameters:

`direction` - Die neue Quellküste.

---

setTrgDirection

`public void` **`setTrgDirection`**`(java.lang.String direction)`

Teilt dieser Kante eine neue Küste zu, in die sie eintreten soll.

Parameters:

`direction` - Die neue Zielküste.

---

getSrcAccessPoint

`public AccessPoint` **`getSrcAccessPoint`**`()`

Gibt den Quellzugangspunkt dieser Kante zurück.

Returns:

AccessPoint - Der Quellzugangspunkt.

---

getTrgAccessPoint

`public AccessPoint` **`getTrgAccessPoint`**`()`

Gibt den Zielzugangspunkt dieser Kante zurück.

Returns:

AccessPoint - Der Zielzugangspunkt.

---

activate

`public void` **`activate`**`()`

Markiert diese Kante als aktiviert.

---

deActivate

`public void` **`deActivate`**`()`

Markiert diese Kante als nicht aktiviert.

---

addGuard

`public void` **`addGuard`**`(java.lang.String guard)`

Fügt einen Teilausdruck der Wächterbedingung hinzu.

Parameters:

`guard` - Teilbedingung des Wächters dieser Kante.

---

setGuard

`public void` **`setGuard`**`(java.lang.String guard)`

Setzt die Wächterbedingung dieser Kante.

Parameters:

`guard` - Die Komplette Wächterbedingung dieser Kante.

---

getGuard

`public java.lang.String` **`getGuard`**`(java.lang.String delim)`

Gibt die Wächterbedingung dieser Kante zurück. Sämtliche Teilbedingungen werden durch Trennzeichen getrennt.

Parameters:
`delim` - Trennzeichen.
Returns:
String Der Guard.

---

getGuard

`public java.lang.String **getGuard**()`

Gibt die Wächterbedingung dieser Kante zurück.
Returns:
String Der Guard.

---

deleteGuard

`public void **deleteGuard**()`

Löscht den Guard dieser Kante.

---

toString

`public java.lang.String **toString**()`

Textuelle Darstellung dieses Steuerungsobjekts.
Overrides:
toString in class BaseShape
Returns:
String Stringrepräsentation dieser Kante.

6.4.2 Automat

Die folgende Klasse befindet sich in der Bibliothek *autmat*.

6.4.2.1 Automat

automat
Class automat

```
java.lang.Object
  |
  +--automat.Automat
```

---

public class **Automat**
extends java.lang.Object
implements java.lang.Runnable

Der Automat des Steuerungseditors interpretiert den Steuerungsgraphen. Dazu wird das Startsymbol an diese Klasse übergeben. Das Startsymbol repräsentiert den Anfangspunkt der Transformation.

Constructor Detail

Automat

`public **Automat**(controlshapes.Start startShape)`

Übergabe von controlshapes.Start, um den Beginn des Steuerungsgraphen zu spezifizieren.
Parameters:
`startShape` - Startsymbol des Steuerungsgraphen.

Method Detail

init

`public void **init**()`

Initialisiert den Automaten. Beim nächsten Starten des Automaten beginnt die Transformation wieder beim Startsymbol.

---

start

`public void **start**()`

Startet den Automaten, so daß die Interpretation des Steuerungsgraphen anfängt. Somit werden Graphersetzungen veranlasst.

---

stop

`public void **stop**()`

Stopt den Automaten, mit der Folge, dass die Transformation beendet wird.

---

run

`public void **run**()`

Die Interpretation des Steuerungsgraphen wird hier implenmentiert.
Specified by:
run in interface java.lang.Runnable

7 LITERATURVERZEICHNIS

[Albayrak98] Dr. Can Albayrak:
Die While Hierarchie für Programmschemata
Shaker Verlag /Aachen August'1998.

[Booc94] Booch, Grady:
Object-oriented Analysis and Design with Applications,
1994, Benjamin/Cummings Publishing Company.

[GRA92] M, Korff, M. Löwe, G. Taentzer u. A. Wagner:
Algebraische Graphgrammatiken
Vorlesungsskript / Oktober 1992.

[Gruner] S. Gruner:
Metatyping is compatible to the Typed SPO Approach.
To appear in: H. Ehrig, G.Taentzer (Eds.) GRATRA 2000 workshop, Berlin 2000.

[Jaco92] Jacobson, Ivar:
Object-oriented Software Engineering, A Use Case driven Approach,
1992, Addison-Wesley Verlag.

[King/Rodgers] P. J. H. King / P. J. Rodgers:
A Graph Rewriting Visual Language for Database Programming
Birkbeck College – University of / University of London Sheffield.

[Mel97] Boris Melemed:
Grundkonzeption und -implementierung einer Attributkomponente für ein Graphtransformationssystem
TU –Berlin / Diplomarbeit 1997

[Michael97] Michael Rudolph:
Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation
TU –Berlin / Diplomarbeit 1997

[MicSys2/99] Microsoft System Journal:
Der Einsatz von Aktivitätsdiagrammen in der UML
2/99.

[Münch] Manfred Münch:
Constraints, Pre- und Postconditions in PROGRES
Rhein.-Westf. Techn. Hochschule / Diplomarbeit 1996

[ObFo97] Objekt Fokus:
Objekt Fokus; Mit objektorientierten Technologie in die Zukunft
September/Oktober 1997, IT-Verlag.

[Rati97] Rational Software Corporation:
Unified Modeling Language, Notation Guide
Januar 1997, Version 1.0, http://www.rational.com.

[Rati99] Rational Software Corporation:
Unified Modeling Language, Notation Guide
1999, Version 1.3, http://www.rational.com

[Rodgers] P. J. Rodgers:
A Graph Rewriting Programming Language for Graph Drawing
University of Kent.

[Rodgers21] P. J. Rodgers:
Constructs for Programming with Graph Rewrites
To appear in: H. Ehrig, G.Taentzer (Eds.) GRATRA 2000 workshop, Berlin 2000 (Paper Nr.21).

[Rumb93] J. Rumbough, M. Blaka, W. Prenerleni, F. Eddy, W. Lorensen:
Objektorientiertes Modellieren und Entwerfen,
1993, Prentice-Hall International, London.

[Schürr] A. Schürr:
PROGRES for Beginners
RWTH – Aachen.

[Schürr97] A. Schürr:
Programmed Graph Replacement Systems.
Kapitel 7 in: G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1.
World Scientefic, Singapore 1997.

[Schürr98] A. Schürr:
The PROGRES Manual (Version 9.x)
RWTH – Aachen

[Schw95] Schwarze, Jochen:
Systementwicklung: Grundzüge der wirtschaftlichen Planung, Entwicklung und Einführung von Informationssystemen,
1995, Neue Wirtschafts-Briefe Verlag, Herne-Berlin.

[Taentzer] Gabriele Taentzer,
How to Integrate Graph Transformation with an Object-Oriented Language
TU – Berlin.

[UML99] Grady Booch, Jim Rumbaugh, Ivar Jacobson
Das UML – Benutzerhandbuch
1999, Addison-Wesley Verlag.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbständig angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Berlin, den ……………………..… ………………………..

(Abgabedatum) (Unterschrift)