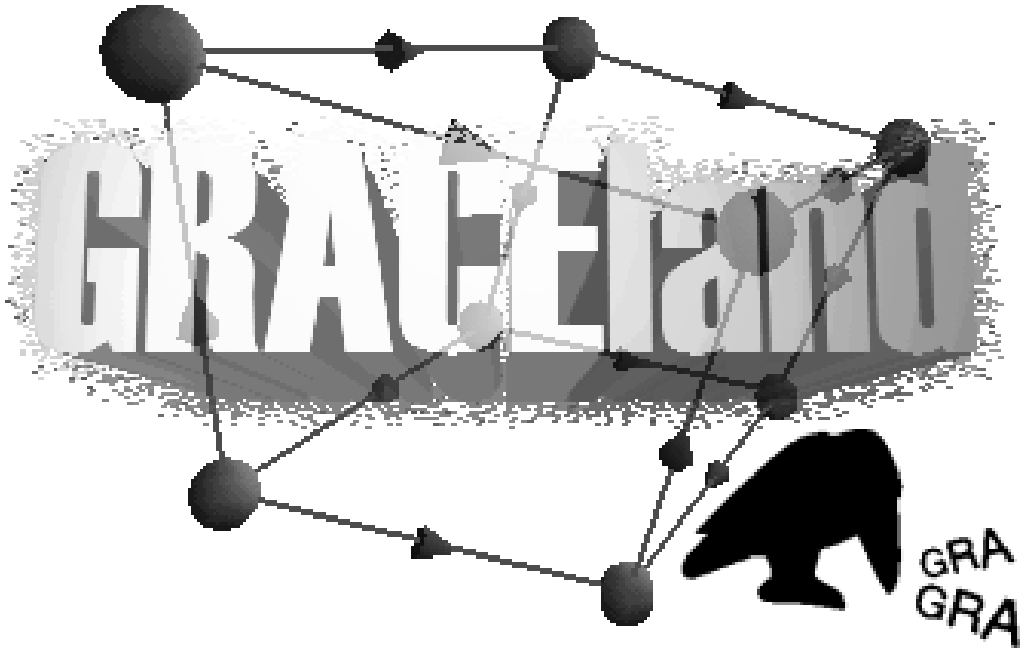
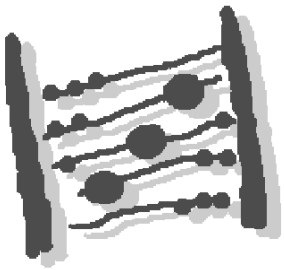

GRACELand
GRA
GRA



Universität Bremen

Fachbereich 3

Mathematik und Informatik

Diplomarbeit

GRACEland

Ein 3D-Editor und Interpreter für die graph- und regelbasierte Sprache GRACE

Martin Faust

Betreuer: Prof. Dr. Hans-Jörg Kreowski

Gutachter: Prof. Dr. Friedrich Wilhelm Bruns

Bremen, den 20. Juli 1998

Inhaltsverzeichnis

1 EINLEITUNG

Graphen werden seit langem in verschiedenen Zusammenhängen angewandt. Sie bilden die Grundlage von vielen Systemen, wie z.B. Spezifikationsmethoden (UML [Bur97], Booch [Boo94], ...). Zusammen mit Regeln bilden sie ein neues Programmiersprachen-Paradigma.

Man unterscheidet zwischen klassischen, einfachen Graphersetzungssystemen, den Graph-Grammatiken, und programmierten Graphersetzungssystemen. Graph-Grammatiken sind eine Verallgemeinerung von Chomsky-Grammatiken. Relativ neu hingegen sind programmierte Graphersetzungssysteme, die eine visuelle Programmiersprache darstellen. Sie zeichnen sich durch Mechanismen zur Strukturierung der Programme und Steuerung des Kontrollflusses aus.

Das Ziel der Arbeit ist es, die Sprache GRACE zu implementieren. GRACE ist ein Akronym für graph- und regelzentrierte Spezifikations- und Programmiersprache (GRAph and rule CEntered). Die Sprache wird seit ca. fünf Jahren von Forscherinnen und Forschern aus Europa entwickelt. Bisher existiert sie nur in der Theorie, so daß diese Arbeit die erste Implementierung dieser Sprache darstellt. In den Entwurf der Sprache GRACE flossen Erfahrungen aus den bestehenden Graphersetzungssystemen PROGRES ([Sch94]) und AGG ([Rud97]) ein. Gleichzeitig wurden neue Konzepte, u.a. Module, eingeführt, um den Anforderungen an einer Spezifikations- und Programmiersprache gerecht zu werden.

Mit GRACEland wird eine visuelle Entwicklungsumgebung für GRACE vorgestellt. Erstmals wird im Zusammenhang mit Graphersetzungssystemen eine Virtual-Reality-Schnittstelle eingesetzt, um der inhärenten dreidimensionalen Struktur von vielen Graphen gerecht zu werden.

An ausgewählten Anwendungen wird dann die Modellierung mit Graphen und Graphregeln, sowie der Einsatz von GRACEland betrachtet. Ziel ist es, den Leserinnen und Lesern einen Überblick über die Anwendungsmöglichkeiten zu geben.

In der Einleitung des Kapitel 2 werden Eigenschaften und Vorteile von Graphersetzungssystemen beschrieben. Daran schließt sich eine Einführung in die theoretischen Grundlagen von Graphen, Graphregeln und Graphersetzung an. Die Sprache GRACE wird im Kapitel 3 vorgestellt.
Kapitel 4 zeigt die verschiedenen Facetten der Implementierung. Im Anschluß wird die visuelle Entwicklungsumgebung GRACEland vorgestellt. Kapitel 6 zeigt an einigen ausgewählten Anwendungen die Einsatzmöglichkeiten von Graphersetzungssystemen. Den Abschluß bildet Kapitel 7 mit einer kritischen Reflexion der Arbeit.

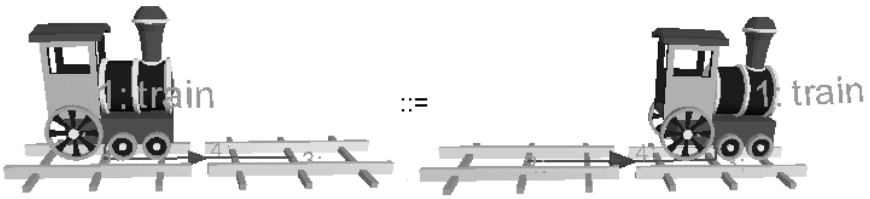
2 GRAPHERSETZUNGSSYSTEME

2.1 Einleitung

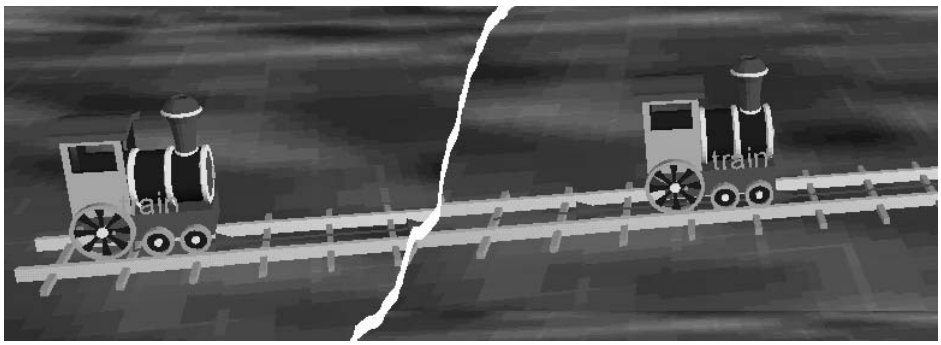
Graphersetzungssysteme bilden ein neues Paradigma für Programmiersprachen, das auf Graphen und Regeln basiert. Im Gegensatz zum imperativen Paradigma, welches relativ maschinennah ist, ist dieses auf einer sehr hohen Abstraktionsstufe (High-Level-Paradigma) angesiedelt, da keine Konzepte eines bestehenden Rechners benutzt werden.

Grundlage von Graphersetzungssystemen sind Graphen, ein sehr häufig verwendetes Konzept, da Graphen intuitiv, anschaulich und leicht zu handhaben sind. Die Mächtigkeit von Graphen basiert auf der graphischen Visualisierung, die dem Graphen innewohnt. Die Visualisierung macht es leicht, Zusammenhänge zwischen einzelnen Komponenten zu erkennen, da sie durch Linien zwischen den Komponenten angezeigt werden. Durch die Verwendung von verschiedenen Repräsentationen von Knoten und Kanten kann die Anschaulichkeit von Graphen weiter gesteigert werden. Man findet Graphen als Basis u.a. in folgenden Konzepten: Petri-Netzen ([Bau90]), Diagrammen (UML [Bur97) und Datenbankmodellen (ER [Gog96], EER [Gog97]).

Regeln sind der andere Teil von Graphersetzungssystemen. Mit Hilfe von Regeln lassen sich lokale Änderungen von Graphen einfach beschreiben. Abbildung 2-1 zeigt ein Beispiel einer Regel und ihrer Anwendung auf einen Graphen.



a)



b)

Abbildung 2-1: a) Graphregel b) Linke Seite: Graph vor Ausführung der Regel b) Rechte Seite: Graph nach Ausführung der Regel

Abbildung 2-1 ist gleichzeitig ein Beispiel für visuelle Programmierung. Die Stärke der visuellen Programmierung ist es, weitestgehend ohne Text auszukommen. Text wird immer dann benötigt, wenn z.B. Parameter berechnet werden müssen. Auch diese Operationen ließen sich graphisch ausdrücken, was aber zu einer Überladung und damit zu einem Nachteil werden könnte. Es ist wichtig, einen Mittelweg zu finden, um beide Konzepte sinnvoll miteinander zu verbinden. Graphersetzungssysteme sind als Grundlage für visuelle Programmierung geradezu prädestiniert, da eine breite theoretische Grundlage und damit auch eine exakte Semantik existiert.

Welchen Vorteil bieten Graphersetzungssysteme gegenüber anderen Systemen?

Zum einen bieten sie eine Unterstützung für Graphen, die Basis von vielen Konzepten sind, und zum anderen erweitern sie Graphen um dynamisches Verhalten durch Graphregeln. Dadurch bilden Graphersetzungssysteme ein einheitliches Framework, welches den Benutzer auf verschiedenen Ebenen unterstützt. Abgerundet wird das System durch einen Interpreter, der in der Lage ist, ein "Graphprogramm" auszuführen. Im Kapitel 6 wird ausführlicher auf die Einsatzgebiete und Vorteile von Graphersetzungssystemen eingegangen. An dieser Stelle sei erwähnt, daß das Konzept der Petri-Netze um eine dynamische Komponente erweitert werden konnte, die bisher nicht erfaßt wurde. Dieses Ergebnis ist bedeutend, da es sich um ein praxisrelevantes Ergebnis handelt.

Grundsätzlich muß man zwischen zwei Arten von Graphersetzungssystemen unterscheiden: einfache und programmierte. Bei einfachen Systemen wird die Regel zufällig aus einer Regelmenge ausgewählt, während bei programmierten Systemen ein Kontrollfluß existiert, der die Regelauswahl einschränkt. Bevor wir uns den programmierten Graphersetzungssystemen und damit GRACE zuwenden, werden in den folgenden Kapiteln zunächst die wichtigsten theoretischen Grundlagen von Graphersetzungssystemen beschrieben.

2.2 Graphen

> *„Die großartige Schönheit eines Graphen besteht darin, daß er unklare, schwer durchschaubare Rätsel auf ein wunderbares Ding aus Linien und Knotenpunkten reduzieren kann." ([Oli95], Seite 273)*

Graph ist aber nicht gleich Graph, obwohl jeder Graph aus einer Menge von Knoten und Kanten zwischen Knoten besteht. Der Unterschied liegt in der variierenden Definition des Kantenbegriffs. Verbindet eine Kante zwei Knoten miteinander, wobei zwischen Quell- und Zielknoten unterschieden wird, so spricht man von *gerichteten Graphen*. Bei *ungerichteten Graphen* wird nicht zwischen Quelle und Ziel differenziert. Verbindet eine Kante n Knoten miteinander, so spricht man von *Hypergraphen*. Knoten und/oder Kanten werden oftmals noch durch Markierungen mit zusätzlicher Information angereichert.

Durch die verschiedenen Ausprägungen läßt sich ein „maßgeschneidertes" Modell für die jeweilige Situation auswählen, da jedes Modell leicht unterschiedliche Eigenschaften besitzt. Allerdings ist es schwer bzw. zum Teil unmöglich, Erkenntnisse von einer Graphklasse auf eine andere zu übertragen. Dieses Problem wird im Zusammenhang mit Konvertierungen zwischen Graphklassen im Kapitel 3.4 wieder zur Sprache gebracht.

Wenn im weiteren von Graphen die Rede ist, so bezieht sich das auf eine abstrakte Basisklasse Graph, von der sich konkrete Graphen, wie z.B. gerichtete Graphen, ableiten. Ansonsten wird der Graphtyp[1] explizit angegeben.

Für die weitere Arbeit wird die Klasse der gerichteten, markierten Graphen herausgegriffen und näher untersucht. An ihr werden die verschiedenen Konzepte erläutert.

Definition 2-1 :Gerichteter, markierter Graph

Sei Σ ein Markierungsalphabet.

Ein gerichteter, markierter Graph ist ein System $G=(V, E, source, target, label)$ bestehend aus:

- Einer endlichen Menge von Knoten V (engl. vertices)
- Einer endlichen Menge von Kanten E (engl. edges)
- Zwei Funktionen $source:E\rightarrow V$ und $target:E\rightarrow V$, die jeder Kante $e\in E$ ihre Quelle source(e) bzw. ihr Ziel target(e) zuordnen.
- Einer Funktion label: $V\cup E \rightarrow \Sigma$, die jedem Knoten und jeder Kante ihre Markierung (engl. label) zuordnet[2].

Diese Definition erlaubt parallele Kanten zwischen zwei Knoten. Graphen, die parallele Kanten zulassen, werden auch als *Multigraphen* bezeichnet.

---

[1] Graphtyp und Graphklasse werden synonym verwendet.

[2] Mit $\cup$ ist die disjunkte Vereinigung gemeint.

Um die Zugehörigkeit einer Komponente zu einem bestimmten Graphen zu betonen, wird der Name des Graphen als Index an die Komponenten gehängt: V_G, E_G, $source_G$, ... Dadurch können mehrere Graphen auseinandergehalten werden.

Graphen können auf verschiedene Arten und Weisen dargestellt werden. Die am häufigsten benutze Visualisierung ist die, in der Knoten als Kreise oder Rechtecke und Kanten durch Linien mit Pfeilen dargestellt werden (vgl. Abbildung 2-2). Die Art der Darstellung und die Anordnung der Symbole hat keinen Einfluß auf die zugrundeliegende Theorie von Graphen. Sie dient vielmehr dazu, dem Benutzer eine intuitive, übersichtliche bzw. ästhetische Darstellung des Graphen zu geben. Mehr zum Thema Visualisierung von Knoten und Kanten findet man in Kapitel 5.2.1. [Jun94] zeigt alternative Repräsentationen von Graphen, wie z.B. Adjazenzmatrizen.

Die folgende Abbildung zeigt die in dieser Arbeit verwendeten Darstellungen von Graphen.

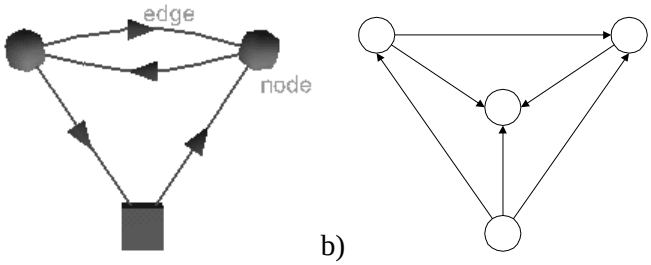


Abbildung 2-2: Darstellungen von Graphen

Bei der Beschriftung "edge" und "node" in Abbildung 2-2 a) handelt es sich um Markierungen der oberen Kante bzw. des rechten Knotens. Alle anderen Graphelemente weisen keine Markierung auf.

2.3 Graphersetzung

Die Basisidee von Graphersetzung ist die Ersetzung eines Graphen durch einen anderen. Die Ersetzungsvorschrift wird in einer Regel[3] $r:L \rightarrow R$ festgehalten, in der L und R Graphen sind. L wird als linke Seite (engl. left-hand side) und R als rechte Seite (engl. right-hand side) der Regel bezeichnet. Eine Anwendung der Regel auf einen Graphen G bedeutet, daß ein Vorkommen von L durch R ersetzt wird.

[3] In manchen Aufsätzen auch als Produktion bezeichnet. Hier wird der Begriff Regel vorgezogen.

Bevor dieser Prozeß näher spezifiziert wird, ein Beispiel zur Veranschaulichung:

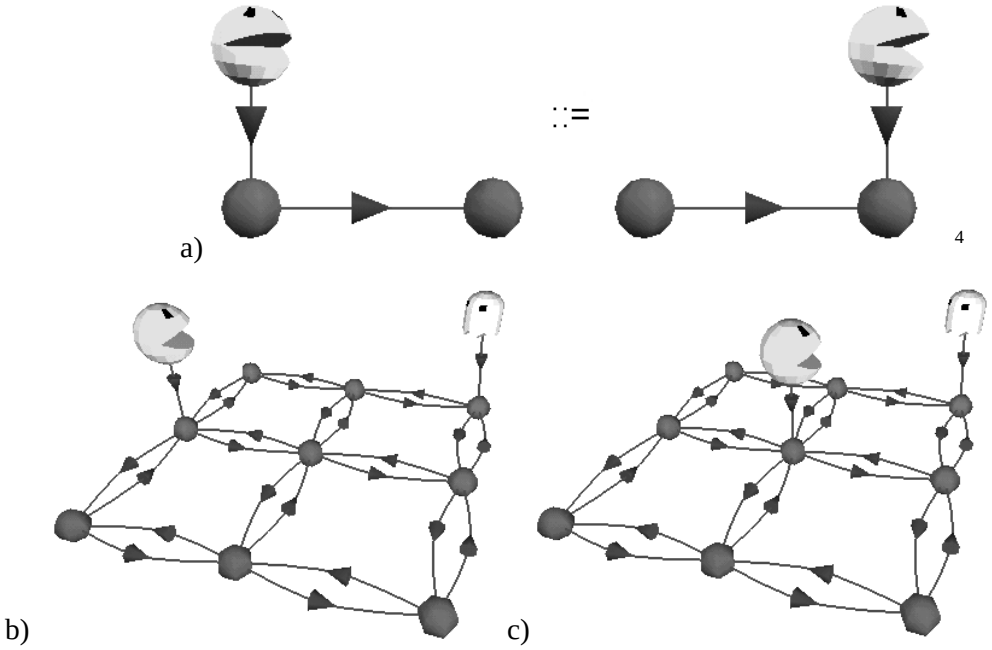


Abbildung 2-3: a) Graphregel b) Graph vor der Ausführung der Regel
c) Graph nach der Ausführung der Regel

Dieses Beispiel[5] zeigt, wie sich der Pacman auf dem Spielfeld bewegt. Als blaue Kugeln sind dabei die Plätze auf dem Spielfeld gekennzeichnet. Diese Plätze sind durch Kanten verbunden, welche die vorhandenen Wege darstellen. Eine Figur befindet sich auf dem Feld, zu dem sie eine Verbindung (Figur→Platz) hat. Der Geist hat in diesem Beispiel nur eine Statistenrolle.

Die Regel besagt, daß der Pacman sich von einem Platz zu einem, durch eine Kante verbundenen, anderen bewegen kann. In Abbildung 2-3 bewegt er sich in die Mitte des Spielfeldes. Dabei wurde die Regel intuitiv korrekt angewendet, indem das Ziel der ausgehenden Kante des Pacmans zum mittleren Knoten umgeleitet wurde. Aus ästhetischen Gründen wurde die Figur mit in die Mitte bewegt.

Um beliebige Regeln anwenden zu können, müssen folgende Fragen geklärt werden:

1. An welcher Stelle kann eine Regel angewendet werden ?
2. Wie sieht der Ersetzungsschritt der linken durch die rechte Seite aus ?
3. Wann darf eine Regel angewendet werden ?

---

[4] Im Beispiel wurden die Markierungen durch die Darstellung der Knoten kodiert. Dabei haben die blauen Knotenpunkte und alle Kanten keine Markierungen. Der Pacman und der Geist besitzen eine implizite Markierung ("Pacman" bzw. "Ghost").

[5] Die Inspiration für dieses Beispiel war ein Vortrag auf der European School on Graph Transformation (siehe dazu [Roz97], Kapitel 3, Seite 163 ff).

2.3.1 Ansatzsuche

Hinter dem Begriff der Ansatzsuche, auch als Matching bezeichnet, befindet sich die Suche nach einem Bild der linken Seite der Regel im Graphen. Was bedeutet Bild eines Graphen ? Informell gesprochen, muß nach einem Teilgraphen gesucht werden, der genauso aufgebaut ist, wie die linke Seite. D.h. es muß eine Abbildung gefunden werden, die jeder Kante und jedem Knoten der linken Seite ein Bild im Graphen zugeordnet, so daß Konnektivität und Markierungen erhalten bleiben. Eine solche Abbildung wird als Graphenmorphismus oder kurz als Morphismus bezeichnet.

Definition 2-2: Graphenmorphismus

Seien A und B zwei gerichtete, markierte Graphen. Dann besteht ein Graphenmorphismus $m:A \rightarrow B$ aus zwei Abbildungen $m_V:V_A \rightarrow V_B$ und $m_E:E_A \rightarrow E_B$, so daß für alle $v \in V_A$ und $e \in E_A$ gilt:

1. $m_V(source_A(e)) = source_B(m_E(e))$,
2. $m_V(target_A(e)) = target_B(m_E(e))$,
3. $label_A(v) = label_B(m_V(v))$ und
4. $label_A(e) = label_B(m_E(e))$.

Der Morphismus m(A) wird auch als das Bild von A in B unter m bezeichnet. Von einem *Graphenisomorphismus* m spricht man, wenn die Abbildungen m_V und m_E bijektiv sind.

Die Bedingung 1. und 2. erhalten die Konnektivität des Graphen. Es werden keine zusammenhängenden Teile auseinandergerissen. Bedingung 3. und 4. lassen nur dann eine Zuordnung zu, wenn die Markierungen von Bild und Original gleich sind.

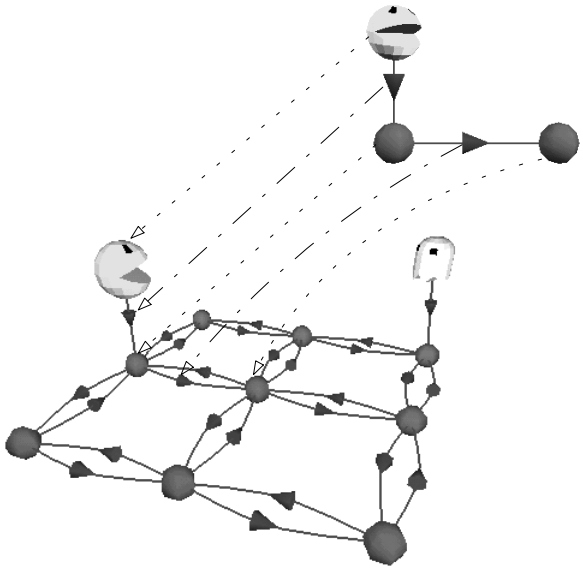


Abbildung 2-4: Graphenmorphismus

Abbildung 2-4 zeigt den Graphenmorphismus, der in der Regelanwendung von Abbildung 2-3 benutzt wurde. Die Linienart gibt an, um welche Abbildung (m_V oder m_E) es sich handelt.

2.3.2 Regelanwendung

Bisher wurde Regelanwendung als Ersetzen einer linken durch eine rechte Seite aufgefaßt. In einer konkreten Situation, wie z.B. in Abbildung 2-3, müßte demnach die linke Seite aus dem Graph entfernt und die rechte Seite hinzufügt werden. Dieses würde aber zu hängenden Kanten (engl. dangling edges) führen, deren Ziel- und/oder Quellknoten gelöscht wurde(n). Eine Möglichkeit, mit diesen Kanten umzugehen, ist es, sie aus dem Graphen zu entfernen. Aber auch dann ist noch nicht geklärt, wie die rechte Seite zum Graphen hinzugefügt werden muß.

Betrachtet man die Graphregel aus Abbildung 2-3 a) genauer, so stellt man fest, daß Teilgraphen der linken und rechten Seite von der Transformation unberücksichtigt bleiben. Dieses bringt uns zum Konzept des Klebegraphen, der auch als Kontextgraph oder Interfacegraph bezeichnet wird. Der Klebegraph ist Teilgraph der linken und rechten Seite der Regel und wird durch die Anwendung der Regel nicht verändert.

Eine Regelanwendung sieht wie folgt aus:

1. Löschen der linken Seite bis auf den Klebegraphen.
2. Hinzufügen der rechten Seite bis auf den Klebegraphen.

Mit diesem Konzept ist das Problem der Einbettung der rechten Seite in den Graphen gelöst, da der Klebegraph nicht aus dem Graphen entfernt wird. Ist der Klebegraph leer, so wird die linke Seite aus dem Graphen entfernt, wobei hängende Kanten auftreten können. Wie dieser Fall gehandhabt wird, hängt vom verwendeten Ansatz ab. Der Begriff Ansatz, nicht zu verwechseln mit Ansatzsuche, definiert, wie eine Regel angewendet wird. Entweder werden hängende Kanten einfach gelöscht (z.B. Single-Pushout [EHK+97]) oder durch Bedingungen verboten (z.B. Double-Pushout [CMR+97]). In dieser Arbeit werden hängende Kanten durch eine Kontaktbedingung verboten.

Definition 2-3: Graphregel

Eine Graphregel r ist ein Tupel bestehend aus:

- Einer linken Seite L (engl. left-hand side),
- Einem Klebegraphen K (engl. gluing graph oder interface graph) mit $L \supseteq K \subseteq R$ und
- Einer rechten Seite R (engl. right-hand side).

Eine andere Möglichkeit der Definition ist folgende:

Eine Graphregel ist ein Tupel bestehend aus:

- Drei unabhängigen Graphen L, K und R und
- Zwei injektive Graphenmorphismen m_l, m_r, mit $L \xleftarrow{m_l} K \xrightarrow{m_r} R$.

Die zweite Definition wird im algebraischen Ansatz der Graphersetzung benutzt. Sie ist als äquivalent zu der ersten Definition zu sehen, besitzt aber den Vorteil, daß die Entitäten L, K und R unabhängig voneinander sind. Dadurch wird eine Implementierung einfacher, da keine gemeinsamen Knoten und Kanten verwaltet werden müssen, weshalb u.a. diese Definition hier vorgezogen wird.

Definition 2-4: Regelanwendung bzw. direkte Ableitung

Seien r eine Graphregel mit $r = (L, K, R, m_l, m_r)$ und G ein Graph.

Sei m ein Graphenmorphismus $m:L \rightarrow G$, der die folgenden Bedingungen erfüllt:

1. Die *Kontaktbedingung* ist erfüllt, wenn für alle $e \in E_G - m_E(E_L)$ aus $source_G(e) \in m_V(V_L)$ bereits $source_G(e) \in m_V(V_K)$ und aus $target_G(e) \in m_V(V_L)$ bereits $target_G(e) \in m_V(V_K)$ folgt.
2. Die *Identifizierungsbedingung* ist erfüllt, wenn für alle Knoten $v_1, v_2 \in V_L$ mit $v_1 \neq v_2$ und $m_V(v_1) = m_V(v_2)$ gilt: $v_1, v_2 \in V_K$ und für alle Kanten $e_1, e_2 \in E_L$ mit $e_1 \neq e_2$ und $m_E(e_1) = m_E(e_2)$ gilt: $e_1, e_2 \in E_K$.

Dann läßt sich der Graph H durch Anwenden der Regel r im Ansatz m direkt ableiten:

1. Den Kontextgraphen D erhält man durch Entfernen der linken Seite L bis auf den Klebegraphen K.
   $V_D = V_G - (m_V(V_L - m_{lV}(K)))$
   $E_D = E_G - (m_E(E_L - m_{lE}(K)))$
2. Den Graphen H erhält man, indem alle Elemente der rechten Seite R hinzugefügt werden, die nicht im Klebegraphen K vorhanden sind.
   $V_H = V_D + (V_R - m_{rV}(K))$
   $E_H = E_D + (E_R - m_{rE}(K))$
   $source_H : E_H \rightarrow V_H$
   $$source_H(e) = \begin{cases} source_D(e) & e \in E_D \\ d_V(source_R(e)) & e \in (E_R - m_{rE}(K)) \\ source_R(e) & \text{sonst} \end{cases}$$
   $target_H : E_H \rightarrow V_H$
   $$target_H(e) = \begin{cases} target_D(e) & e \in E_D \\ d_V(target_R(e)) & e \in (E_R - m_{rE}(K)) \\ target_R(e) & \text{sonst} \end{cases}$$
   $label_H : V_H \cup E_H \rightarrow \Sigma$
   $$label_H(e) = \begin{cases} label_D(e) & e \in E_D \\ label_R(e) & e \in E_H - E_D \end{cases}$$
   $$label_H(v) = \begin{cases} label_D(v) & v \in V_D \\ label_R(v) & v \in V_H - V_D \end{cases}$$

Die direkte Ableitung läßt sich durch ein Diagramm (siehe Abbildung 2-5) veranschaulichen. Im Diagramm sind die Morphismen zwischen den einzelnen Graphen dargestellt, wobei m_l, m_r und m gegeben sind. Der Morphismus d:K→D ist die Einschränkung von m auf Elemente aus dem Klebegraphen K. m* ist durch d und die Abbildung der Elemente aus R-m_r(K) auf Elemente aus H-D gegeben. Die Graphenmorphismen m_l* und m_r* lassen sich durch Inklusion herleiten.

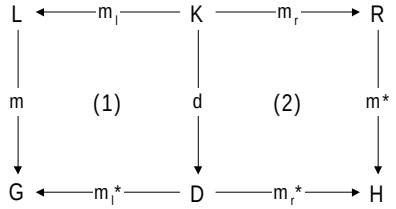


Abbildung 2-5: Diagramm der Regelanwendung[6]

Die untere Zeile gibt dabei die Veränderung des Graphen an. G ist der Originalgraph. Der Zwischengraph D ergibt sich aus G, indem die linke Seite bis auf den Klebegraphen entfernt wird. H entsteht durch Hinzufügen der rechten Seite bis auf den Klebegraphen zu D.

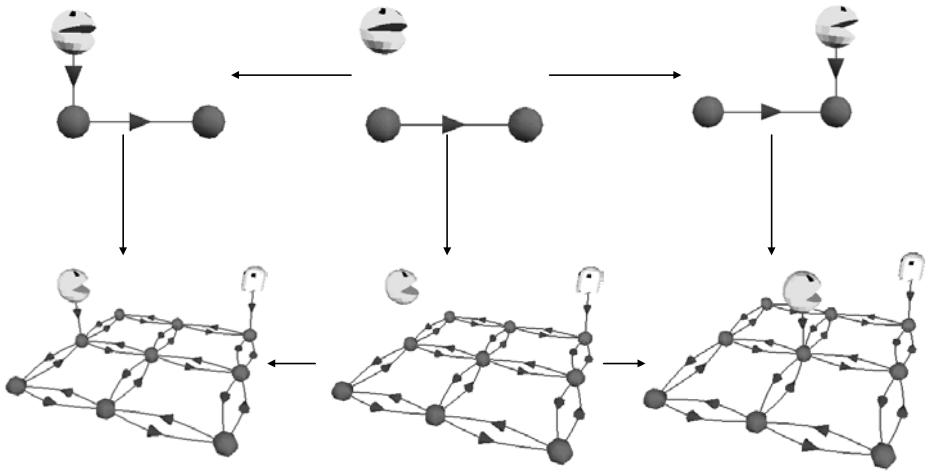


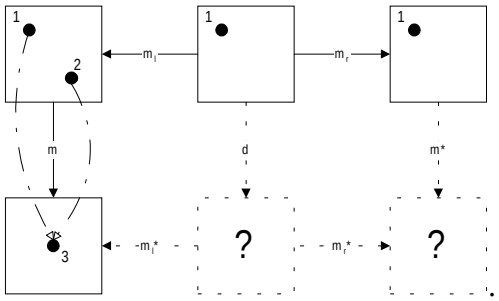
Abbildung 2-6: Diagramm für die Regelanwendung aus Abbildung 2-3

Ist H durch direktes Ableiten aus G durch Regel r im Ansatz m entstanden, so schreibt man dieses als: $G \underset{r,m}{\Rightarrow} H$. Ist es nicht wichtig, wo oder welche Regel angewendet wurde, dann schreibt man: $G \underset{r}{\Rightarrow} H$, $G \underset{P}{\Rightarrow} H$ oder $G \Rightarrow H$. P ist dabei eine Regelmenge, in der die tatsächlich angewendete Regel enthalten ist.

[6] Bei dem Diagramm handelt es sich um ein sog. Double-Pushout Diagramm. Da hier nicht die zugrundeliegende Kategorientheorie verwendet wird, wird es einfach als Diagramm der Regelanwendung bezeichnet und auf eine detailliertere Einführung verzichtet.

Die Kontaktbedingung sichert zu, daß beim Löschen keine hängenden Kanten entstehen. Für jeden Knoten der gelöscht wird, müssen auch alle ein- und ausgehenden Kanten gelöscht werden.

Die Identifikationsbedingung sichert zu, daß nur Teile des Klebegraphen durch den Morphismus miteinander verschmolzen werden können. Damit ist sicher, daß alle Elemente gelöscht werden, die von der Regel spezifiziert wurden. Dadurch wird zugesichert, daß keine undefinierten Zustände, wie in folgender Situation, entstehen:



Im Beispiel ist der Zwischengraph D nicht definiert, da der Knoten 1 nicht erhalten bleibt. Somit kann kein Morphismus d:K→L gefunden werden.

Kontakt- und Identifizierungsbedingung zusammen garantieren, daß die Einschränkungen von $source_G$, $target_G$ und $label_G$ auf V_D und E_D wieder Abbildungen sind. Dadurch ist gesichert, daß D ein Teilgraph von G ist.

Nach der Definition der Regelanwendung können Markierungen von Klebeknoten und -kanten nicht verändert werden. Um Markierungen zu ändern, muß das Element gelöscht und wieder hinzugefügt werden, was nicht immer möglich ist. Um dennoch ein Ummarkieren durchführen zu können, werden Markierungen als spezielle Knoten aufgefaßt, die über Kanten mit Knoten verbunden sind.

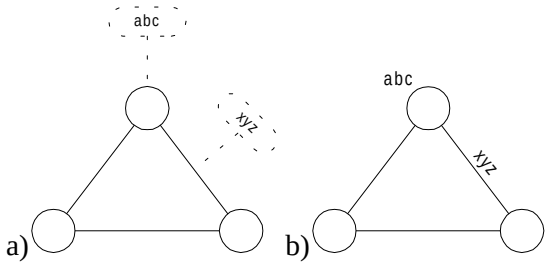


Abbildung 2-7: a) Interne Sicht eines Graphs
b) Externe Sicht des Graphen aus a)

Abbildung 2-7 zeigt die zwei Sichten auf den Graphen. In der internen Sicht liegen Markierungen als spezielle Knoten und Kanten vor, die in der externen Sicht verborgen sind. Dort tauchen sie als "normale" Markierung der Knoten auf. Mit Hilfe dieser Modellierung können auch Markierungen von Elementen des Klebegraphen verändert werden, ohne daß sich nach außen hin etwas ändert.

Wenn mehrere Regelanwendungen hintereinander ausgeführt werden, so spricht man von einer Ableitung.

Definition 2-5: Ableitung

Eine Ableitung ist eine Sequenz von direkten Ableitungen: $G_0 \underset{r1,m1}{\Rightarrow} G_1 \underset{r2,m2}{\Rightarrow} ... \underset{rk,mk}{\Rightarrow} G_k$. Man schreibt auch kurz $G_0 \overset{*}{\underset{P}{\Rightarrow}} G_k$ oder $G_0 \overset{*}{\Rightarrow} G_k$.

2.4 Graph-Grammatiken

Eine Graph-Grammatik faßt verschiedene Komponenten zu einem System zusammen. Dieses System enthält hinreichend Informationen um Ableitungen zu starten. Die Ableitungen beginnen dabei in einem Startgraphen und enden in Graphen, die eine terminale Markierung aufweisen. Es werden nur Regeln aus der Grammatik benutzt, um die Graphen abzuleiten.

Die Ergebnisse, d.h. die abgeleiteten terminalen Graphen, werden in der Graphsprache festgehalten.

Definition 2-6: Graph-Grammatik

Sei Σ ein Markierungsalphabet.

Eine Graph-Grammatik ist ein System GraGra=(N, T, P, I), wobei die Komponenten folgende Bedeutung haben:

- N ist eine Menge von nichtterminalen Zeichen mit $N \subseteq \Sigma$,
- T ist eine Menge von terminalen Zeichen mit $T \subseteq \Sigma$,
- P ist eine Menge von Regeln, deren linke Seiten mindestens ein nichtterminales Zeichen enthält und
- I ist der Startgraph.

Die Forderung, daß jede linke Seite einer Regel mindestens ein nichtterminales Zeichen enthält, liegt an der engen Verwandtschaft mit Chomsky-Grammatiken. Graph-Grammatiken stellen die Verallgemeinerung dieser Grammatiken dar. So läßt sich jede Chomsky-Grammatik in eine Graph-Grammatik umsetzen (siehe dazu [HK92], Teil 2, Seite 3).

Gegenüber den Chomsky-Grammatiken macht die Einschränkung der linken Seiten der Regeln nur dann Sinn, wenn die enge Beziehung zwischen den beiden Grammatiken aufrecht erhalten werden soll. Ansonsten kann und sollte die Einschränkung wegfallen, da Graphregeln mit einer leeren linken Seite und deren Anwendung auf einen Graphen wohldefiniert ist. Es wird kein Teilgraph ersetzt, sondern nur etwas hinzugefügt. Bei Chomsky-Grammatiken ist dieser Fall undefiniert, da auf Zeichenketten gearbeitet wird.

Definition 2-7: Graphsprache

Sei GraGra=(N, T, P, I) eine Graph-Grammatik.

Dann beinhaltet die *ausschöpfende Graphsprache* S(GraGra) alle aus dem Startgraphen ableitbaren Graphen:

$$S(GraGra) = \left\{ G \,\middle|\, I \overset{*}{\underset{P}{\Rightarrow}} G \right\}.$$

Die von der Grammatik *erzeugte Graphsprache* L(GraGra) enthält alle Graphen aus S(GraGra), die nur mit terminalen Zeichen markiert sind:

$$L(GraGra) = \left\{ G \in S(GraGra) \,\middle|\, \forall v \in V_G : label_G(v) \in T \wedge \forall e \in E_G : label_G(e) \in T \right\}.$$

Mit den Begriffen der Graph-Grammatik und der von ihr erzeugten Graphsprache sind wir nun in der Lage, ein Graphersetzungssystem zu spezifizieren und auszuwerten. Betrachten wir dazu das Beispiel der wohlstrukturierten Flußdiagramme.

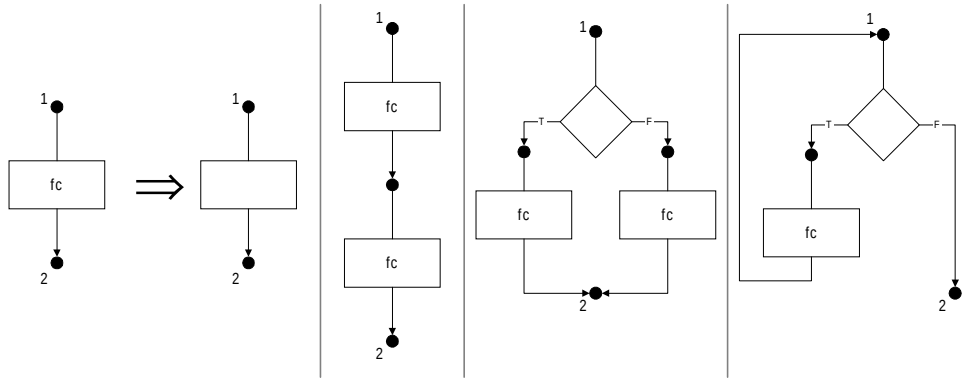


Abbildung 2-8: Wohlstrukturierte Flußdiagramme (aus [HK92], Teil 1, Seite 32)

Der initiale Graph ist dabei identisch mit der linken Seite aus Abbildung 2-8. Die Markierung fc ist als nichtterminales Zeichen zu sehen und alle anderen als terminale. Der Klebegraph ist durch die Knoten 1 und 2 gegeben. Im Vergleich dazu die Chomsky-Grammatik in Backus-Nauer-Form: S ::= V:=E | S;S | if B then S else S fi | while B do S od.

3 GRACE

3.1 Vorbereitung

Wir sind nun in der Lage, eine abstrakte Maschine zu beschreiben, welche die erzeugte Graphsprache einer Graph-Grammatik aufzählt. Die Maschine erhält dazu als Eingabe eine Graph-Grammatik. Intern führt sie Ableitungen durch und gibt jeden abgeleiteten Graph mit einer terminalen Markierung aus. Einen Prototyp der abstrakten Maschine findet man in [HK92]:

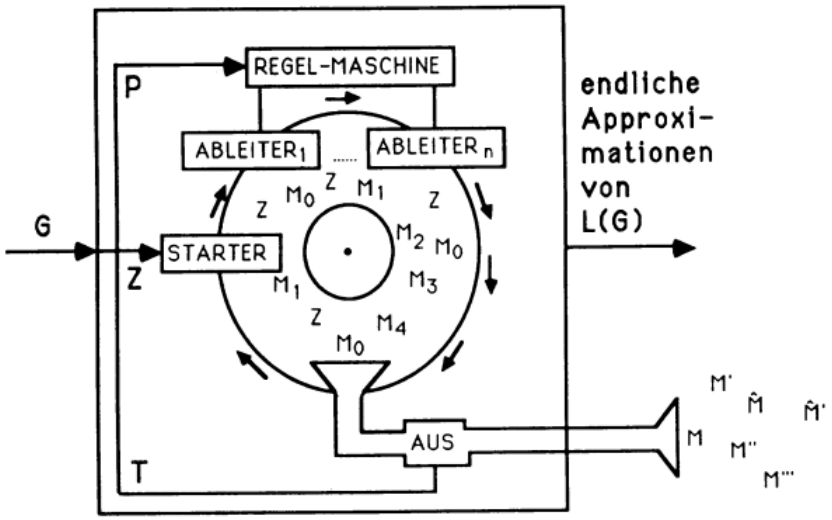


Abbildung 3-1: Graph-Grammatik Maschine (aus [HK92], Teil 2, Seite 1)

Wenn man die Maschine laufen läßt, werden immer neue Graphen produziert und ausgegeben. Nach und nach entstehen so immer größere endliche Approximationen einer möglicherweise unendlichen Graphsprache. Dazu wird vorausgesetzt, daß alle möglichen Regelanwendungen irgendwann einmal stattfinden. Ein Beispiel einer unendlichen Graphsprache sind die wohlstrukturierten Flußdiagramme aus Abbildung 2-8. Dabei kann es passieren, daß der Benutzer sehr lange warten muß, bevor der erste Graph ausgegeben wird, was u.a. durch die zufällige Auswahl der Regeln bedingt ist.

Der schwache Punkt der Graph-Grammatik Maschine ist das Fehlen eines Kontrollflusses, mit dessen Hilfe die Auswahl von Regeln eingeschränkt werden könnte. Dadurch kann kein Einfluß auf die Reihenfolge der Regelanwendung genommen werden, was für manche Berechnungen wichtig ist. So kann es passieren, daß die Maschine oft an- und ausgeschaltet werden muß, bis die korrekte Reihenfolge vorliegt.

Dieses führt uns zu den programmierten Graphersetzungssystemen, welche über einen Kontrollfluß die Regelauswahl einschränken. Als Konsequenz der Einschränkung der Regelauswahl kann z.B. die Anzahl von Regelanwendungen festgelegt werden. Soll zuerst Regel 1 einmal und dann Regel 2 einmal angewendet werden, so muß die Regelauswahl im Schritt 1 auf Regel 1 und im Schritt 2 auf Regel 2 begrenzt werden. Dadurch ergibt sich die gewünschte Bedingung. Der Benutzer wird allerdings nicht mit dieser low-level Programmierung konfrontiert, sondern beschreibt die Kontrollbedingung in einer Sprache, was z.B. so aussehen kann: apply once rule1; apply once rule2.

Lagen die Programme bisher als lose Sammlung von Regeln vor, so bieten programmierte Graphersetzungssysteme Mechanismen zur Strukturierung, wie z.B. Prozeduren, die von Berechnungen abstrahieren.

Programmierte Graphersetzungssysteme sind eine vollständige Programmiersprache, wie C++ oder Prolog. Der Unterschied besteht darin, daß das zugrundeliegende Paradigma neu ist und dadurch die Strukturen ungewohnt sind.

3.2 Die Sprache GRACE

Die Sprache GRACE wird seit ca. 5 Jahren von Forscherinnen und Forschern aus Aachen, Bremen, Berlin und Leiden entwickelt. Das Ziel war es, eine neue Spezifikations- und Programmiersprache zu entwickeln. Die Sprache sollte dabei unter anderem folgende Features enthalten:

- Regelbasierte Graphersetzung,
- Unterstützung verschiedener Graphenklassen,
- Ansatzunabhängigkeit,
- Kontrollbedingungen und
- Strukturierungskonzepte.

In die Sprache fließen dabei Erfahrungen aus bestehenden Systemen, wie z.B. PROGRES ([Sch94]) und AGG ([Rud97]), ein. Es wurde nicht versucht ein System zu entwickeln, das auf den bestehenden aufbaut, sondern es wurde ein komplett neues entworfen. Die wichtigsten Errungenschaften von GRACE sind die Offenheit gegenüber verschiedenen Graphersetzungs-ansätzen und das Modulkonzept, was bislang einzigartig bei Graphersetzungssystemen ist.

Grundlage von GRACE sind Graphregeln, die atomare Einheiten darstellen. Nur Regeln können den Zustand eines Graphen verändern. Alle anderen Mechanismen dienen entweder der Komplexitätskontrolle oder zur Steuerung der Regelanwendung.

Transformationseinheiten (engl. transformation units) dienen der Komplexitätskontrolle von Programmen. Sie abstrahieren von einer Berechnung, die in einem initialen Zustand startet und in einem terminalen endet und identifizieren diese mit einem Namen. Die interne Ableitungssequenz wird durch Kontrollbedingungen überwacht, wodurch der Nichtdeterminismus bei der Regelauswahl eingeschränkt wird. An welcher Stelle eine Regel auf einen Graphen angewendet wird, entzieht sich der Kontrolle des Benutzers.

Transformationseinheiten können auf bereits bestehende zurückgreifen, d.h. sie können andere Einheiten aufrufen.

Eine wichtige Erweiterung dieses Konzeptes stellen Module dar, die Transformationseinheiten zu Einheiten zusammenfassen. Module weisen eine Import- und Exportstruktur auf, so daß Programmteile importiert und somit wiederverwendet werden können. Ein Programm kann nun aus vielen einfachen Teilen aufgebaut werden.

Wie wichtig Module sind, zeigen Spezifikationen in PROGRES, die selbst bei kleinen Systemen an die 40 Seiten lang sind ([Sch98]). Module reduzieren nicht den Umfang, sondern helfen, das Programm für den Benutzer übersichtlicher zu gestalten. Module werden im Bereich der Programmiersprachen seit langem intensiv benutzt. Die Argumente für die Einführung von Modulen lassen sich auch auf GRACE anwenden, weshalb hier auf eine Wiederholung verzichtet werden soll.

3.3 Graphersetzungsansatz

Um die Anforderungen von GRACE zu erfüllen, benötigen wir den Begriff eines allgemeingültigen, abstrakten Graphersetzungsansatzes. Dieser faßt Klassen von Graphen, Regeln, einen Anwendungsoperator, Graphklassenausdrücke und Kontrollbedingungen zusammen. Die Intention von Graphklassenausdrücken ist es, Mengen von Graphen zu beschreiben, was u.a. für die initialen und terminalen Graphen von Transformationseinheiten benötigt wird. Kontrollbedingungen sollen die nichtdeterministische Regelauswahl einschränken.

Den abstrakten Ansatz kann man als generisches Objekt auffassen, daß später durch konkrete ersetzt wird. Basis ist der Anwendungsoperator $\Rightarrow$, der eine binäre Relation auf Graphen für alle Regeln bildet. Dieses ist nur möglich, da alle Graphansätze einen Anwendungsoperator (vgl. u.a. [ER97], [DKH97]) definieren.

Definition 3-1: Graphersetzungsansatz (graph transformation approach)

Ein Graphersetzungsansatz $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$ besteht aus:

- einer Klasse von Graphen $\mathcal{G}$,
- einer Klasse von Regeln $\mathcal{R}$,
- einem Anwendungsoperator $\Rightarrow$, welcher eine binäre Relation auf Graphen für alle $r \in \mathcal{R}$ definiert, d.h. $\underset{r}{\Rightarrow} \subseteq \mathcal{G} \times \mathcal{G}$,
- einer Klasse von Graphklassenausdrücken $\mathcal{E}$, so daß für alle $e \in \mathcal{E}$ $SEM(e) \subseteq \mathcal{G}$ gilt und
- einer Klasse von Kontrollbedingungen $\mathcal{C}$ über einer Menge von Bezeichnern ID, so daß $C \in \mathcal{C}$ eine binäre Relation $SEM_E(c) \subseteq \mathcal{G} \times \mathcal{G}$ für jede Abbildung $E:ID \rightarrow 2^{\mathcal{G} \times \mathcal{G}}$ bildet[7]. Eine Funktion $names:C \rightarrow 2^{ID}$, die jeder Kontrollbedingung die Menge der benutzten Bezeichner zuordnet.

[7] Mit 2^M wird die Potenzmenge einer Menge M bezeichnet.

Die Abbildung E:ID$\rightarrow 2^{\mathcal{G}\times\mathcal{G}}$ wird als Umgebung (engl. environment) bezeichnet und bildet jeden Bezeichner auf eine binäre Relation auf Graphen ab. Die Kontrollbedingungen werden in Bezug zu einer Umgebung gesetzt, die den verwendeten Bezeichnern (names(C)) eine Bedeutung zuordnet. Dabei wird die aktuelle Umgebung meistens so definiert, daß sie Regeln auf $\Rightarrow_r$ und Transformationseinheiten auf deren Semantik SEM(t) abbildet.

Sei ID={rule1, rule2, rule3} und $\mathcal{C}$ gleich der Menge der regulären Ausdrücke REG(ID) ([AEH+96], Seite 9) über ID, dann können Kontrollbedingungen wie z.B. rule1;rule2;rule3* geschrieben werden. Diese Kontrollbedingung bedeutet, daß zunächst rule1 ausgeführt wird, dann rule2 und dann beliebig oft rule3. Daß Kontrollbedingungen nicht immer eine endliche Sequenz von Ableitungen beschreiben, zeigt das Beispiel. Wenn die linke Seite der Regel rule3 leer ist, kann die Sequenz beliebig lang werden.

Üblicherweise wird zur Definition von Kontrollbedingungen eine Sprache über die Bezeichner ID benutzt. Weitere Beispiele von möglichen Kontrollbedingungen findet man in [KK98] und [AEH+96].

Graphklassenausdrücke, oder kurz Graphausdrücke, werden benutzt, um Mengen von Graphen zu beschreiben, die bestimmte Eigenschaften aufweisen. Es existieren verschiedene Arten, die Teilklassen zu beschreiben. So kann jeder Ausdruck, der eine Teilmenge von $\mathcal{G}$ beschreibt, als Graphausdruck benutzt werden. Eine andere Möglichkeit, eine Menge von Graphen zu beschreiben ist, sie aufzuzählen ($\{G_0, ..., G_k\}\subseteq \mathcal{G}$), was für eine kleine Anzahl an Graphen gut funktioniert. Diese Methode wird meistens dann eingesetzt, wenn man bestimmte Graphen z.B. als initialen oder terminalen erzwingen möchte. Wenn $\mathcal{G}$ die Klasse der markierten Graphen über einem Alphabet Σ ist, dann kann jede Teilmenge $L\subseteq\Sigma$ als Graphausdruck dienen:

$$SEM(L)=\{G\in \mathcal{G} \mid G \text{ ist nur mit Elementen aus L markiert}\}.$$

Möchte man alle Graphen zulassen, so kann man den Ausdruck "all" definieren: SEM(all) = $\mathcal{G}$.

3.4 Transformationseinheiten

Eine Transformationseinheit läßt sich dann über diesen Ansatz wie folgt definieren.

Definition 3-2: Transformationseinheit (transformation unit)

Eine Transformationseinheit über einen Ansatz $\mathcal{A}$ ist ein System tunit=(I, U, R, C, T), wobei:

- $I\in \mathcal{E}$ die Menge der initialen Zustände beschreibt,
- U eine endliche Menge von benutzten Transformationseinheiten über $\mathcal{A}$ ist,
- $R\subseteq\mathcal{R}$ eine endliche Menge von Regeln ist,
- $C\in \mathcal{C}$ eine Kontrollbedingung ist und
- $T\in \mathcal{E}$ die Menge der terminalen Zustände beschreibt.

Man beachte, daß nach Definition von U keine zwei Ansätze miteinander gemischt werden können. Interessant wäre die Frage, ob es möglich wäre bzw. sinnvoll ist, verschiedene Ansätze zu mischen. Wenn ja, wie könnte diese Mischung aussehen? Im Bereich von Programmiersprachen ist dieses Problem bei Typen als Typumwandlung bekannt. Lösungen sehen entweder eine automatische Konvertierung durch den Übersetzer oder eine explizite Konvertierung durch den Programmierer vor. Wie die Theorie von Graphen gezeigt hat, lassen sich Erkenntnisse eines Graphersetzungansatzes nicht ohne weiteres auf einen anderen übertragen. Eine automatische Konvertierung wird deshalb nicht generell möglich sein. In Einzelfällen ist eine Konvertierung sicherlich möglich.

Bei Definition 3-2 handelt es sich um eine iterative Definition von Transformationseinheiten, d.h. initiale Transformationseinheiten besitzen keine Importe (U=∅), auf denen andere dann aufbauen können. Die Semantik einer Transformationseinheit ist eine binäre Relation auf Graphen, die Paare von Graphen (G, G') enthält. G ist dabei ein initialer und G' ein terminaler Graph, wobei G' durch Ableiten aus G entstanden ist und das Paar (G, G') von der Kontrollbedingung erlaubt ist.

Definition 3-3: Interleaving-Semantik einer Transformationseinheit

Sei tunit=(I, U, R, C, T) eine Transformationseinheit über einen Ansatz $\mathcal{A}$. Sei names(C) in der disjunkten Vereinigung von U und R enthalten, E(tunit):ID$\rightarrow 2^{\mathcal{G}\times\mathcal{G}}$ die Umgebung mit E(tunit)(r)= $\Rightarrow_r$ für r∈R, E(tunit)(t)=SEM(t) für t∈U und E(tunit)(x)=∅ sonst. Dabei sei die Interleaving-Semantik SEM(t)$\subseteq\mathcal{G}\times\mathcal{G}$ für alle t∈U definiert (azyklische Uses-Struktur).

Die Interleaving-Semantik SEM(tunit) besteht dann aus allen Paaren (G, G'), so daß:

1. G∈SEM(I) und G'∈SEM(T),
2. Es existiert eine Sequenz von Graphen $G_0,...,G_n\in\mathcal{G}$: $G=G_0\Rightarrow...\Rightarrow G_n=G'$, wobei $(G_i,G_{i+1})\in\Rightarrow_R$ oder $(G_i,G_{i+1})\in$SEM(t) mit t∈U, und
3. (G, G')∈$SEM_{E(tunit)}$(C) gilt.

Sequenzen wie in Punkt 2. werden Interleaving-Sequenzen genannt. Ein Schritt entspricht entweder einer Regelanwendung oder einem Aufruf einer Transformationseinheit. Durch den iterativen Aufbau der Transformationseinheiten läßt sich die Interleaving-Sequenz zu einer Sequenz von direkten Ableitungen expandieren (vgl. Abbildung 3-2). Hier verbirgt sich der Begriff der Ableitung, wie er in Definition 2-5 eingeführt wurde.

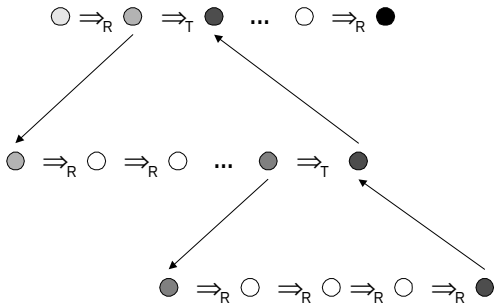


Abbildung 3-2: Expandierte Interleaving-Sequenz

Punkt 3. sichert zu, daß das Paar (G, G') von der Kontrollbedingung zugelassen ist, wodurch die Auswahl der Interleaving-Sequenz eingeschränkt wird. Im Kapitel 4.4.1 wird auf Kontrollbedingungen näher eingegangen.

3.5 Module

Bisher bestehen GRACE-Programme aus einer Sammlung von Transformationseinheiten über einen Ansatz $\mathcal{A}$. Ein großer Nachteil ist, daß alle Transformationseinheiten sichtbar sind, was dazu führt, daß auch Hilfseinheiten sichtbar sind. Hilfseinheiten sind Unterprogramme, die benötigt werden, um eine Berechnung zu definieren. Diese haben nur in einem bestimmten Zusammenhang eine Relevanz und sollten deshalb für Außenstehende nicht sichtbar sein.

Es besteht zwar eine Struktur innerhalb der Transformationseinheiten, doch sie selbst liegen als lose Ansammlung vor. Dabei werden alle Arten von Einheiten zusammengemischt. Im Hinblick auf eine Programmier- und Spezifikationssprache ist es essentiell, einen Mechanismus zu haben, der die Einheiten strukturiert. Dieser Mechanismus sollte es ermöglichen, seine Programme aus einzelnen kleinen Teilen aufzubauen, wobei der Zugriff auf einzelne Komponenten kontrolliert (information hiding) werden kann.

Module sind ein solches Konzept. Module bestehen aus zwei Teilen: der Schnittstelle und der Implementierung. Die Schnittstelle gibt den Zugriff auf bestimmte Komponenten des Moduls frei, indem Namen von Transformationseinheiten und/oder Regeln exportiert werden. Andere Module können die exportierten Komponenten verwenden, sehen aber deren Implementierung nicht. Im Implementierungsteil werden die Komponenten definiert. Darüber hinaus kann dieser Teil zusätzliche Transformationseinheiten besitzen, die nach außen nicht sichtbar sind.

Definition 3-4: Modul[8]

Ein Modul ist ein Tripel mod=(IM, N, DEF, EX), wobei:

- $IM \subseteq ID$ die Menge der importierten Namen,
- $N \subseteq ID$ die Menge der definierten Namen ist
- $DEF: N \rightarrow \mathcal{P} \cup T_{\mathcal{A}}$ eine Abbildung, die jedem Namen $ide \in N$ eine Regel oder eine Transformationseinheit zuordnet,
- $EX \subseteq N$ die Menge der exportierten Namen ist mit $EX \neq \emptyset$ (und damit auch $N \neq \emptyset$),

und außerdem folgendes gilt:

$N \cap IM = \emptyset$ und $used(mod) \subseteq N \cup IM$, mit $used(mod) = \bigcup_{tunit:(I, R, C, T, DEF, N) \in DEF} (names(C) - N_{tunit})$.

[8] Grundlage der Definition ist der Artikel [HHK98]. Abweichend von diesem Vorschlag werden lokale Regeln in Transformationseinheiten, mit einem üblichen Gültigkeitsbereich (siehe [Lou94], Seite 140), zugelassen.

Ein fundamentaler Abstraktionsmechanismus in Programmiersprachen ist die Benutzung von Namen, denen eine bestimmte Semantik zugeordnet wird. Die Bindung eines Namens an eine Transformationseinheit oder Regel wird durch die Abbildung DEF vorgenommen. Dadurch kann nun über Namen auf Transformationseinheiten oder Regeln zugegriffen werden.

Dabei haben sich allerdings die Transformationseinheiten ein wenig geändert. In der Definition 3-2 wurde die Menge der benutzten Transformationseinheiten U pro Einheit aufgeführt, was durch das Modulkonzept nicht mehr notwendig ist. Diese werden jetzt durch den Import IM des Moduls repräsentiert.

Definition 3-5: Transformationseinheit (transformation unit) eines Moduls

Eine Transformationseinheit ist ein Tupel $tunit_{mod}$=(I, R, C, T, DEF, N), wobei:

- $I \in \mathcal{E}$ die Menge der initialen Zustände beschreibt,
- $R \subseteq \mathcal{R}$ eine endliche Menge von Regeln ist,
- $C \in \mathcal{C}$ eine Kontrollbedingung ist,
- $T \in \mathcal{E}$ die Menge der terminalen Zustände beschreibt,
- $N \subseteq ID$ die Menge der definierten Namen ist und
- $DEF: N \rightarrow R$ eine Bindung von Namen zu lokalen Regeln vornimmt.

DEF wird in einer Transformationseinheit durch DEF' überschrieben, um einen lexikalischen Gültigkeitsbereich zu definieren. Besitzt eine lokale Regel den gleichen Namen, wie eine vorher vorgenommene Bindung, so wird der Name entsprechend der lokalen Zuordnung interpretiert. Nach Beendigung der Transformationseinheit gelten wieder die vom Modul vorgenommenen Bindungen. Die Abbildung DEF' ist wie folgt definiert:

$$DEF'_{tunit} : N_{tunit} \cup N_{mod} \rightarrow \mathcal{R} \cup T_{\mathcal{A}}$$

$$DEF'_{tunit}(x) = \begin{cases} DEF_{tunit}(x) & x \in N_{tunit} \\ DEF_{mod}(x) & x \in N_{mod} - N_{tunit} \end{cases}.$$

Wenn der Bezeichner x lokal definiert ist, so handelt es sich dabei um eine Graphregel $DEF_{tunit}(x) \in R_{tunit}$, da nur sie lokal definiert werden kann. Ansonsten wurde der Bezeichner vom Modul definiert. Importierte Namen müssen gesondert behandelt werden, da sie nicht durch die Funktion DEF des Moduls erfaßt werden. Deren Semantik wird durch eine spezielle Umgebung verwaltet.

Die Interleaving-Semantik einer Transformationseinheit kann nun wie folgt definiert werden:

Definition 3-6: Interleaving-Semantik einer Transformationseinheit eines Modules

Sei $tunit_{mod}$ = (I, R, C, T, DEF, N) eine Transformationseinheit in einem Modul mod.

Sei $E:IM_{mod} \rightarrow 2^{\mathcal{G} \times \mathcal{G}}$ beliebig gewählt, sei dann die Umgebung $E(tunit):ID \rightarrow 2^{\mathcal{G} \times \mathcal{G}}$ gegeben durch:

$$E(tunit)(x) = \begin{cases} \Rightarrow_{DEF'_{tunit}(x)} & DEF'_{tunit}(x) \in \mathcal{R} \\ SEM(DEF'_{tunit}(x)) & DEF'_{tunit}(x) \in T_{\mathcal{A}} \\ E(x) & x \in IM_{mod} - N_{tunit} \end{cases}.$$

Dabei sei die Interleaving-Semantik $SEM(t) \subseteq \mathcal{G} \times \mathcal{G}$ für alle von tunit benutzten Transformationseinheiten t aus dem Modul mod definiert. Die Interleaving-Semantik SEM(tunit) besteht dann aus allen Paaren (G, G'), so daß:

1. $G \in SEM(I)$ und $G' \in SEM(T)$,
2. Es existiert eine Sequenz von Graphen $G_0,...,G_n \in \mathcal{G}$: $G = G_0 \overset{n1}{\Rightarrow} G_1 \overset{n2}{\Rightarrow} ... \overset{nk}{\Rightarrow} G_k = G'$, wobei $(G_i, G_{i+1}) \in \overset{n}{\Rightarrow} = E(tunit)(n)$ mit $n \in N_{mod} \cup IM_{mod} \cup N_{tunit}$ und
3. $(G, G') \in SEM_{E(tunit)}(C)$ gilt.

3.6 GRACE-Programme

Ein GRACE-Programm besteht aus einer Menge von Modulen, so daß jeder Import einen korrespondierenden Export besitzt. Gestartet wird ein Programm durch den Aufruf einer exportierten Transformationseinheit bzw. Regel.

Abbildung 3-3 zeigt ein GRACE-Programm zur Berechnung des Fibonacci-Baumes. In der Abbildung wurde die in Kapitel 5.2.3 vorgestellte Visualisierung benutzt. Sie stellt Module als Text dar, in dem Bilder von verwendeten Graphregeln integriert werden.

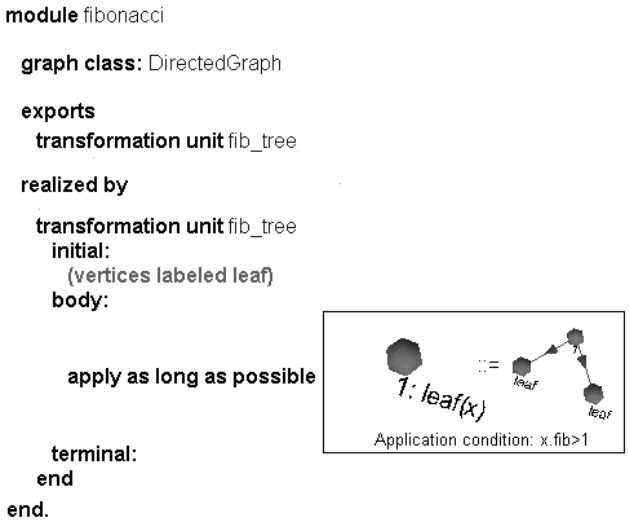


Abbildung 3-3: GRACE-Programm

Im Definitionsteils des Moduls fibonacci wird zunächst der verwendete Graphersetzungsansatz angegeben (graph class). Unter dem Stichwort 'exports' sind die exportierten Namen aufgeführt. Die Funktion DEF ist implizit vorhanden und wird durch Bezeichner vor den exportierten Namen definiert. So handelt es sich bei dem Namen fib_tree um eine Transformationseinheit. Im Implementierungsteil, eingeleitet durch 'realized by', wird die Transformationseinheit fib_tree programmiert. Auf nähere Details wird an dieser Stelle verzichtet, da sie im weiteren Schritt für Schritt eingeführt werden.

4 IMPLEMENTIERUNG

4.1 Struktur

Die Allgemeinheit von GRACE und die hochgesteckten Ziele machen eine Implementierung nicht leicht. Die Implementierung muß so offen sein, daß verschiedene Ansätze und Graphklassen benutzt werden können und gleichzeitig eine effiziente Ausführung möglich ist.

Als Programmiersprache wurde C++ gewählt, was vor allem wegen der Wiederverwendung von bereits existierenden Komponenten (u.a. dreidimensionale Visualisierung) geschah. Ziel der Implementierung ist es, ein objektorientiertes Framework zu erstellen, das an die jeweiligen Bedürfnisse von speziellen Graphklassen angepaßt werden kann. Bei der Implementierung wurde Wert auf die Nähe zur Theorie gelegt, d.h. Einheiten der Theorie sollten sich im Code wiederfinden. Dieses ist vor allem für die spätere Erweiterung von großer Bedeutung.

Grundlage der Sprache GRACE ist der abstrakte Graphersetzungsansatz, der verschiedene Einheiten zu einem System zusammenfaßt. In der Implementierung wird das System in zwei Module aufgespalten, um eine strikte Trennung zwischen Operationen auf Graphen und GRACE-Programmen zu gewährleisten. Der Vorteil liegt in der Kompaktheit der einzelnen Module, die so einfacher gewartet bzw. erweitert werden können.

Insgesamt besteht die Implementierung aus folgenden Modulen:

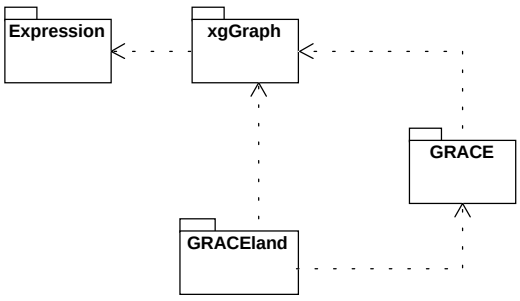


Abbildung 4-1: Modulstruktur der Implementierung

Die Grundlage bildet dabei das Modul xgGraph, welches die verschiedenen Graphklassen implementiert. Das Modul Expression stellt einen Interpreter für Ausdrücke zur Verfügung, so daß Berechnungen auf Attributen durchgeführt werden können. GRACE implementiert die eigentliche Graphsprache. Auf der obersten Ebene befindet sich das Modul GRACEland, welches eine integrierte Entwicklungsumgebung zur Verfügung stellt.

4.2 Modul xgGraph

Das Modul xgGraph implementiert ein abstraktes Graphenmodell, das die Grundlage für die weitere Arbeit bildet. Das Modell bildet dabei Graphenmorphismen, Graphen, Graphregeln und die Anwendung von Regeln auf Graphen ab.

Beim Design ist darauf zu achten, daß konkrete Graphklassen effizient in das vorgegebene Modell integriert werden können – Effizienz im Sinne von Einfachheit der Erweiterung und Schnelligkeit des erweiterten Codes. Eine einfache Erweiterung kann dadurch gesichert werden, indem nur eine minimale Schnittstelle vorgegeben wird. Gleichzeitig wird dadurch vermieden, daß abgeleitete Klassen in ein zu enges Schema gepreßt werden. Somit können abgeleitete Klassen effiziente Algorithmen, z.B. zur Morphismensuche implementieren.

Die Definition von Basisklassen ist relativ einfach, da man für jede Entität nur eine abstrakte Klasse bereitstellen muß. Schwieriger wird dann die Auswahl von Attributen und Methoden der Klassen. Sie müssen so definiert sein, daß sie Allgemeingültigkeit besitzen.

Die Basisstruktur des Moduls sieht wie folgt aus:

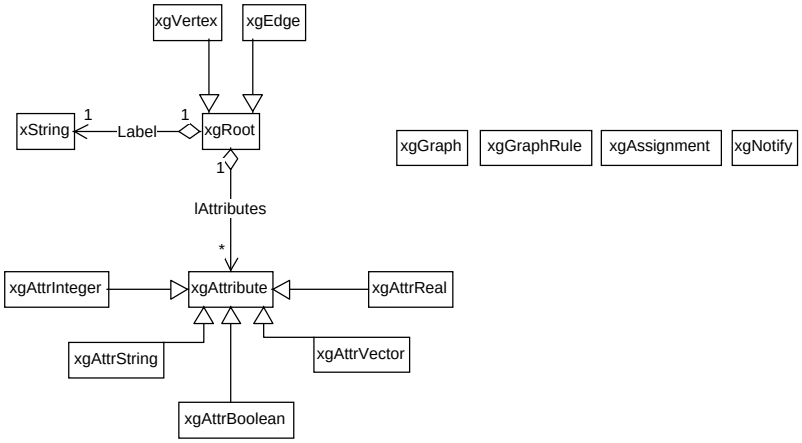


Abbildung 4-2: Basisklassen des Moduls xgGraph

Die Klasse xgRoot bildet die Basis für Kanten und Knoten. In ihr wurden deren gemeinsame Attribute zusammengefaßt. So hat jeder Knoten und jede Kante eine Markierung und eine Liste von Attributen. Es ist zu beachten, daß man sich dadurch nicht auf eine bestimmte Klasse von Graphen festgelegt hat. Je nach Graphklasse kann man die Information entweder nutzen oder auch nicht. Da die meisten in der Literatur verwendeten Ansätze Markierungen und Attribute für Knoten und Kanten benutzen, wurden diese mit in die Basisklasse aufgenommen, um eine effiziente Verwaltung zu ermöglichen. Dieses Konzept ist sehr flexibel und vielfältig verwendbar. Es wird u.a. eingesetzt, um zusätzliche Informationen für die Visualisierung zu speichern, wie es vom Modul GRACEland benötigt wird.

Ein Attribut wird durch einen Namen und einen Wert beschrieben. Dabei werden folgende Typen unterschieden:

- Bool'sche Wahrheitswerte (xgAttrBoolean),
- Fließkommazahlen (xgAttrReal),
- Ganze Zahlen (xgAttrInteger),
- Zeichenketten (xgAttrString) und
- n-dimensionale Vektoren von Fließkommazahlen (xgAttrVector).

Die übrigen Klassen werden in den nächsten Kapiteln näher erläutert.

4.2.1 Eventhändler

Eine Anforderung an die Visualisierung war, daß auch Zwischenschritte von Graphersetzungen angezeigt werden sollten. Damit die Visualisierung weiß, wann die Darstellung erneuert werden muß, wurde ein Eventhändling implementiert. Nachdem ein Zwischenschritt beendet wurde, wird eine Nachricht (Event) an den entsprechenden Händler geschickt. Dieser kann nun die Nachricht verarbeiten, indem z.B. die graphische Ausgabe aktualisiert wird.

Dem Senden liegt ein synchrones Message-Passing Verfahren zugrunde, d.h. der Sender wird so lange blockiert, bis der Empfänger es verarbeitet hat. Asynchrone Verfahren können auf dieses Verfahren aufgesetzt werden. Implementiert wird die Nachrichtenübermittlung durch die Klasse xgNotify. Sie stellt eine Methode Signal zur Verfügung, über die Events verschickt bzw. empfangen werden. Senden entspricht dem Aufruf der Methode und Empfangen dem Aktivieren durch einen Aufruf.

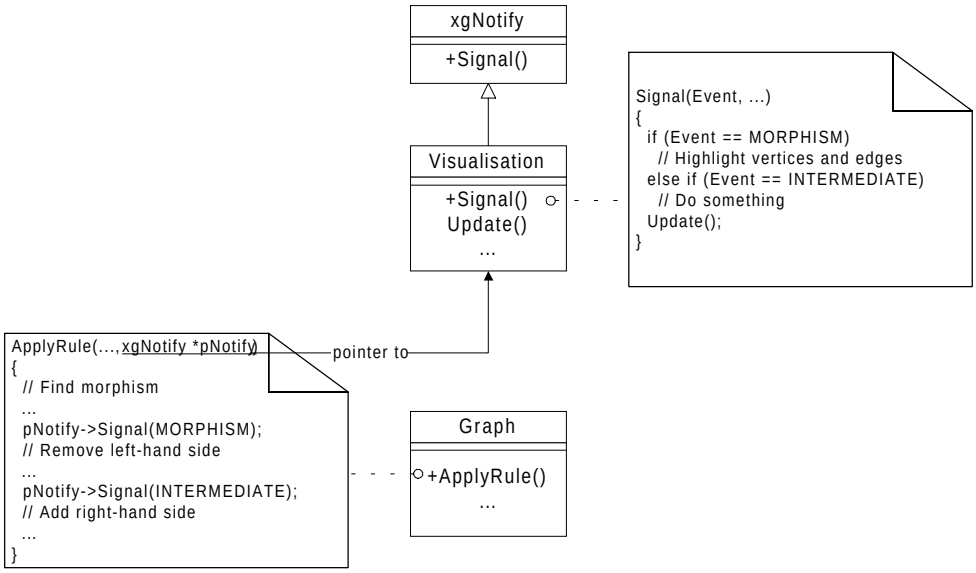


Abbildung 4-3: Beispiel Eventhändling

Abbildung 4-3 zeigt, wie das Eventhändling eingesetzt wird. Ein Eventhändler leitet sich von xgNotify ab und überschreibt die Methode Signal. Dort werden die gewünschten Events (MORPHISM, INTERMEDIATE) abgefangen und in bestimmte Aktionen umgesetzt. Wenn nun eine Regel auf einen Graphen angewendet wird, so bekommt sie einen Zeiger auf einen Eventhändler, z.B. Visualisierung, übergeben, wodurch Nachrichten an diesen geschickt werden können. Nachdem ein Morphismus gefunden wurde, sendet die Methode ApplyRule ein Signal an die Visualisierung (pNotify->Signal(MORPHISM)), die ihrerseits entsprechende Kanten und Knoten markiert. Nachdem dieses geschehen ist, wird mit der Anwendung der Regel fortgefahren.

Als Parameter werden der Methode Signal das Event und maximal vier zusätzliche Parameter übergeben. Das Modul xgGraph definiert folgende grundlegende Events:

Event	Bedeutung	1	2	3	4
xgEVENT_RULE_MORPHISM	Ein Graphenmorphismus wurde gefunden.	Graphregel	Morphis-mus	-	-
xgEVENT_RULE_INTERMEDIATE	Ein Zwischenschritt bei der Regelanwendung	Graphregel	Morphis-mus	-	-
xgEVENT_RULE_EXECUTED	Die Regelanwendung ist abgeschlossen.	Graphregel	Morphis-mus		-
xgEVENT_LAYOUT_INITIAL	Initiale Konfiguration des Graphen wurde erreicht.	Zeiger auf den Graphen	-	-	-
xgEVENT_LAYOUT_ITERATION	Ein Layoutschritt ist beendet.	Zeiger auf den Graphen	-	-	-
xgEVENT_LAYOUT_TERMINAL	Terminale Konfiguration des Graphen wurde erreicht.	Zeiger auf den Graphen	-	-	-
Ab xgEVENT_USER	Benutzer definierte Events.				

4.2.2 Graphen

Graphen werden durch die Klasse xgGraph abgebildet. Konkrete Klassen leiten sich von dieser ab und füllen sie mit entsprechend weiterer Funktionalität.

Als einziges Attribut wird dem Graphen seine Graphklasse zugeordnet, um zur Laufzeit unterscheiden zu können, um welche Art es sich handelt. Da nicht alle Typen im voraus bekannt sind, wird der Typ durch eine Zeichenkette abgebildet. Dabei muß vom Programmierer darauf geachtet werden, daß nicht zwei verschiedene Graphklassen denselben Namen benutzen. Dieses sollte allerdings nicht vorkommen, da den verschiedenen Klassen schon in ihrer Definition eindeutige Namen zugeordnet werden.

In der Klasse werden folgende Methoden definiert:

- Clone - Erzeugt eine Kopie des Graphen,
- Clear - löscht alle Knoten und Kanten des Graphen,
- Empty - testet, ob der Graph leer ist oder nicht,
- ApplyRule - wendet eine Regel an,
- Match - sucht nach einem Graphenmorphismus,
- CompleteMatch - vervollständigt einen Teilmorphismus,
- GetType - gibt die Graphklasse zurück und
- Is - testet, ob die Graphklasse der übergebenen entspricht.

Die wichtigste Methode der Klasse ist ApplyRule, die die Schnittstelle zur Anwendung einer Regel auf den Graphen definiert. Es existieren zwei Ausprägungen dieser Methode. In der ersten Version wird der Methode eine Graphregel und ein Morphismus übergeben. Dabei wird vorausgesetzt, daß der Morphismus wohldefiniert ist (vgl. nächstes Kapitel). Dann wird die Regel unter dem Morphismus ausgeführt.

In der zweiten Version wird nur die anzuwendende Graphregel übergeben. Jetzt wird intern nach einem oder mehreren Morphismen gesucht und die Regel dort angewendet. Dazu wird ein zusätzlicher Parameter angegeben, der Auskunft gibt, wie eine Regel angewendet werden soll:

- Fair, d.h. der Morphismus wird zufällig ausgewählt,
- First, d.h. der erste Morphismus wird benutzt oder
- Parallel, d.h. alle gleichzeitig möglichen Morphismen werden benutzt.

Dabei wird durch diese Modi nicht genau festgelegt, wie sie implementiert werden. Dieses bleibt der konkreten Klasse überlassen. So gibt es z.B. für Fair verschiedene Verfahren, die sich hauptsächlich in ihrer Komplexität und ihrem Laufzeitverhalten unterscheiden.

Beide Versionen der Methode ApplyRule bekommen als Parameter einen Zeiger auf einen Eventhändler, der die empfangenen Events in Aktionen umsetzt. Der Rückgabewert gibt Aufschluß darüber, ob die Ableitung erfolgreich war oder was für ein Fehler aufgetreten ist.

Zur Ansatzsuche stehen die Funktionen Match und CompleteMatch zur Verfügung. Match sucht nach einen Graphenmorphismus für die linke Seite einer Regel. Welcher Morphismus schließlich gewählt wird, ist für den Benutzer nicht steuerbar. Das Ziel von CompleteMatch ist es, einen Teilmorphismus zu vervollständigen. Die Teilzuordnung kann z.B. durch den Benutzer geschehen, indem er manuell Teile der linken Seite zuweist. Dadurch wird die Auswahl der Morphismen eingeschränkt und für den Benutzer steuerbar.

4.2.3 Graphenmorphismus

Morphismen werden zweigeteilt implementiert, wobei der eine Teil die vom anderen vorgenommenen Zuordnungen speichert. Die Speicherung der Zuordnungen geschieht durch die Klasse xgAssignment. Erzeugt werden Morphismen in von xgGraph abgeleiteten Klassen, was die enge Verbundenheit zwischen Graphen und Morphismen widerspiegelt.

Nach Definition bestehen Morphismen m aus zwei Funktionen m_V und m_E, die Elemente aus einer Menge auf Elemente einer anderen abbilden. Die Funktionen lassen sich als eine Liste von Zuordnungen (S/D) implementieren, so daß S auf D abgebildet wird. Wird die Klasse xgAssignment als Morphismus für eine Regelanwendung verwendet, so muß der Erzeuger die Vorschriften aus Definition 2-2 oder dem Ansatz entsprechender Definition sicherstellen.

Die Klasse xgAssignment weist folgende Struktur auf:

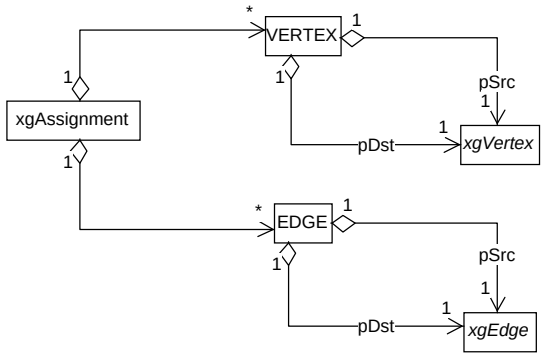
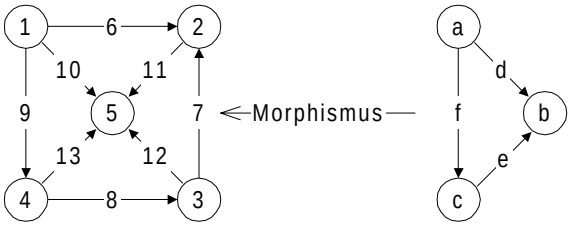


Abbildung 4-4: Struktur der Klasse xgAssignment

Der Name xgAssignment hat seinen Ursprung in den Zuweisungen (engl. to assign) der Knoten und Kanten.

Beispiel:

Gegeben seien folgende Graphen, wobei die Zahlen und Buchstaben keine Markierungen darstellen, sondern die Elemente der Graphen mit Namen versehen:



Gesucht ist ein Morphismus vom rechten Graphen in den linken.

Ein möglicher Morphismus ist durch ⟨Knoten: (a/3), (b/5), (c/2); Kanten: (d/12), (e/11), (f/7)⟩ gegeben.

4.2.4 Graphregeln

Die Klasse xgGraphRule definiert eine abstrakte Graphregel basierend auf der Definition 2-3 von Seite 13:

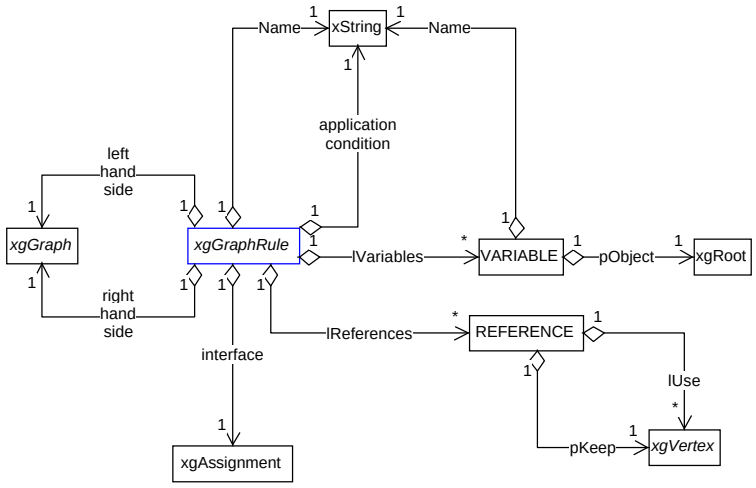


Abbildung 4-5: Struktur einer Graphregel

Jede Graphregel besitzt eine linke und rechte Seite. Sie werden durch einen Graphen repräsentiert, der sich von der Basisklasse xgGraph ableitet. Ein expliziter Klebegraph, wie in der Definition, existiert nicht mehr. Er wurde durch eine Zuordnung von Elementen aus der rechten Seite und der linke Seite ersetzt. Der Klebegraph ist implizit in dieser Konstruktion enthalten, d.h. er kann aus diesen Informationen erzeugt werden. Diese Konstruktion basiert auf der Tatsache, daß der Klebegraph in der linken und rechten Seite enthalten ist:

$$L \supseteq K \subseteq R \text{ bzw. } L \xleftarrow{m_l} K \xrightarrow{m_r} R .$$

Wenn eine Regel ausgeführt wird, so benötigt man den Klebegraphen nicht explizit, sondern nur die Information, ob ein Knoten oder eine Kante Teil des Klebegraphen ist. Im Endeffekt wird so ein Graph bzw. Morphismus eingespart und dadurch die Handhabung von Graphregeln vereinfacht.

Anwendungsbedingung

Eine Graphregel besitzt zudem noch eine Anwendungsbedingung, die spezielle Eigenschaften fordert, d.h. die möglichen Morphismen, unter denen eine Regel ausgeführt werden kann, eingeschränkt. Eine Bedingung ist ein Ausdruck, der von dem Modul Expression ausgewertet werden kann. Welche Form der Ausdruck hat, hängt von der verwendeten Bibliothek ab.

Variablen

Damit in der Anwendungsbedingung überhaupt auf die Attribute eines Knoten oder einer Kante zugegriffen werden kann, wird das Konzept der Variablen eingeführt. Eine Variable ist ein Platzhalter für einen Knoten oder eine Kante auf der linken Seite einer Regel. Mit Hilfe des Punktoperators kann man auf die Attribute zugreifen. So kann z.B. die Markierung eines

Knotens mit der Variablen x durch x.label abgefragt werden. Dabei wird das Bild des Knoten oder der Kante benutzt. D.h. nachdem ein Morphismus gefunden wurde, wird auf die Attribute des Bildes unter dem Morphismus zugegriffen: x.label bedeutet dann m(x).label. In einigen Fällen ist es aber auch wichtig, auf die Attribute des Originals[9] zugreifen zu können, was durch Anhängen von ".org" geschehen kann ("x.label.org").

Wenn ein Morphismus für eine Regel mit Variablen gefunden wurde, wird als nächstes eine Tabelle mit den aktuellen Werten der Attribute initialisiert. Dabei wird für jedes Attribut eines Knotens oder einer Kante, welcher bzw. welche mit einer Variablen versehen ist, je ein Eintrag unter dem Namen "Variablenname.Attributname" und "Variablenname.Attributname.org" erzeugt. In der Anwendungsbedingung kann über die Variablen auf die Werte der Knoten und Kanten zugegriffen werden. Einzelheiten des Ausdruckshändlers werden im Kapitel 4.3 beschrieben.

Referenzen

Eine Erweiterung gegenüber Definition 2-3 stellt das Konzept der Referenzen dar, die es erlauben, ein- und ausgehende Kanten von Knoten zu sichern und wiederherzustellen. Dadurch kann man zerstörerische Graphregeln schreiben, die sich später wieder mit dem restlichen Graphen verbinden.

Betrachten wir dazu ein Beispiel:

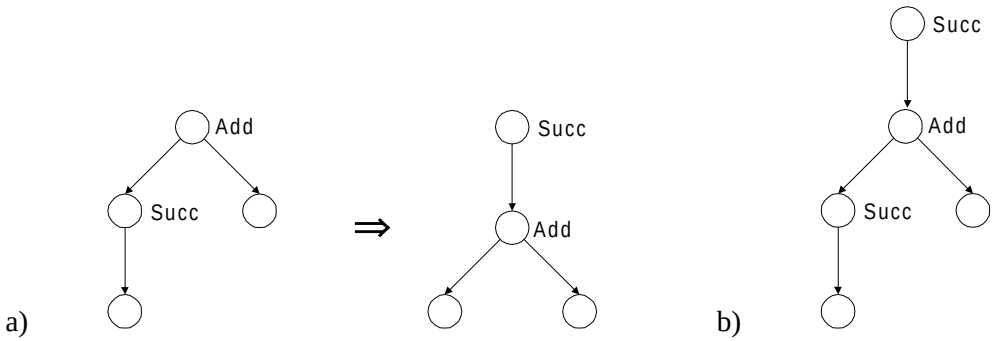


Abbildung 4-6: a) Regel Add(Succ(x),y)::=Succ(Add(x,y)) ([], Seite)
b) Beispielgraph

Diese Regel läßt sich auf den Graphen aus Abbildung 4-6 b) auf Grund von einer hängenden Kante (von Succ zu Add) nicht anwenden. Unter Umständen läßt sich diese Situation durch Modifizierung der Regel umgehen, doch nicht immer wird der gewünschte Effekt erzeugt oder ist ein Umschreiben der Regel möglich, wie z.B. bei der Regel Add(0, x) ::= x. Die Regel aus Abbildung 4-6 darf nicht nur die Markierungen des Add Knotens ändern, sondern muß die Knoten erhalten, da sie u.U. spezielle Attribute besitzen, die nicht verloren gehen dürfen.

Graphisch läßt sich das Konzept wie folgt visualisieren:

[9] Das Original ist ein Knoten oder eine Kante der linken Seite der Graphregel.

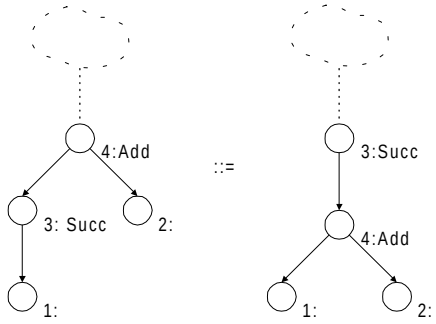


Abbildung 4-7: Visualisierung von Referenzen

Nachdem ein Ansatz für die linke Seite gefunden wurde, werden alle Referenzen zu Add gesichert. Jetzt wird die linke Seite aus dem Graphen entfernt und die rechte hinzugefügt. Als letztes werden Succ alle gesicherten Referenzen zugewiesen. Durch die so geschriebene Regel bleiben alle Knoten erhalten und werden nur neu miteinander verbunden. Dabei ist gesichert, daß die Attribute der Knoten vor und nach der Ausführung dieselben sind. Mögliche Anwendungsgebiete sind Term Graph Rewriting ([AEH+96]) oder Programmoptimierungen.

4.2.5 Implementierung von gerichteten Graphen

Dieses Kapitel ist der Implementierung von gerichteten Graphen gewidmet. Grundlage ist dabei die Definition 2-1 von Seite 9.

Um effiziente Algorithmen implementieren zu können, muß die zugrunde liegende Datenstruktur optimal sein. Im Zusammenhang mit Graphen kann Effizienz als die Zeit angesehen werden, mit der man den Graphen durchlaufen kann. In vielen Algorithmen ist es wichtig, von einem Knoten alle Nachbarn zu erreichen bzw. die ein- und ausgehenden Kanten zu ermitteln. Hier bietet sich eine Darstellung durch Adjazenzlisten, wie in Abbildung 4-8, an. Ein Graph besteht aus zwei Listen, eine für Knoten und eine für Kanten. Jeder Knoten besitzt ferner eine Liste von ein- und ausgehenden Kanten. Eine Kante wird durch ein Tupel von Knoten repräsentiert. Dadurch, daß nur Referenzen benutzt werden, kann ohne Zeitverlust auf das Ziel einer Kante zugegriffen werden. Es muß kein Umweg über die Knotenliste gemacht werden.

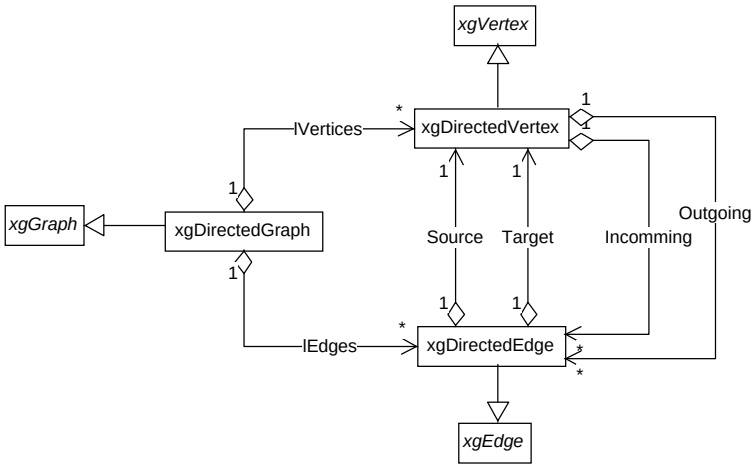


Abbildung 4-8: Gerichtete Graphen

Die Ansatzsuche geschieht durch einen erweiterten Backtracking-Algorithmus. Die Erweiterung besteht darin, daß Grapheigenschaften früh geprüft werden, um so den Suchraum zu verkleinern. Der Algorithmus erzeugt zunächst Zuweisungen für alle Knoten und ordnet danach die Kanten zu. Ein Knoten v1 wird einem Knoten v2 nur dann zugewiesen, wenn die Markierungen gleich sind und die Zahl der eingehenden und ausgehenden Kanten von v2 größer oder gleich der von v1 ist. Nachdem die Zuordnung durchgeführt wurde, werden Verbindungen überprüft. D.h. existiert im Graphen der linken Seite einer Regel eine Verbindung zwischen v1 und einem schon zugewiesenen Knoten v, so muß diese auch unter dem Morphismus vorhanden sein. Dadurch wird vermieden, daß weit auseinander liegende Knoten benutzt werden, die keine Beziehung zueinander haben. Eine weitere Optimierung besteht darin, daß Knoten mit vielen Kanten zuerst geprüft werden, da es wahrscheinlicher ist, daß weniger Knoten mit vielen Kanten existieren, als Knoten mit wenigen, wodurch sich der Suchraum weiter einschränken läßt.

Die Ansatzsuche stoppt, sobald ein Graphenmorphismus gefunden wurde. Fair wird die Ansatzsuche dadurch, daß die Knoten- und Kantenliste zufällig sortiert werden. Welche Knoten und Kanten gewählt werden, ist trotz der Sequentialität der Tiefensuche nicht mehr bestimmbar und damit liegt eine faire Auswahl des Morphismus vor. Der Vorteil dieser Methode ist die Vermeidung zusätzlicher Datenstrukturen und führt damit zu einer Reduzierung der Komplexität des Algorithmus. Das Sortieren der Liste geschieht nach folgendem Algorithmus:

```
Input: a list (aList)
Output: random sorted list (RandomOrder)

e = Elements of list 'aList';
for i=0 to e do begin
   RandomOrder.InsertFirst( aList[ Random(Elements of list 'aList') ] );
   aList.RemoveActual();
end
```

Das Sortieren der Liste beansprucht kaum Zeit und macht sich erst bei Zehntausenden von Knoten im Sekundenbereich bemerkbar. Die Laufzeit der Algorithmus hängt vom Operator [ ] ab, der das i-te Element einer Liste zurückgibt. In der bisherigen Implementierung wird die verkettete Liste durchlaufen und die besuchten Elemente mitgezählt. Je weniger Elemente durchlaufen werden müssen, desto schneller geschieht die Sortierung. Begrenzt man die Auswahl des nächsten Elementes auf n-Elemente, so reduziert sich zwar die Fairness, aber dafür kann die Laufzeit weiter optimiert werden:

Random((*Elements of list 'aList'* > n) ? n : *Elements of list 'aList'*)]).

Eine andere Methode, eine faire Ansatzsuche zu implementieren, ist, alle Morphismen zu bestimmen und zufällig einen davon auszuwählen. Für kleine Graphen ist dieses noch durchführbar, doch ansonsten ineffizient. Eine weitere Möglichkeit ist, alle Zuordnungen für einen Knoten zu bestimmen und darunter eine auswählen. Diese Methode ist auf Grund des enormen Speicherbedarfs nicht sinnvoll. Der Sortierungsansatz stellt einen guten Kompromiß zwischen Laufzeit, Komplexität und Fairness dar. In der aktuellen Implementierung wurde auf eine Optimierung verzichtet, da bei Zehntausenden von Knoten Zeiten im Sekundenbereich akzeptabel sind.

Die Attribute von Knoten und Kanten werden nach der Anwendung einer Regel neu bestimmt, indem sie zusammengefaßt und Werte bereits existierender Attribute überschrieben werden. Markierungen werden durch die in der rechten Seite aufgeführten ersetzt, wobei die einzige Ausnahme eine * Markierung ist. Auf der linken Seite einer Regel steht * als Platzhalter für alle im Markierungsalphabet vorkommenden Zeichenketten und auf der rechten Seite für die Erhaltung der aktuellen Markierung.

4.3 Modul Expression

Das Modul Expression wird dazu benutzt, um Ausdrücke auszuwerten. Die Ausdrücke werden eingesetzt, um Werte von Attributen zu berechnen oder die Anwendungsbedingung zu definieren.

Ausdrücke lassen sich in verschiedenen Sprachen definieren, z.B. Perl, Java oder C++. Wegen der Ähnlichkeit zwischen C++ und Java-Ausdrücken, der Einfachheit und ihrer weiten Verbreitung geben wir ihnen den Vorzug.

Ein C++/Java-Ausdruck enthält:

- Konstanten von primitive Typen (u.a. char, short, int, float, double),
- Variablen,
- Arithmetische Operatoren (+, -, *, /),
- Bool'sche Operatoren (&&, ||, !, ==, !=, <, <=, >, >=),
- Operatoren zur Bitmanipulation (&, |, ^, ~) und
- Bedingte Ausdrücke (condition ? then : else).

Ausdrücke können daneben auch Funktionsaufrufe enthalten. Im Hinblick auf die Aufgaben des Ausdruckhändlers werden Operatoren zur Bitmanipulation nicht berücksichtigt. Des weiteren werden nicht alle primitiven Typen unterstützt. Es werden nur die Typen der Attribute unterstützt: integer, real, boolean und string[10]. Ferner sind Operatoren, die Seiteneffekte erzeugen (++, --), nicht erlaubt.

Die Elemente der Ausdrücke lassen sich in fünf Kategorien einteilen: Operatoren, Bedingte Ausdrücke, Konstanten, Variablen und Funktionsaufrufe.

Dieses führt zu folgender Hierarchie:

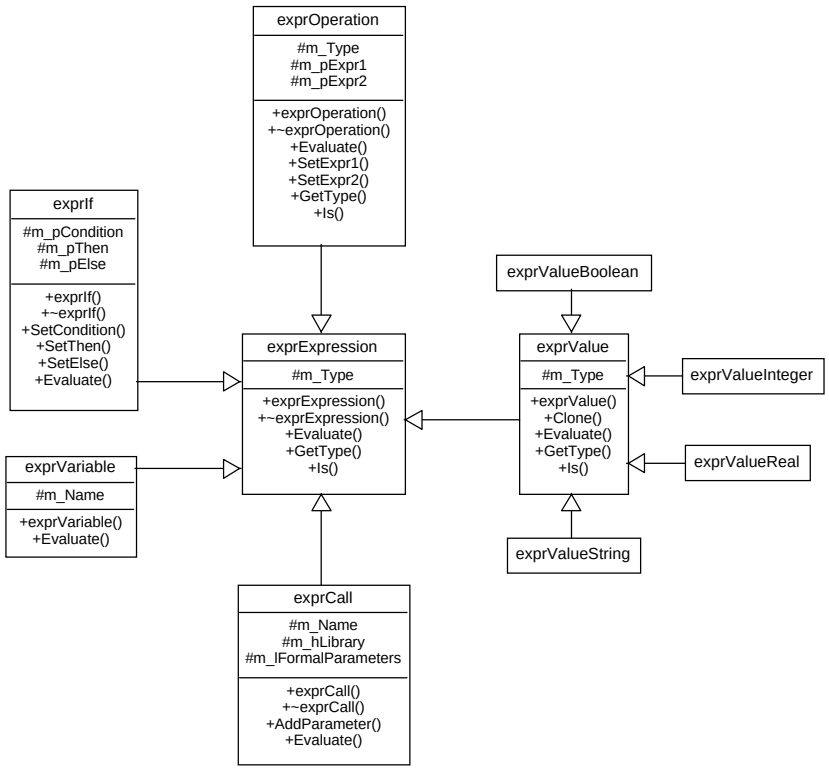


Abbildung 4-9: Kategorien des Expression-Handlers

Die abstrakte Klasse exprExpression definiert gemeinsame Attribute und Methoden der verschiedenen Kategorien. Das einzige Attribut gibt den Typ der Kategorie an (In C++ existiert kein instanceof Befehl wie in Java, so daß man Typinformation manuell verwalten muß). Die Methode Evaluate wird aufgerufen, um den Wert des Ausdrucks zu bestimmen. Sie muß deshalb von den abgeleiteten Klassen überschrieben werden und mit entsprechenden Berechnungen gefüllt werden. Operatoren werden durch die Klasse exprOperation abgebildet. Bei ihnen wurde darauf verzichtet, weitere Unterklassen zu bilden, da sonst die Zahl an Klassen zu groß geworden wäre. Die Klasse exprIf implementiert bedingte Ausdrücke. Für Funktionsaufrufe ist exprCall zuständig. Eine Funktion befindet sich dabei in einer dynamischen Bibliothek (Windows DLL oder Unix ELF), die zur Laufzeit geladen wird. Eine

[10] Der Datentyp vector ist bisher nicht implementiert.

Funktion erhält als ersten Parameter eine Liste mit den aktuellen Parametern aus dem Ausdruck und zusätzlich einen Zeiger auf die aktuelle Umgebung. Der Zugriff auf die Umgebung hilft, einige Ausdrücke elegant zu implementieren (siehe dazu Kapitel 6.4). Die abstrakte Klasse exprValue definiert die Basis für Konstanten, von der sich konkrete ableiten. Variablen werden durch exprVariable implementiert.

Somit ist die interne Struktur des Ausdruckshändlers festgelegt, dessen Schnittstelle zur Außenwelt durch die Klasse exprCPU definiert wird. Nachdem der Klasse exprCPU ein Ausdruck, d.h. eine Zeichenkette, übergeben wurde, wird zunächst der Ableitungsbaum erstellt (siehe Abbildung 4-10). Danach wird der Baum rekursiv abgearbeitet. Als Ergebnis erhält man einen Ausdruck vom Typ exprValue.

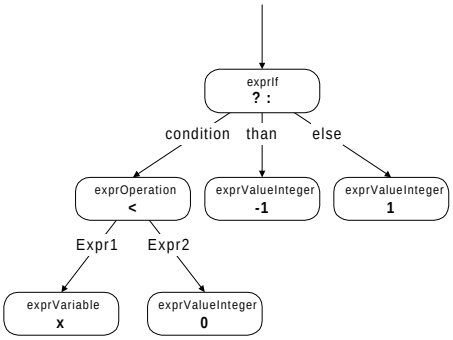


Abbildung 4-10: Ableitungsbaum von (x<0)?-1:1

Die Typkorrektheit eines Ausdruckes wird ausschließlich zur Laufzeit überprüft, was an den untypisierten Variablen liegt. Ein Ausdruck ist typkorrekt, wenn die Typen der Operanden aller Operationen gleich sind. So erzeugt z.B. 1 < 2.0 einen Laufzeitfehler, da 1 vom Typ integer und 2.0 vom Typ real ist. Eine Ausnahme bilden bedingte Ausdrücke, die nur fordern, daß die Bedingung vom Typ boolean ist. Dieses liegt am lazy evaluation Mechanismus, der den then- bzw. else-Teil nur dann auswertet, wenn er benötigt wird.

Variablen werden durch eine Indirektion über eine Tabelle vom Typ exprTable ausgewertet. Dabei wird der Variablen der aktuelle Wert aus der Tabelle zugeteilt. Bevor eine Berechnung gestartet wird, muß also erst die Tabelle mit den Werten der Variablen initialisiert werden. Der Typ der Variablen wird somit erst zur Laufzeit durch den Eintrag in der Tabelle festgelegt.

Die Grammatik der Ausdrücke in EBNF Notation ist.

```
EXPRESSION  ::= CONDITIONAL.
CONDITIONAL ::= OR ["?" EXPRESSION ":" CONDITIONAL].
OR          ::= AND { "||" AND}.
AND         ::= EQUAL { "&&" EQUAL}.
EQUAL       ::= RELATION { ("==" | "!=") RELATION}.
RELATION    ::= ADD { ("<" | ">" | "<=" | ">=") ADD }.
ADD         ::= MULT { ("+" | "-") MULT}.
MULT        ::= UNARY { ("*" | "/") UNARY}.
UNARY       ::= [("-" | "!")] PRIMARY.
PRIMARY     ::= LITERAL | IDENTIFIER | "(" EXPRESSION ")".
LITERAL     ::= NUMBER | STRING | BOOLEAN | FUNCTION.
FUNCTION    ::= "$(" IDENTIFIER "." IDENTIFIER ")" PARAMETER.
```

```
PARAMETER   ::= ["(" EXPRESSION {"," EXPRESSION} ")"].
BOOLEAN     ::= "true" | "false".
```

Es gelten folgende Regelungen:

- Eine Zahl wird als ganze Zahl (exprValueInteger) angesehen, wenn sie keinen Nachkommateil enthält: 0, 1, 2
- Eine Zahl wird zur Fließkommazahl (exprValueReal), wenn sie einen Nachkommateil besitzt: 0.0, 1.0, 2.0
- Die Operatoren <=, >=, -, !, /, * sind auf Zeichenketten nicht definiert. Der Operator + wird als Konkatenation angesehen: "abc" + "def" = "abcdef".
- Die Operatoren ||, &&, !, ? sind nur auf Bool'schen Ausdrücken definiert.

4.4 Modul GRACE

Das Modul GRACE implementiert die Kontrollstrukturen für die Graphersetzung und Strukturierungsmechanismen. Die Kontrollstrukturen entsprechen den Kontrollbedingungen im Graphersetzungsansatz. Strukturierungsmechanismen sind die Transformationseinheiten und Module. Ziel dieses Moduls ist es, einen Interpreter zu definieren, der GRACE-Programme ausführen kann.

4.4.1 Kontrollbedingungen

Kontrollbedingungen schränken die Reihenfolge der Regelanwendung beim Ableiten ein. Bei der Implementierung stellte sich die Frage, welche Kontrollbedingungen nötig sind, um eine ausreichende Kontrolle über die Anwendung der Regeln zu besitzen. Das Interessante an Kontrollbedingungen ist, daß man sie unabhängig von einer bestimmten Graphklasse definieren kann, z.B. als Sprache über dem Alphabet der Bezeichner ID. Der Vorteil ist eine einheitliche Struktur der Programme. Damit kann auch zwischen verschiedenen Graphklassen gewechselt werden, ohne eine komplett neue Sprache lernen zu müssen. Ein Kritikpunkt an GRACE war seine Offenheit gegenüber verschiedenen Ansätzen, was zwangsläufig zu unterschiedlichem Aussehen der Programme und Komplikationen führen sollte. Dadurch, daß nur noch eine Kontrollsprache existiert, besteht der Unterschied nur noch in den Graphklassenausdrücken und in den Regeln. Somit wurde der Unterschied auf ein Minimum reduziert.

Beim Entwurf der Kontrollsprache stand der Aspekt der Einfachheit und Lesbarkeit im Vordergrund. So sollte möglichst darauf verzichtet werden, eine komplett neue Sprache zu entwerfen, sondern vielmehr gängige Konstrukte bestehender Sprachen übernommen werden. Nach längerem Überlegen habe ich mich für eine Kontrollsprache entschieden, die imperative Programmkonstrukte, wie bedingte Anweisungen (if-then-else), Schleifen (while), sequentielle Komposition und Regelanwendung anbietet. Durch die Benutzung einer Sprache bleiben die Kontrollbedingungen erweiterbar, ohne deren Kern zu ändern.

Zur Definition von Kontrollbedingungen wird folgende Sprache zugelassen:

```
CONTROL   ::= STATEMENT {";" STATEMENT }.
STATEMENT ::= CALL | APPLY | IF | WHILE.
CALL      ::= "call" IDENTIFIER.
APPLY     ::= "apply" MODE IDENTIFIER [{("|"|"<") IDENTIFIER}].
MODE      ::= "once" | "as long as possible" |(NUMBER "times").
IF        ::= "if" GCEXPRESSION "then" CONTROL
              ["else" CONTROL] "end".
WHILE     ::= "while" GCEXPRESSION "do" CONTROL "end".
```

Der Befehl *call* ruft in einer externen Bibliothek eine Funktion auf. Damit lassen sich z.B. Dialoge oder Programme ansprechen und Daten zwischen GRACE und Anwendungen austauschen.

Das Kommando *apply* wendet Regeln auf dem Graphen an. Eine Regel läßt sich einmal (*once*), n-mal (*times*) oder solange wie möglich (*as long as possible*) anwenden. Als Parameter erhält dieses Kommando eine Menge von Bezeichnern, aus der die Regel bzw. Transformationseinheit ausgesucht wird. Die Auswahl geschieht entweder zufällig (|) oder die erste Regel/Transformationseinheit, die anwendbar ist (<), wird ausgewählt. Mit Hilfe von < kann man eine Ordnung auf der Menge definieren, was z.B. beim Pacman-Spiel benötigt wurde, um die Regeln korrekt zu implementieren. Die Regel PMkill muß zuerst überprüft werden, da sonst der Pacman dem Geist davonlaufen könnte, was nicht den Spielregeln entspricht.

Die Grammatik läßt eine Mischung der Operatoren "|" und "<" zu, was aber per Software verboten wird. Der angegebene Bezeichner ist entweder eine Regel oder eine Transformationseinheit.

Befehle lassen sich sequentiell komponieren (;). Soll z.B. erst Regel 1 angewendet werden und dann Regel 2, so lautet die Kontrollbedingung: apply once rule1; apply once rule2.

Aktive Kontrollbedingungen

Nach ihrer Definition sind Kontrollbedingungen passiv, da sie nicht direkt in die Ableitung eingreifen. Es wird nur überprüft, ob der Start- und Zielgraph erlaubt ist. Dadurch wird nicht unbedingt nur eine Ableitung festgelegt (Existenzquantor in Definition 3-6, 2.), sondern u.U. auch mehrere. Um den Test (G, G')$\in$ SEM(C) durchzuführen, kann man nicht alle Tupel aus SEM(C) bestimmen, da es unendlich viele Graphen G$\in \mathcal{G}$ gibt. Es wird ein anderes Konzept für die Praxis benötigt als in der Theorie.

Um die Semantik der Sprache festzulegen, wird die Methode der operationellen Semantik verwendet, die von Hennessy und Plotkin ([HP79]) eingeführt wurde.

Die Idee ist es, mit Hilfe einer Transitionsrelation $\rightarrow$ zwischen Konfigurationen die Semantik rekursiv über die Struktur der Sprache zu definieren. Die Transitionsrelation legt dabei fest, wie einzelne Konstrukte von einer abstrakten Maschine abgearbeitet werden. Die Konfigurationen, im folgenden auch als Kontext bezeichnet, sind die Graphen, auf denen gearbeitet wird. Eine Transition der Form: $< S, G > \rightarrow < S', G' >$ bedeutet, daß wenn S in der Konfiguration G ausgeführt wird, die Folgekonfiguration $< S', G' >$ ist.

Definition 4-1: Aktive Kontrollbedingungen

Eine aktive Kontrollbedingung definiert Transitionrelationen $< C, G > \rightarrow < C', G' >$ zwischen zwei Konfigurationen. Eine Konfiguration besteht dabei aus einem Anweisungsteil und einem Kontext. Der Anweisungsteil ist eine Kontrollbedingung und der Kontext ein Graph. Dann kann die Semantik von Kontrollbedingungen über die Struktur von C definiert werden:

$$SEM(C) = \{(G,G') \mid < C, G > \overset{*}{\rightarrow} < \lambda, G' >, \text{ mit } G,G' \in \mathcal{G}\}.$$

Das Aktive an den Kontrollbedingungen ist, daß jetzt Schritt für Schritt die Befehle entsprechend der Kontrollbedingung ausgeführt werden. Gleichzeitig wird dabei die geforderte Interleavingsequenz berechnet und der Test $(G, G') \in SEM(C)$ durchgeführt.

Die Interleavingsequenz ergibt sich aus den Anwendungen der Transitionsrelationen (18) und (19) (siehe Seite 98). Hier wird der Graph entweder durch eine Regel oder eine Transformationseinheit verändert. Nachdem die Sequenz berechnet wurde, muß man nur alle Ableitungsschritte hintereinander aufschreiben und bekommt so den Nachweis der Sequenz. Der Test $(G, G') \in SEM(C)$ wird per Definition durchgeführt.

Für jedes in der Sprache vorkommende Konstrukt müssen Transitionsrelationen definiert werden, die das Sprachkonstrukt abarbeiten. Oftmals werden zwei oder mehr Transitionen benötigt, um die Arbeitsweise vollständig beschreiben zu können. Auszugsweise wird hier die Definition von *apply once* $c_1 |...| c_n$ gezeigt. Die vollständigen Transitionsrelationen sind im Kapitel 8.1 aufgeführt.

Die Transitionsrelationen für *apply once* $c_1 |...| c_n$ lauten:

(1) $<$*apply once* $c_1 \mid ... \mid c_n$, G$> \rightarrow <c_i,$ G$>$, applicable(c_i, G)
(2) $<$*apply once* $c_1 \mid ... \mid c_n$, G$> \rightarrow <\lambda, \perp>$, reduced($\{c_1, ... , c_n\}$, G)

Relation (1) behandelt den Fall, daß die Komponente c_i anwendbar ist. Der Begriff Komponente wird als Platzhalter für eine Transformationseinheit bzw. eine Regel verwendet, wodurch zusätzliche Transitionen vermieden werden. Der Ausdruck wird zu c_i ausgewertet, die dann im nächsten Schritt ausgeführt wird. Ist keine Komponente anwendbar, so enden wir im Fehlerzustand $<\lambda, \perp>$, da gefordert wurde, daß irgendeine Komponente einmal ausgeführt werden soll.

Endet die Berechnung im Fehlerzustand, so bedeutet dieses nicht notwendigerweise, daß keine Transitionssequenz gefunden werden kann. Dieses liegt unter anderem am Vorhandensein von nichtdeterministischen Kontrollstrukturen, wodurch mehrere Berechnungspfade existieren. D.h. der aktuelle Berechnungspfad war eine Sackgasse. Nun muß ein Schritt zurückgegangen werden und die nächste Alternative ausprobiert werden.

Abbildung 4-11 zeigt die möglichen Berechnungspfade für eine Kontrollbedingung. Kann z.B. Regel 3 nur nach Regel 1 ausgeführt werden, so endet der obere Pfad im Fehlerzustand, der untere dagegen nicht.

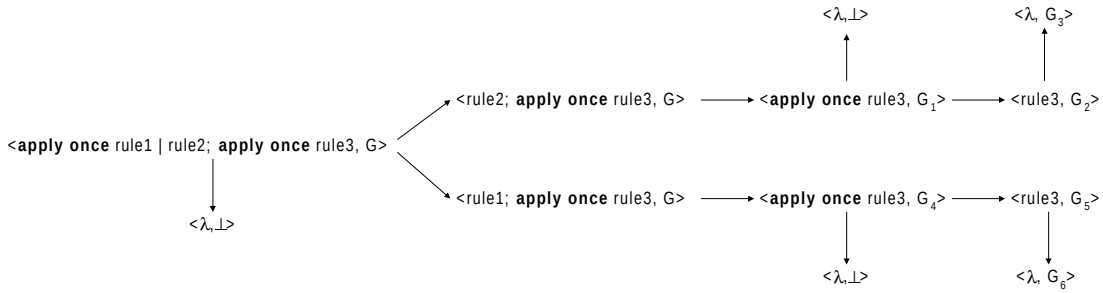


Abbildung 4-11: Transitionssequenz

Durch ein Backtracking-Verfahren werden Schritt für Schritt alle Berechnungspfade abgearbeitet, so daß eine Lösung gefunden wird, falls sie existiert. Diese Aussage trifft nur dann zu, wenn jeder Berechnungspfad von endlicher Länge ist. Ansonsten wird eine unendliche Berechnung gestartet, die nicht in einem Fehlerzustand endet und somit kein Backtracking-Schritt mehr stattfindet. In vielen Fällen existieren unendliche Pfade, so daß dieses Verfahren nicht unbedingt zum Ziel kommt. Oftmals ist Ergebnis der Graphersetzung nicht nur von der Anwendungsreihenfolge der Regeln, sondern auch von den verwendeten Graphenmorphismen abhängig. So kann es sein, daß eine Regel an einer bestimmten Stelle angewendet werden muß, damit die Berechnung nicht in einem Fehlerzustand endet. Durch die Integration der Morphismen in den Backtracking-Schritt steigt die Anzahl der zu durchsuchenden Pfade stark an.

Die meisten Deadlock-Situationen entstehen, wenn die Kontrollstrukturen nicht benutzt werden um Nichtdeterminismus einzuschränken. Müssen Regeln in einer bestimmten Reihenfolge angewendet werden, so sollte die Kontrollbedingung dieses widerspiegeln.

Eine Alternative zu Tiefensuche ist Breitensuche. Hier werden alle möglichen Pfade gleichzeitig untersucht. Zwar ist Unendlichkeit kein Hindernis mehr, doch der Speicher- und Verwaltungsaufwand übersteigt jegliche Grenzen.

Eine komplett andere Lösung ist ein Fail-Stop-Interpreter, der in vorliegenden Implementierung benutzt wird. Endet eine Berechnung in einem Fehlerzustand, so hält der Interpreter mit einer Fehlermeldung an. Dieses Verfahren spiegelt den, trotz der Kontrollstrukturen, verbleibenden Nichtdeterminismus wider und versucht nicht ihn komplett aufzulösen. Der Vorteil liegt in der einfachen Struktur des Interpreters. Ist der Berechnungspfad unendlich, so spielt die zugrunde liegende Architektur keine Rolle. Ein Nachteil ist, daß Probleme nicht gelöst werden können, wenn sie ein Backtracking Verfahren benötigen.

Laufzeitumgebung für Kontrollbedingungen

Eine Kontrollbedingung wird zur Laufzeit in eine Sequenz von Objekten umgewandelt, die dann iterativ abgearbeitet wird.

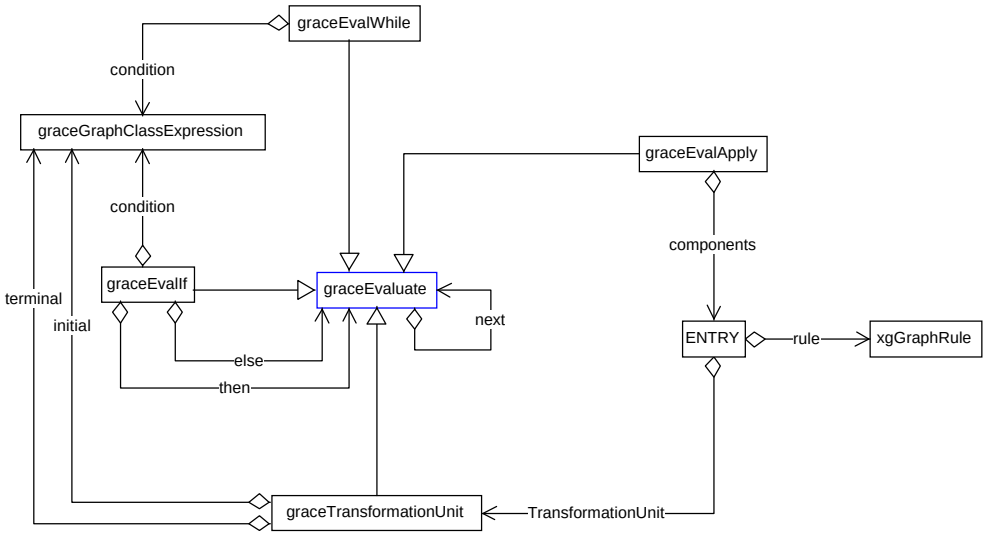


Abbildung 4-12: Laufzeitumgebung für Kontrollbedingungen

Abbildung 4-12 zeigt die Objekte der Laufzeitumgebung und ihre Relationen zueinander. Basis ist die Klasse graceEvaluate, die eine sequentielle Komposition von Objekten ermöglicht. Gleichzeitig definiert sie die Schnittstelle zur Auswertung einer Kontrollbedingung. Jede konkrete Klasse muß diese Schnittstelle überschreiben und mit entsprechender Funktionalität füllen. Die Klasse graceEvalIf z.B. definiert ihre Auswertung wie folgt:

```
PC = null;
if (condition)
   PC = then;
else if (else)
   PC = else
while (PC)
{
   PC.Evaluate();
   PC = PC.next;
}
```

Als erstes wird ermittelt, ob der Graphausdruck wahr oder falsch ist. Je nach Ergebnis wird entweder der then- oder der else-Teil abgearbeitet. Durch die Dezentralisierung muß die anschließende while-Schleife immer dort benutzt werden, wo verschiedene Befehle hintereinander auftauchen können. Durch eine Zentralisierung, in der ein Zeiger auf den nächsten Befehl zurückgegeben wird, könnte man sich die einzelnen while-Schleifen sparen. Dieses macht es jedoch schwieriger die Klasse graceEvalApply zu implementieren, da sie selbst interne Schleifen besitzt. Es müßten zusätzliche Sprungbefehle eingesetzt werden, die an die entsprechenden Stellen zurückspringen. So konnte komplett darauf verzichtet werden und ein einfacher und schneller Parser und Interpreter implementiert werden.

Nachdem ein Programm in seine Laufzeitumgebung umgesetzt wurde, wird die Berechnung durch den Aufruf von Evaluate am Programmanfang gestartet. Dieses kann aber nicht an

beliebiger Stelle geschehen, sondern wird nur auf Transformationseinheiten erlaubt. Des weiteren braucht der Benutzer sich um dieses nicht zu kümmern, da es durch die Klasse graceCPU automatisch durchgeführt wird. Mehr dazu im Kapitel 4.4.5.

4.4.2 Graphausdrücke

Graphausdrücke beschreiben Teilmengen von Graphen. In GRACE-Programmen werden sie benutzt, um zu testen, ob ein gegebener Graph Element dieser Menge ist oder nicht. Sie werden damit als Prädikat gce: $\mathcal{G} \times \mathcal{E} \rightarrow$ {true, false} angesehen, wobei gce(G, E)=true gdw. G$\in$ SEM(E) und gce(G, E)=false sonst.

Graphausdrücke werden, wie Kontrollbedingungen, über Transitionsrelationen definiert:

$$\text{SEM(E)}=\{G \in \mathcal{G} \mid < E, G > \xrightarrow{*} < \lambda, G' > \}.$$

Die Berechnung des Prädikates gce wird dann auf die Berechnung der Transitionsequenz $< E, G > \xrightarrow{*} < \lambda, G' >$ zurückgeführt. Dieses Konzept ist sehr flexibel und erlaubt neben speziellen Ausdrücken der Graphklasse auch die Verwendung von Graphtransformationen zur Beschreibung der Eigenschaften von Graphen. In der aktuellen Implementierung wird der Aufruf von Transformationseinheiten zugelassen.

Da Graphausdrücke unabhängig von einer bestimmten Graphklasse behandelt werden, benötigen wir ein Konzept, wie die Ausdrücke abstrakt definiert werden können. Durch die textuelle Speicherung und Repräsentation von GRACE-Programmen, werden Graphausdrücke als Zeichenketten behandelt. Die Bedeutung der Zeichenkette wird dann von der jeweiligen Implementierung bestimmt.

4.4.3 Transformationseinheiten

Transformationseinheiten bestehen aus einem Graphausdruck zur Beschreibung von initialen und terminalen Graphen. Die Kontrollbedingung wird durch eine Sequenz von Befehlen implementiert.

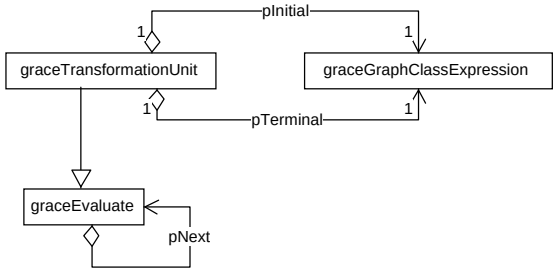


Abbildung 4-13: Transformationseinheit

In der bisherigen Implementierung besitzen Transformationseinheiten keine lokalen Regeln, da es einige Probleme mit der Verwaltung beim Laden und Speichern gab. Diese Probleme sind

darauf zurückzuführen, daß mehrere Graphersetzungsansätze unterstützt werden. In der nächsten Version wird dieses behoben sein.

4.4.4 Module

Module bestehen aus zwei Listen von Transformationseinheiten. Die eine enthält die exportierten und die andere die internen Einheiten. Darüber hinaus enthält ein Modul einen Parser für GRACE-Programme, um eine Laufzeitumgebung aufzubauen. Wenn ein Modul andere importiert, so wird der Übersetzungsprozeß an eine neue Instanz der Klasse graceModule übergeben und zunächst dieses geladen. Damit keine Module doppelt geladen werden wird dieses über eine zentrale Einheit gesteuert.

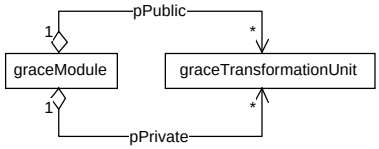


Abbildung 4-14: Module

Module besitzen aus den gleichen Gründen wie Transformationseinheiten keine lokalen Regeln.

4.4.5 GRACE-Programme

GRACE-Programme bestehen aus einer Sammlung von Modulen, so daß jeder Import einen korrespondierenden Export besitzt. Ein Modul besitzt folgende Syntax:

```
MODULE         ::= "module" IDENTIFIER TYPE [USES] EXPORTS
                   IMPLEMENTATION "end.".
TYPE           ::= "graph class:" IDENTIFIER.
USES           ::= "uses" IDENTIFIER {"," IDENTIFIER } ";".
EXPORTS        ::= "exports" EUNIT {";" EUNIT }.
EUNIT          ::= "transformation unit" IDENTIFIER.
IMPLEMENTATION ::= "realized by" UNIT {";" UNIT }.
UNIT           ::= "transformation unit" IDENTIFIER
                   INITIAL BODY TERMINAL "end".
BODY           ::= "body:" CONTROL.
INITIAL        ::= "initial:" [GRAPHEXPRESSION].
TERMINAL       ::= "terminal:" [GRAPHEXPRESSION].
CONTROL        ::= STATEMENT {";" STATEMENT }.
STATEMENT      ::= CALL | APPLY | IF | WHILE.
CALL           ::= "call" IDENTIFIER.
APPLY          ::= "apply" MODE IDENTIFIER [{("|"|"<") IDENTIFIER}].
MODE           ::= "once" | "as long as possible" |
                   (NUMBER "times").
IF             ::= "if" GCEXPRESSION "then" CONTROL
                   ["else" CONTROL] "end".
WHILE          ::= "while" GCEXPRESSION "do" CONTROL "end".
```

Diese Syntax wird vom Übersetzer in die Laufzeitumgebung umgesetzt (siehe Abbildung 4-12, Abbildung 4-13 und Abbildung 4-14). Dabei ergibt sich nun die Schwierigkeit, daß ein als Zeichenfolge kodierter Graphausdruck in eine Instanz einer entsprechenden Klasse umgesetzt werden muß. Ferner müssen Regeln nachgeladen werden. Beide Aktionen sind abhängig von der verwendeten Graphklasse. Wenn der Übersetzer an eine solche Stelle kommt, muß er dafür sorgen, daß die richtige konkrete Klasse anstelle der abstrakten benutzt wird.

Da der Übersetzer aber nur auf abstrakten Klassen arbeitet, weiß er nicht, welche Klasse er benutzen muß. Dieses kann nur derjenige entscheiden, der die konkreten Klassen implementiert hat. Wir müssen also wieder einmal eine abstrakte Schnittstelle definieren, mit deren Hilfe der Übersetzer diese Aufgabe lösen kann.

Wir führen dazu das Konzept des Graphklassenhändlers ein. Dieses sind dynamische Bibliotheken, die zur Laufzeit vom Betriebssystem nachgeladen werden können. Sie besitzen eine feste Schnittstelle mit folgenden Funktionen:

- gcGraphClassExpression
- gcLoadGraph
- gcSaveGraph
- gcLoadGraphRule

Die Methode gcGraphClassExpression bekommt als Parameter die Zeichenkette mit dem Graphausdruck und erzeugt eine Instanz der entsprechenden Klasse. Als Ergebnis wird ein Zeiger auf einen Graphausdruck zurückgegeben.

Zum Laden bzw. Speichern von Graphen sind die Funktionen gcLoadGraph und gcSaveGraph vorgesehen. GcLoadGraphRule lädt eine Graphregel.

Wenn der Übersetzer an eine Stelle kommt, wo er Graphen, Graphausdrücke oder Graphregeln laden muß, so geschieht dieses über eine Indirektion über die Instanz graceDatabase der Klasse graceDB. Die Klasse graceDB verwaltet Module, Regeln und Graphklassenhändler. Wird ein Objekt angefordert, daß noch nicht geladen wurde, wird dieses automatisch nachgeladen. Für jede benutzte Graphklasse muß eine Bibliothek vorliegen, die die Schnittstelle implementiert. Welche Bibliothek geladen wird, hängt von der angegebenen Graphklasse in der Moduldefinition ab. Es gilt dabei folgende Regel: Name der Bibliothek = Name der Graphklasse.

Die Struktur der Klasse graceDB ist:

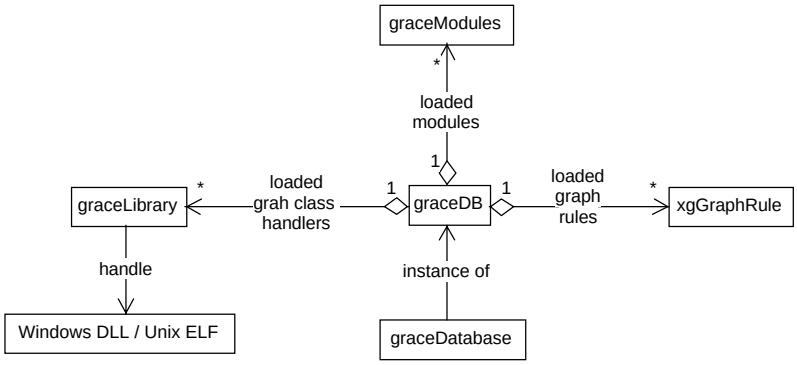


Abbildung 4-15: Struktur der Klasse graceDB

4.4.6 Zentrale Prozeßeinheit

Ein GRACE-Programm wird durch die Angabe des initialen Graphen und dem Namen einer Transformationseinheit gestartet. Regeln werden nicht als Programm angesehen, da sie vom Framework direkt ausgeführt werden können. Nachdem das Modul geladen wurde, wird ein Thread gestartet und die Berechnung in den Hintergrund verlagert. Ist ein Programm gestartet, so läuft die Transformation so lange bis sie entweder terminiert (vgl. [Plu98]) oder der Benutzer die Berechnung abgebrochen hat.

Diese Vorgänge werden in der Klasse graceCPU in einer einfachen Schnittstelle gekapselt.

4.5 Zusammenfassung

Mit diesem Design wurde den Anforderungen von GRACE genüge getan und ein einheitliches, konsistentes und dennoch erweiterbares Konzept geschaffen. Einen großen Teil hat die Entscheidung für einen Interpreter dazu beigetragen. Die dezentrale Verwaltung und Ausführung der Instruktionen kann leicht erweitert und an neue Anforderungen angepaßt werden.

Normalerweise sind Interpreter langsamer als Compiler, doch hier ist dieses nicht der Fall. Die Geschwindigkeit des Interpreters hängt von zwei Faktoren ab:

- Bestimmung von gce(G, E), wobei G ein Graph und E ein Graphausdruck ist und
- Bestimmung eines Graphenmorphismus.

Alle anderen Faktoren arbeiten mit linearen Zeitaufwand, da sie lediglich einige Zeiger verfolgen oder setzen. Ein auf den ersten Blick nicht sichtbarer Zeitfaktor ist die Visualisierung, doch dazu später mehr.

Relativ viel Zeit benötigt das Erzeugen des Ableitungsbaumes, wenn auch ein single pass Übersetzer benutzt wird, und das Laden der dynamischen Bibliotheken. Es handelt sich hier um einen einmaligen Vorgang, so daß die verbrauchte Zeit nicht weiter ins Gewicht fällt.

5 GRACELAND

5.1 Einleitung

In diesem Kapitel wird eine Entwicklungsumgebung für GRACE vorgestellt. Die Entwicklungsumgebung faßt verschiedene Editoren und eine Oberfläche für den Interpreter zusammen. Ziel ist es, dem Benutzer eine größtenteils graphische Schnittstelle zur Verfügung zu stellen, um die gesamte Mächtigkeit von Graphen und Graphersetzungssystemen auszunutzen.

> *"There is now a reasonable amount of evidence to suggest that the judicious use of graphics and direct manipulation interaction are important techniques in producing powerful and expressive application interfaces." ([DOL94], Seite 27)*

Nach [DOL94] spielt, neben der Visualisierung, die Interaktion eine wichtige Rolle. Es wird die direkte Interaktion angesprochen, in der die graphischen Entitäten gleichzeitig als Ein- und Ausgabemedium benutzt werden. Eine textuelle Eingabe, welche die Ausgabe manipuliert, entspricht nicht einer direkten Interaktion, da zur Eingabe ein anderes Medium benutzt wird. In einigen Fällen ist es aber entscheidend auf ein anderes Medium zurückzugreifen. Soll z.B. ein Objekt exakt plaziert werden, so kann dieses u.U. einfacher mit der Angabe der Koordinaten geschehen.

Interessant ist, daß keine vollständig graphische Schnittstelle gefordert wird. Vielmehr wird vom klugen Einsatz der Techniken gesprochen. Dieser Aspekt wird uns im Zusammenhang mit der Visualisierung und Eingabe von GRACE-Programmen noch näher beschäftigen. Dort wird eine Mischung von verschiedenen Medien favorisiert, um für jede Aufgabe das "optimale" zu benutzen.

In den folgenden Kapiteln wird zunächst der Frage nachgegangen, wie die verschiedenen Elemente visualisiert werden können. Danach werden die Interaktionsformen beschrieben.

5.2 Visualisierung

Zweidimensionale Darstellungen von Graphen und Graphregeln sind heute Standard, die u.a. in Programmen wie VCG ([San94]), AGG und PROGRES eingesetzt werden. Die Qualitäten sind als durchweg gut anzusehen. Doch erst wenige Tools nutzen den dreidimensionalen Raum zur Visualisierung (GV3D, [Fra97]), da für viele Applikationen eine zweidimensionale Darstellung ausreicht.

In dieser Arbeit wird eine dreidimensionale Darstellung angestrebt, da sie mächtiger als eine zweidimensionale Visualisierung ist.

“Viewing a graph in a Virtual Reality display is three times as good as a 2D diagram.” ([WF94])

Dies ist der Titel eines Artikels von Ware und Franck, in dem sie für eine dreidimensionale Darstellung von Graphen plädieren. Sie verglichen verschiedene Darstellungsarten miteinander und stellten dabei fest, daß die Fehlerrate bei den gestellten Aufgaben im VR Modus am geringsten war.

Ein weiterer Grund für eine dreidimensionale Darstellung ist, daß viele Graphen eine inhärente 3D Struktur besitzen. So läßt sich jeder nicht planare Graph im Dreidimensionalen angemessener darstellen, was zu einer erheblichen Steigerung der Darstellungsqualität führt. Viele, wenn nicht sogar alle Kreuzungen von Kanten können vermieden werden.

Eine wichtige Eigenschaft einer dreidimensionalen Visualisierung ist es, daß Graphen entsprechend ihren Vorbildern eins zu eins abgebildet werden können. So können z.B. Topologien von Fördersystemen exakt modelliert werden. Knoten können entsprechend der Positionen der Originale im Realen angeordnet und adäquat durch Modelle dargestellt werden. Dadurch wird der Graph für den Benutzer übersichtlicher und verständlicher.

Auch soll nicht unerwähnt bleiben, daß 2D eine Teilmenge von 3D ist, weshalb man durch 3D nur gewinnen kann. Jede zweidimensionale Ausgabe läßt sich in einer dreidimensionalen darstellen, indem die zusätzliche Dimension auf einen konstanten Wert gesetzt und dadurch ignoriert wird.

Wie viele Vorteile die dritte Dimension in der Praxis bringt, muß sich erst noch zeigen.

5.2.1 Graphen

Knoten

Bei Knoten kann man zwei Symbolarten unterscheiden: Standardsymbole und nicht Standardsymbole. Unter Standardsymbole werden Kugeln, Würfel, Zylinder und Kegel zusammengefaßt, die "primitive" Objekte im dreidimensionalen Raum darstellen. Sie können in allen Graphen benutzt werden, ohne ihnen eine implizite Bedeutung zuzuordnen[11]. Wenn aber spezielle Symbole verwendet werden, so kann dieses unter Umständen eine implizite Interpretation hervorrufen.

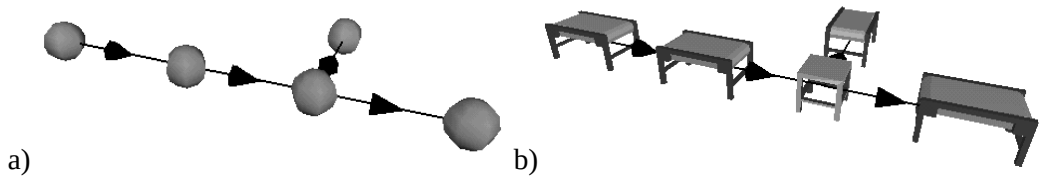


Abbildung 5-1: Interpretation von Symbolen

[11] Eine Ausnahme sind Petri-Netze, die Kugeln und Quader als spezielle Symbole verwenden.

Abbildung 5-1 zeigt ein Beispiel. Die beiden Graphen sind exakt identisch in ihrer Anordnung. In der Variante b) wurden die Knoten durch spezielle Symbole aus dem Bereich der Fördertechnik ersetzt. Bei b) scheint es sich um eine Anordnung von Förderbänden und einem Knotenpunkt zu handeln. Der zugrunde liegende Graph ist exakt der gleiche wie in Variante a). Der Effekt beruht auf der visuellen Wahrnehmung des Menschen. Wenn er ein Symbol erkennt, so setzt er es in dem ihm bekannten Kontext und interpretiert es somit. Diese "Illusion" kann man nutzten, um die Graphen mit mehr Informationen anzureichern. Soll z.B. ein Fördersystem modelliert werden, so bietet es sich an, die Knoten durch spezielle Symbole zu ersetzen, um die Wiedererkennung zu verbessern. Würde man in diesem Fall Variante a) benutzen, so müßte man mehr erklären, was ein Knoten für eine Bedeutung hat, auch wenn die Anordnung genauso wie in der Realität wäre.

Welche Repräsentation für Knoten genommen wird, wird dem Benutzer überlassen. Es werden neben den Standardsymbolen auch benutzerdefinierte zugelassen.

Die Repräsentation eines Knotens wird durch folgende Attribute beeinflußt:

- string representation = "...",
- vector scale = (x y z),
- vector color = (r g b),
- vector orientation = (x y z) und
- vector position = (x y z).

Das Attribut representation definiert die zu verwendene Geometrie. Dabei sind folgende Bezeichner reserviert: sphere für eine Kugel, cone für einen Kegel, cylinder für einen Zylinder und cube für einen Würfel. Fehlt das Attribut oder kann die Geometriedatei nicht geladen werden, wird automatisch eine Kugel als Darstellung gewählt. Benutzerdefinierte Geometrien werden im 3rd Format beschrieben (siehe Anhang).

Die Größe des Objektes wird durch scale gesteuert, indem ein Faktor für jede Achse angegeben wird. Fehlt das Attribut, so wird ein Skalierungsfaktor von 1 angenommen.

Standardsymbole können farblich verändert werden, indem die gewünschte Farbe im Attribut color angegeben wird. Dabei liegt das RGB-Modell zugrunde, das Farben durch die Angabe des Rot-, Grün- und Blauanteils kodiert (vgl. [FDF+96]). Ist keine Farbe angegeben, so wird die Farbe r=0 g=0.5 b=1, ein ins Cyan gehender Blauton, benutzt. Die r, g, b Werte müssen zwischen 0 und 1 liegen, wobei 0 für keinen Anteil steht und 1 für den maximalen.

Über orientation ist es möglich, ein Objekt um seinen Mittelpunkt zu drehen. Ist dieses Attribut nicht vorhanden, wird eine Rotation um 0° angenommen. Die Winkel werden in Grad angegeben.

Darüber hinaus besitzt ein Knoten noch eine Position, die durch das Attribut position vom Typ vector gegeben ist. Wenn das Attribut nicht vorhanden ist, wird der Ursprung als Position angenommen.

Kanten

Kanten können ebenfalls auf viele verschiedene Arten gezeichnet werden. Basis für eine Kante ist eine Verbindung, die den Quell- mit dem Zielknoten verbindet. Die Verbindung kann eine gerade Linie, ein Linienzug oder eine Kurve sein. Eine gerade Linie bietet sich dann an, wenn keine Knoten geschnitten werden. Andernfalls sollte die Linie außen um die Knoten herumgeführt werden. Wie genau die Linie aussieht, hängt von der jeweiligen Situation ab. Diese Entscheidung wird nicht von GRACEland durchführt, sondern Layout-Algorithmen überlassen. In der vorliegenden Version von GRACEland werden nur gerade Linien und speziell gebogene Linienzüge unterstützt.

Da wir es mit gerichteten Graphen zu tun haben, besitzen Kanten zusätzlich Pfeile, die die Richtung angeben. Pfeile lassen sich an verschiedenen Positionen zeichnen: am Anfang, Ende oder in der Mitte. Wenn ein Pfeil am Anfang bzw. am Ende der Linie gezeichnet wird, so muß der Pfeil soweit zurückgesetzt werden, bis er nicht mehr vom Objekt verdeckt ist, was einen relativ hohen Rechenaufwand zur Folge hätte. Werden hingegen die Pfeile in der Mitte der Linie gezeichnet, so kann auf komplizierte Berechnungen verzichtet werden. Auch in diesem Fall kann ein Pfeil von einem Objekt verdeckt werden, was nur noch durch ein Ausweichen auf Linienzüge vermieden werden kann.

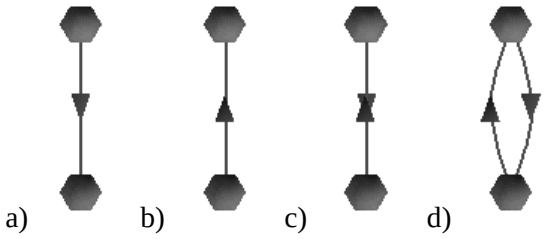


Abbildung 5-2: Knoten und Kanten

Abbildung 5-2 zeigt Kanten zwischen Knoten. In a) und b) verbindet jeweils eine, in c) und d) zwei Kanten die beiden Knoten. Die Darstellung in c) ist nicht glücklich, da sich die beiden Linienzüge überlagern und kaum unterschieden werden können. Wenn noch mehr Kanten hinzukommen, wird das Ganze noch unübersichtlicher und man weiß nicht, wie viele Linienzüge die beiden Knoten miteinander verbinden. Eine korrekte Lösung für zwei Kanten zeigt d). Dort werden die Kanten aneinander vorbeigeführt, so daß sie nicht überlappen. Abbildung 5-3 zeigt die Konstruktion, um Kanten überschneidungsfrei zu halten:

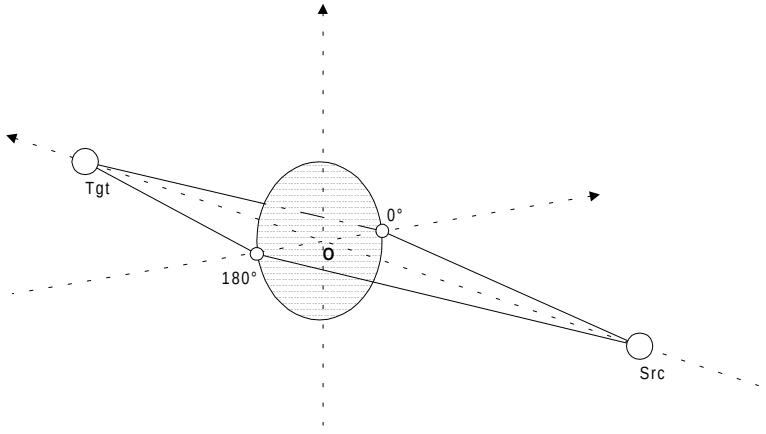


Abbildung 5-3: Überschneidungsfreie Kanten

Alle Kanten berühren den um O gespannten Kreis, der in N verschiedene Segmente unterteilt wird. N ist dabei die Anzahl der Kanten zwischen den Knoten Src und Tgt. Jede Kante geht durch den Anfang ihres Segments, d.h. $\alpha = Nr * \frac{360°}{N}$. Jede Kante erhält eine Nummer Nr, die zwischen 0 und N-1 liegt. Die Einteilung der Nummern geschieht zufällig und hängt von der Reihenfolge in der Kantenliste ab. Der Punkt auf dem Kreisbogen kann dann mit Hilfe von Sinus und Cosinus berechnet werden:

$$P_{Kreisbogen} = \begin{pmatrix} 0 \\ r_{Kreisbogen} * \sin(\alpha) \\ r_{Kreisbogen} * \cos(\alpha) \end{pmatrix}.$$

Die Koordinaten wurden so gewählt, daß in der Draufsicht eine Linienführung wie in Abbildung 5-2 d) erscheint. Nun muß der Punkt noch in das Weltkoordinatensystem transformiert werden.

1. Rotation um den Winkel $\beta = \angle(Src, Tgt)$
2. Translation an den Mittelpunkt von Src und Tgt, mit $T = \frac{Src + Tgt}{2}$

Diese Berechnung gilt für Kanten mit Quelle Src und Ziel Tgt. Bei Kanten mit Quelle Tgt und Ziel Src werden geringfügige Änderungen benötigt. Dieses liegt daran, daß der Winkel α abhängig von der Sichtweise auf den Kreis ist. Ein Winkel α von Src aus gesehen entspricht einem Winkel 360°-α von Tgt aus gesehen. Des weiteren ist das Vorzeichen von $P_{Kreisbogen}$ umzudrehen, da der Winkel ß negiert ist ($\beta = \angle(Src, Tgt) = -\angle(Tgt, Src)$) und auf die Berechnung von β keinen Einfluß genommen werden kann.

Zur Darstellung der Linien werden Bézier Kurven benutzt. Ein Bézier Kurvensegment wird durch die Angabe von vier Kontrollpunkten spezifiziert. Die Kurve ist dann durch $Q(t)=(1-t)^3P_1 + 3t(1-t)^2P_2 + 3t^2(1-t)P_3 + t^3P_4$, mit $0 \le t \le 1$ definiert. Eine ausführliche Einführung in die Theorie der parametrisierten Kurven findet man in [FDF+96], [Fel92] oder [Wat93].

Für uns sind Bézier Kurven interessant, weil sie die Punkte P_1 (t=0) und P_4 (t=1) interpolieren, d.h. durch die Punkte verlaufen. Somit können wir P_1 gleich Src und P_2 gleich Tgt setzen. Die verbleibenden Parameter P_2 und P_3 steuern den Verlauf der Kurve zwischen den Eckpunkten. Ziel ist es, daß die Kurve den berechneten Punkt auf dem Kreisbogen interpoliert. Wir setzen dazu P_2 und P_3 auf den in Weltkoordinaten umgerechneten Punkt $P_{Kreisbogen}$.

Allerdings verläuft die Kurve nicht genau durch den gewünschten Punkt, sondern kommt nur nah an ihn heran. Dafür verlaufen aber alle Kurven durch einen Kreis, dessen Radius etwas geringer als $r_{Kreisbogen}$ ist. Da wir hier nicht wirklich daran interessiert sind, daß die Kurve durch einen bestimmen Kreis geht, sondern nur die Kantenzüge entwirren wollten, ist damit die Aufgabe gelöst. Alle Kanten werden durch diese Konstruktion so geführt, daß sie sich nicht mehr berühren (bei entsprechend großem $r_{Kreisbogen}$). Gezeichnet werden die Kurven durch eine lineare Approximation von Teilstrecken, wodurch eine hohe Geschwindigkeit und eine unnötige Genauigkeit vermieden wird.

Das Aussehen der Kanten wird durch die Attribute color und scale definiert. Color ändert die Farbe der Linie, welche standardmäßig auf Rot gesetzt ist. Mit Hilfe von scale kann die Größe des Pfeiles geändert werden.

An dieser Stelle zeigt sich die Flexibilität des Attributkonzeptes aus Kapitel 4.1, welches erlaubt, zusätzliche Informationen in den Knoten und Kanten zu speichern, ohne die Graphtransformationskomponente zu beeinflussen.

Markierungen (Labels) von Knoten und Kanten werden als Text den Knoten bzw. Kanten zugeordnet. Alle anderen Komponenten werden nicht visualisiert, wodurch ein Überladen der Darstellung vermieden werden soll.

5.2.2 Graphregeln

Die Darstellung von Graphregeln wird auf die Visualisierung des Graphen der linken Seite und der rechten Seite zurückgeführt. Der Interface-Graph wird, wie in der Implementierung, durch die Zuordnung von Entitäten von der linken auf die rechte Seite dargestellt. Entitäten, die sich entsprechen, bekommen eine Nummer zugeordnet.

Die Beschriftung eines Knotens setzt sich aus seiner Interface-Nummer, seinem Label und der Variable zusammen: [Nummer ":"] [Label]["(" Variable ")"].

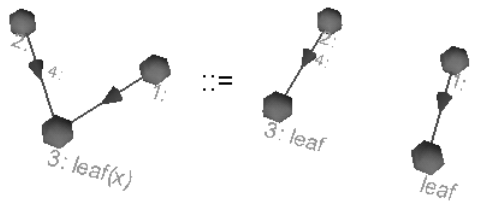


Abbildung 5-4: Darstellung einer Graphregel

5.2.3 Programme

> *"[...] software is very difficult to visualize. Whether we diagram control flow, variable scope nesting, variable cross-references, data flow, hierarchical data structures, or whatever, we feel only one dimension of the intricately interlocked software elephant. If we superimpose all the diagrams generated by the many relevant views, it is difficult to extract any global overview. The VLSI analogy is fundamentally misleading – a chip design is a layered two-dimensional object whose geometry reflects its essence. A software system not." ([Bro95], Seite195)*

Das Problem bei der Visualisierung von Softwaresystemen ist nach Brooks, daß die Geometrie nicht das Wesentliche des Softwaresystems reflektiert. GRACE-Programme sind eine Ausnahme, da Graphregeln graphisch dargestellt werden. Allerdings liegt der Kontrollfluß für die Graphersetzung als Text vor, so daß es sich um eine Mischform handelt.

Die einfachste Art der Visualisierung ist, den Programmtext als Text darzustellen. Um die lexikalischen Einheiten besser unterscheiden zu können, wird zusätzlich noch die Schriftart

oder
-farbe verändert. Für einen mit der Sprache vertrauten Benutzer handelt es sich um eine effektive Art und Weise Programme einzugeben bzw. zu editieren. Ungeübte Benutzer können sich aufgrund der Einfachheit der Sprache schnell einarbeiten.

Zur weiteren Unterstützung des Benutzers, sollte der Texteditor Regeln als Bilder in den Text integrieren oder zumindest eine Vorschau ermöglichen. Diese Art von Editor wurde teilweise schon in PROGRES eingesetzt und hat sich in der Praxis bewährt. Dort wird ein syntaxgesteuerter Editor benutzt, der gleichzeitig die Eingabe bzw. Veränderung von Regeln ermöglicht. Erschwert wird die Eingabe durch die Syntaxsteuerung, wodurch der Schreibfluß immer wieder gestört wird. Für PROGRES ist diese Form des Editors geeignet, da die Sprache sehr umfangreich und kompliziert ist und so viele syntaktische Fehler vermieden werden. Bei GRACE-Programmen hingegen ist die Sprache klein und übersichtlich, so daß eine syntaxgesteuerte Eingabe nur hinderlich wäre.

Abbildung 5-5 zeigt einen Auszug aus einem GRACE-Programm (vgl. Kapitel 5.4.3):

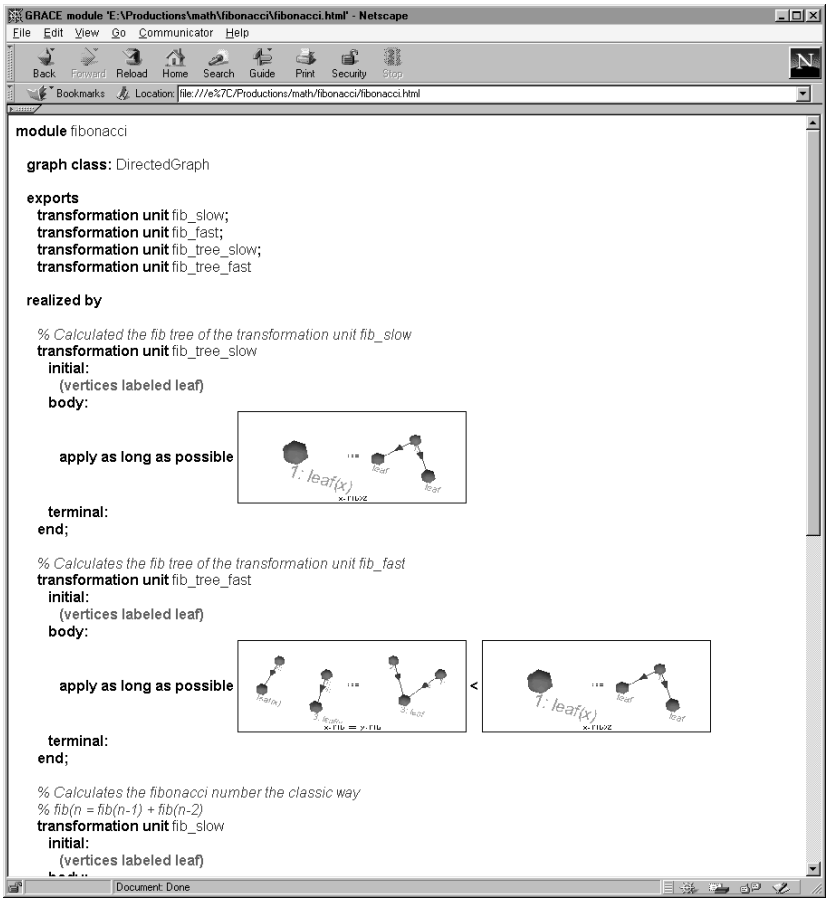


Abbildung 5-5: Textuelle und graphische Darstellung von GRACE-Programmen

Als Alternative bietet sich eine graphische Notation an. Der Benutzer arrangiert Symbole oder Grafiken, die verschiedene Elemente der Sprache repräsentieren, nach und nach zu einem Programm. Basis dieser Notation könnte z.B. ein hierarchischer Ansatz sein, in der ein

Programm Schicht für Schicht entwickelt wird. Auf oberster Ebene würden Modulbeziehungen definiert werden, darunter Transformationseinheiten und später Regeln, Kontrollfluß, etc. . Dazu könnte man einen speziellen Grapheditor benutzen, um so eine konsistente Schnittstelle mit dem Graph- und Regeleditoren erzeugen.

In der aktuellen Version von GRACEland wurde eine komplett graphische Repräsentation der Programme nicht realisiert, da keine Vorteile absehbar waren. Vielmehr ist zu erwarten, daß die Eingabe der Programme ausgebremst wird, da zum einen die Visualisierung viel Rechenzeit benötigt und zum anderen die Interaktion für den Benutzer zu umständlich ist.

5.3 Virtual Reality Benutzungsschnittstelle

Man kann grundsätzlich zwei Arten von VR-Schnittstellen unterscheiden: Desktop VR und Immersive VR. Desktop VR integriert die virtuelle Welt in die graphische Oberfläche des Betriebssystems, d.h. die Welt wird in einem Fenster auf dem Desktop dargestellt. Immersive VR benutzt spezielle Ausgabegeräte (z.B. Head Mounted Devices, Caves) um den Benutzer in die Welt eintauchen zu lassen. Durch das Ausgabegerät wird der Benutzer von der Realität abgekapselt und nimmt nur noch die virtuelle Welt wahr. Eine ausführliche Einführung in das Thema Virtual Reality, sowie Informationen zu Hard- und Software findet der interessierte Leser unter anderem in [AB92], [MG96].

GRACEland benutzt die Desktop VR Variante, um neben der virtuellen Welt auch klassische GUI (Graphical User Interface) Elemente nutzen zu können. Um die virtuelle Welt und die Interaktion mit dieser zu verwalten, wird das VR Toolkit VLE ([Fau97]) eingesetzt. Eine virtuelle Welt besteht, grob gesprochen, aus einer Szene, welche die in der Welt vorhandenen Objekte verwaltet. Der Benutzer sieht einen Ausschnitt der Welt entsprechend seinem Blickpunkt und seiner Blickrichtung.

Abbildung 5-6 zeigt ein Screenshot eines Desktop VR Systems.

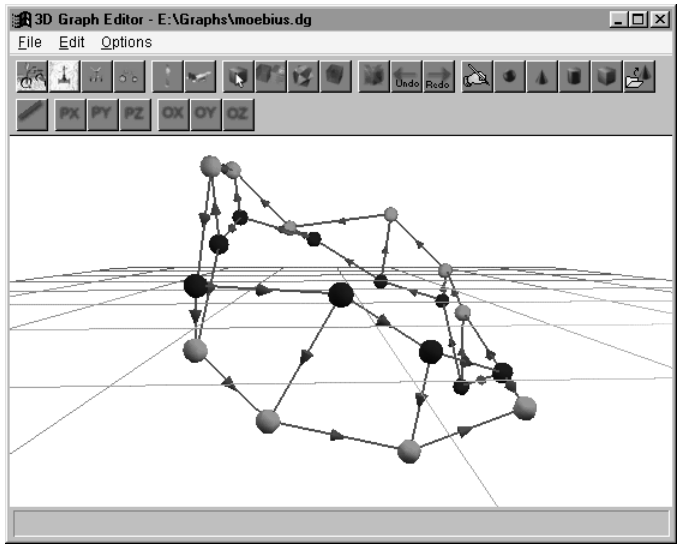


Abbildung 5-6: Desktop VR

Der Benutzer kann nun mit einem Eingabegerät entweder mit der virtuellen Welt interagieren oder z.B. ein Menü aktivieren. Die virtuelle Welt erfordert allerdings eine andere Interaktion, als der klassische zweidimensionale Desktop, da sie insgesamt 6 Freiheitsgrade besitzt. Drei für die Positionierung und drei für die Orientierung. Normalerweise sollten spezielle Eingabegeräte, wie z.B. Datenhandschuh oder 6D Maus, für die Interaktion verwendet werden, doch sie sind nur wenig verbreitet. Da keine exklusive Lösung für spezielle Hardware angestrebt wird, wird die Maus als Eingabegerät benutzt.

Die Interaktion eines zweidimensionalen Eingabegerätes in einem sechsdimensionalen Raum erfordert die Interpretation von Signalen. Es kann keine 1 zu 1 Abbildung stattfinden, da zu wenig Freiheitsgrade vorhanden sind. Die Maus besitzt zwar nur zwei Freiheitsgrade, hat aber noch zusätzlich Tasten. Durch die Kombination von Tasten besitzen wir genügend Dimensionen, so daß in der virtuellen Welt agiert werden kann.

Die Interaktion mit der virtuellen Welt gliedert sich grob in zwei Kategorien: Navigation und Manipulation von Objekten. Navigation umfaßt die Veränderung des Standpunktes und der Blickrichtung und Manipulation von Objekten, deren Translation, Rotation und Skalierung.

Wenn im folgenden die Rede von X-, Y- und Z-Achsen ist, so wird folgendes Koordinatensystem zugrunde gelegt:

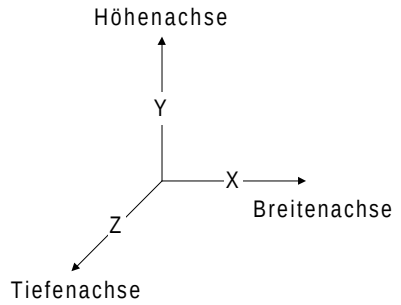


Abbildung 5-7: Rechtshändiges Koordinatensystem

5.3.1 Navigation

Es wurden verschiedene Formen entwickelt, auch als Metaphern bezeichnet, um seinen Standpunkt und seine Blickrichtung zu verändern. In GRACEland sind die zwei am verbreitetsten Metaphern Fly und Walk implementiert. Siehe dazu auch [Han97].

Fly

Im Fly Mode "fliegt" der Benutzer durch die virtuelle Welt in Richtung der aktuellen Blickrichtung. Darüber hinaus kann ein Gleiten nach links und rechts sowie oben und unten stattfinden. Abbildung 5-8 zeigt das zugrunde liegende Flugzeugmodell.

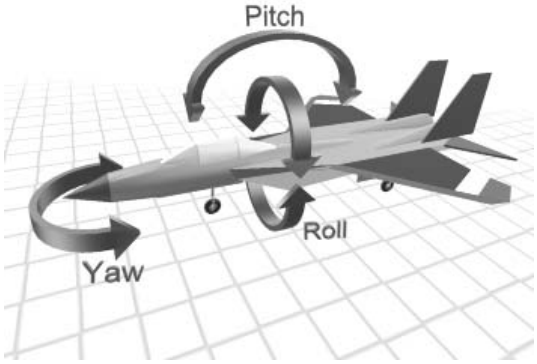


Abbildung 5-8: Fly-Metapher

Die Richtung wird durch die Winkel Yaw (dt. Gieren), Pitch (dt. Nicken) und Roll (dt. Rollen) bestimmt. Die Nase des Flugzeugs gibt die Richtung an, in die geflogen wird. Die Fly-Metapher benutzt kein reales Flugmodell als Grundlage, sondern übernimmt nur die Terminologie. Zusätzlich kann ein Gleiten in Richtung der Flügelspitzen bzw. in Richtung der Leitwerke geschehen.

Die Geschwindigkeit der Änderungen der Position bzw. Orientierung wird vom Benutzer selbst gesteuert. Ausschlaggebend ist die Entfernung der aktuellen Mausposition zu einem Bezugspunkt. Je weiter die Maus vom Bezugspunkt entfernt ist, desto höher ist die Geschwindigkeit.

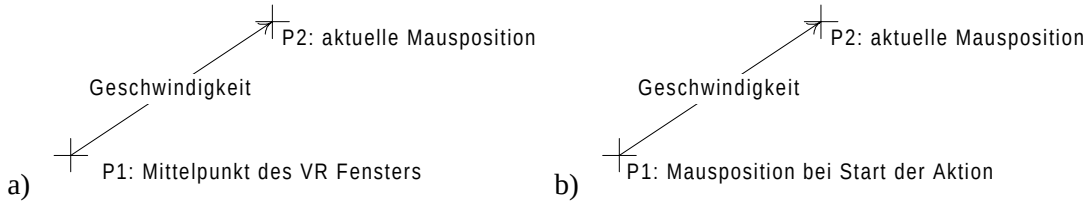


Abbildung 5-9: Bestimmung der Geschwindigkeit

Abbildung 5-9 zeigt zwei mögliche Bestimmungen von Beschleunigung. In einigen Programmen wird die Beschleunigung in Bezug zum Mittelpunkt des VR Fensters gesetzt und in anderen zur Mausposition bei Start der Aktion. In GRACEland wurde die zweite Version implementiert, da sie den Vorteil hat, benutzerfreundlicher zu sein, weil keine unvorhergesehenen abrupten Änderungen vorkommen. Extreme Änderungen treten in der ersten Version aus, wenn eine Aktion weit vom Mittelpunkt entfernt durchgeführt wird, in dessen Folge der Blickpunkt oftmals in der Tiefe der virtuellen Welt verschwindet.

Die Geschwindigkeit ist der Vektor $\overrightarrow{P1P2}$, dessen Komponenten zwei voneinander unabhängige Beschleunigungen definieren. Der Punkt P1 wird bei Drücken einer Maustaste bestimmt und gilt so lange, bis alle Maustasten losgelassen wurden.

Die folgende Tabelle ordnet den Bewegungen der Maus Aktionen der Metapher zu:

Maustaste(n)	Horizontale Bewegung	Vertikale Bewegung
Linke	Verändern des Yaw-Winkels	Fliegen in Blickrichtung
Rechte	Gleiten in Richtung der Flügelspitzen	Gleiten in Richtung der Leitwerke
Beide	Veränderung des Rollwinkels	Verändern der Neigung des "Flugzeuges"

Walk

Bei der Walk-Metapher geht man durch die virtuelle Welt. Es wird ein Mensch als Basis genommen, der in eine Richtung geht. Darüber hinaus kann die Blickrichtung nach unten bzw. oben verändert werden (Look up/down), was keinen Einfluß auf die Bewegungsrichtung hat. Zusätzlich kann ein Gleiten in Schulterrichtung und der Fuß-Hals Richtung geschehen. Dieses hat nichts mehr mit einer Person zu tun, ist aber sinnvoll, um in einer dreidimensionalen Welt zu navigieren.

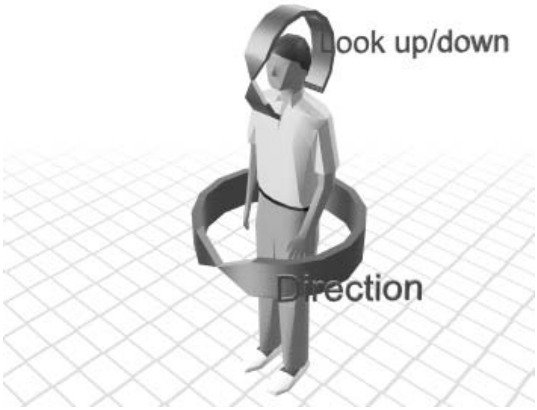


Abbildung 5-10: Walk-Metapher

Die folgende Tabelle ordnet den Bewegungen der Maus Aktionen der Metapher zu:

Maustaste(n)	Horizontale Bewegung	Vertikale Bewegung
Linke	Verändern der Blickrichtung (Direction)	Gehen in Blickrichtung
Rechte	Gleiten in Richtung der Schultern	Gleiten in Fuß-Hals Richtung
Beide	-	Hoch oder Runter blicken

5.3.2 Manipulation von Objekten

Bei der Interaktion mit Objekten werden dreidimensionale Widgets verwendet. Widgets können als eine Art von Vermittler angesehen werden. Wenn Änderungen durchgeführt werden, werden sie zunächst durch die Widgets angezeigt, während das Original unverändert bleibt. Nachdem eine Änderung durchgeführt wurde, werden diese an das Objekt übertragen. Durch das Aussehen definieren Widgets ihre Funktionalität. Die meisten Widgets besitzen sog. Handels, die benutzt werden, um das Objekt zu verändern. Wenn ein bestimmtes Handle benutzt wird, so werden dadurch automatisch Constraints gesetzt. Constraints sperren einzelne Achsen, so daß bestimmte Komponenten bei einer Manipulation nicht verändert werden. Möchte man z.B. ein Objekt nur auf der X-Achse verschieben, so muß man Constraints auf Y- und Z-Achse setzen.

Die hier benutzen Widgets besitzen keine Handels, sondern werden unabhängig durch die Maus gesteuert. Dieses hat Vorteile, da Fenstergrenzen immer behindern, und macht die Implementierung einfacher. Constraints müssen manuell durch den Benutzer vorher oder während der Manipulation gesetzt werden.

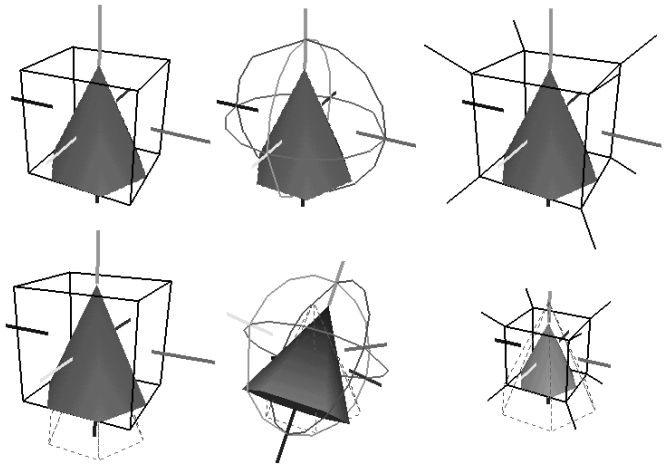


Abbildung 5-11: Widgets Translation, Rotation und Skalierung

Abbildung 5-11 zeigt die drei verschiedenen Widgets. Die obere Reihe zeigt sie in "Ruhestellung", d.h. nachdem ein Objekt selektiert wurde. In der unteren Reihe wurde das Objekt manipuliert. Die alte Konfiguration des Objektes wird gestrichelt dargestellt, so daß der Benutzer die Änderungen verfolgen kann. Allerdings ist dieses bei einer Skalierung nur dann möglich, wenn es sich um eine Verkleinerung handelt, da ansonsten das Original verdeckt ist. Des weiteren wird das Objektkoordinatensystem angezeigt. Violett ist die positive X-Achse markiert, grün die Y-Achse und gelb die Z-Achse. In abgedunkelter Farbe ist die negative Achse gekennzeichnet.

Die Widgets passen sich der Größe des Objektes an, d.h. je flacher ein Objekt ist, desto flacher ist auch das Widget.

Jedes Widget interpretiert die Mausaktionen anders. Dabei wurde, soweit möglich, auf eine Entsprechung zur Mausbewegung geachtet. Die folgende Tabelle zeigt die Interpretation der Mausaktionen.

	Interpretation		
Mausaktion	Widget Translation	Widget Rotation	Widget Skalierung
Linke Maustaste & Horizontale Bewegung	Position X rotiert um den Y-Winkel der Blickrichtung	Rotation Y-Achse	Skalierung in X-Achse
Linke Maustaste & Vertikale Bewegung	Position Z rotiert um den Y-Winkel der Blickrichtung	Rotation X-Achse	Skalierung in Z-Achse
Rechte Maustaste & Horizontale Bewegung	-	Rotation Z-Achse	-
Rechte Maustaste & Vertikale Bewegung	Position Y (Höhe)	-	Skalierung in Y-Achse
Beide Maustasten & Horizontale Bewegung	-	-	Skalierung in X-,Y- und Z-Achse

5.3.3 Schema des verwendeten Treibermodells für den Maussensor

Das verwendete VR Toolkit VLE stellt eine Schnittstelle zur Verfügung, über die jeder Sensor angesteuert werden kann. Es handelt sich dabei um eine abstrakte Basisklasse, deren Funktionalität von abgeleiteten Klassen überschrieben werden muß. Eine ausführlichere Einführung in das Treiberkonzept findet man in [Fau97].

Der Maustreiber VLEwxmouse leitet sich von der Klasse VLEsensor ab und erweitert diesen. Die Methoden OnOpen und OnClose zum Öffnen bzw. Schließen des Sensors haben keine Funktionalität und liefern immer true zurück. In der OnUpdate-Methode werden entsprechend dem aktuellen Modus die dazugehörigen Methoden aufgerufen. In diesen findet dann eine Verarbeitung der Aktionen statt.

Normalerweise wird in der OnUpdate-Methode ein neues Datenpaket vom Sensor geholt und verarbeitet. Dieses läßt die benutzte Oberflächenbibliothek wxWindows jedoch nicht zu. Vielmehr werden Mouseevents an das Objekt geschickt, indem sie auftraten, hier eine Instanz der Klasse VLECanvas. Von dort werden sie über einige Zwischenstationen an die Methode HandleMouseEvent der Klasse VLEwxmouse weitergereicht, die diese Events verarbeitet.

Abbildung 5-12 zeigt schematisch den Zusammenhang und die Interaktion zwischen den verschiedenen Klassen.

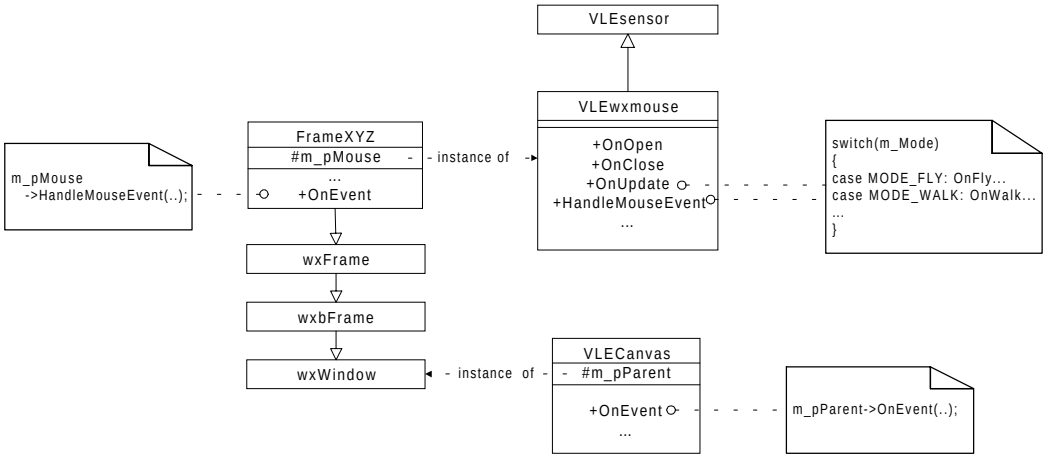


Abbildung 5-12: Schema des verwendeten Maustreibers

5.4 Editoren

5.4.1 Grapheditor

Zum Bearbeiten von Graphen ist der Grapheditor vorhanden, der sich dem Benutzer wie folgt präsentiert:

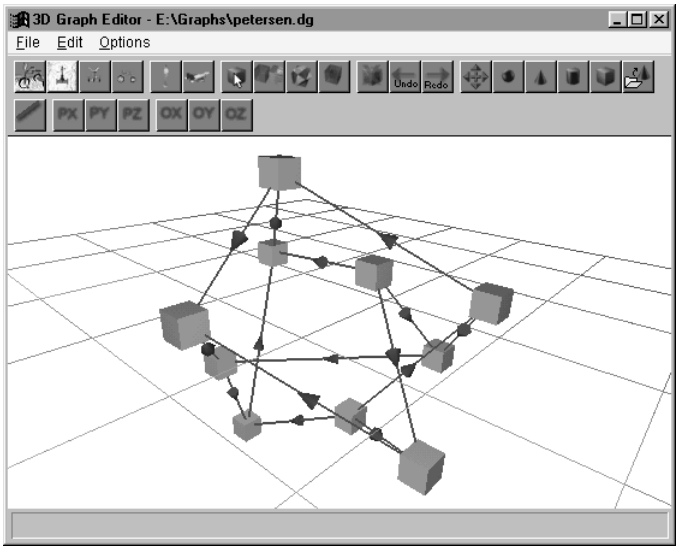


Abbildung 5-13: Screenshot Grapheditor

Abbildung 5-14 zeigt den internen Aufbau des Editors. Grundlage ist ein Graph, dessen Komponenten indirekt über die herkömmliche grafische Oberfläche oder der VR Oberfläche

verändert werden. Die Darstellung des Graphen ist entkoppelt von seiner internen Repräsentation. Über sog. Links wird jeder Entität des Graphen ein grafisches Objekt zugeordnet. Die zwei Modelle des Graphen werden in Übereinstimmung gehalten, d.h. jede Änderung an der virtuellen Welt wird an den Graphen weitergereicht und jede Änderung am Graphen an die virtuelle Welt.

Zugelassene Änderungen an der virtuellen Welt sind Translation, Rotation, Skalierung, Selektion und Verbinden von Knoten. Alle anderen Aktionen werden durch die grafische Benutzungsoberfläche ausgelöst. Das beschriebene Interaktionsmodell läßt sich schematisch so darstellen:

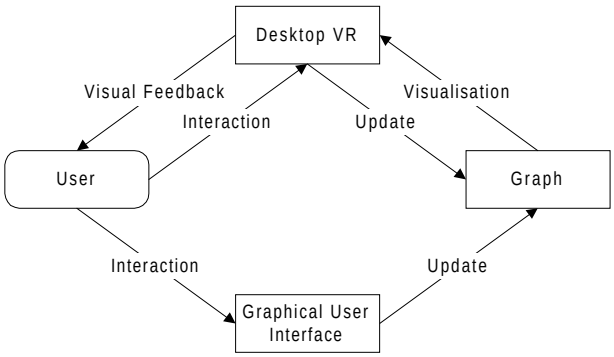


Abbildung 5-14: Interaktionsschema

Implementiert wurde dieses Modell durch eine Model-View-Controller Architektur, wobei anstelle eines zentralen Controllers mehrere dezentrale benutzt werden. Dieses liegt an der propagierten Trennung von Funktionalitäten, damit ein möglichst großer Teil unabhängig von anderen wiederverwendet werden kann. Gerade der Aspekt der Wiederverwendbarkeit spielt eine große Rolle, da die Darstellung von Graphen in verschiedenen Situationen benutzt werden kann.

Modell ist der Graph auf dem gearbeitet wird. View entspricht der virtuellen Welt, die eine Sicht auf den Graphen zeigt. Controller ist u.a. der Maustreiber, der die Signale des Sensors in Aktionen auf die virtuelle Welt umsetzt. Ein anderer Controller ist die Klasse, in der die Events von der graphischen Oberfläche abgefangen und bearbeitet werden.

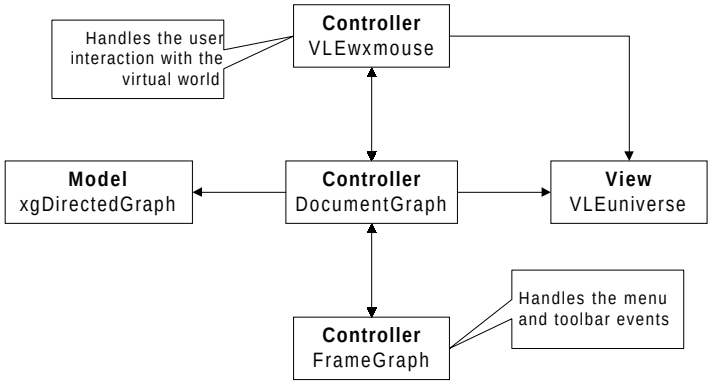


Abbildung 5-15: Model-View-Controller Konzept

Abbildung 5-15 zeigt die am Konzept beteiligten Klassen, wobei DocumentGraph die zentrale Klasse ist. Sie kapselt den direkten Zugriff auf den Graphen in einer Schnittstelle und stellt Funktionen zur Synchronisation mit der virtuellen Welt zur Verfügung. Nahezu alle Aktionen werden in Funktionsaufrufe der Klasse DocumentGraph umgewandelt. Eine Ausnahme stellt das Layouten von Graphen dar, wo direkt auf den Graphen zugegriffen wird (siehe dazu Kapitel 5.6).

Die Befehle des Editors werden hauptsächlich über die Buttonleiste aktiviert. Eine Beschreibung der Icons findet man auf der GRACEland Homepage [Fau98]. Hier soll nur auf einige Extras des Editors eingegangen werden:

- Verschiedene Ansichten eines Graphen,
- Duplizieren von Objekten,
- Undo / Redo und
- Kontextmenü.

Um auf der zweidimensionalen Oberfläche des Monitors eine dreidimensionale Welt darzustellen, bedarf es einer Projektion, wodurch Daten der dreidimensionalen Welt zum Teil verloren gehen. So ist es nicht immer möglich zu entscheiden, ob ein Objekt vor oder hinter einem anderen Objekt ist. Kann man den Graphen aber von verschiedenen Perspektiven betrachten, so helfen die Ansichten dieses Problem zu lösen. Darüber hinaus bietet eine Ansicht z.B. von oben den Vorteil, daß man einen zweidimensionalen Editor simulieren kann, was sinnvoll ist, wenn die dritte Dimension nicht benötigt wird. Ein Graph kann von oben, von links, von vorne oder von einer benutzerdefinierten Perspektive aus betrachtet werden.

Wenn ein Objekt selektiert wurde, so steht dem Benutzer ein Kontextmenü auf der mittleren Maustaste zur Verfügung. Normalerweise wird ein Kontextmenü immer über die rechte Maustaste aktiviert, was hier jedoch nicht möglich ist, da sie bereits für Navigation und Manipulation verwendet wird. Das Menü beinhaltet die Befehle: Change representation (nur für Knoten), Edit attributes, Copy object (nur für Knoten) und Delete.

Um das Konstruieren eines Graphen zu erleichtern, kann ein selektierter Knoten dupliziert werden, wodurch der lange Weg zur Buttonleiste entfällt. Es wird dabei eine Kopie des selektierten Knotens erzeugt, welche die gleichen Attribute besitzt, wie das Original. Die Kopie wird direkt neben dem Original plaziert. Dieses stellt eine große Erleichterung für Knoten mit benutzerdefinierten Geometrien dar, da das Durchsuchen der Dateibäume entfällt.

Veränderungen an Position, Orientierung und Skalierung der Knoten werden mitprotokolliert und können rückgängig gemacht (Undo) und wiederhergestellt (Redo) werden. Wenn nach einem Undo Schritt eine neue Aktion, wie z.B. Verschieben eines Objektes, durchgeführt wird, so wird der Redo Buffer gelöscht.

Wird der Befehl "Change repesentation" aktiviert, so erscheint ein Dialog, in dem der Benutzer die Repräsentation eines Objektes ändern kann.

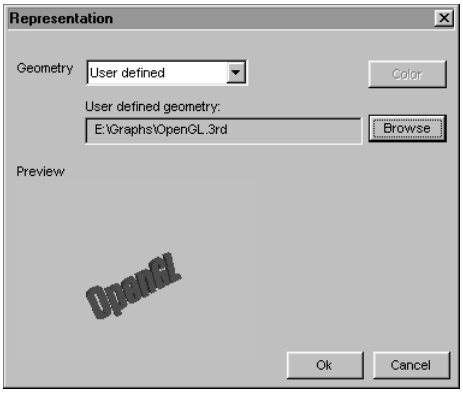
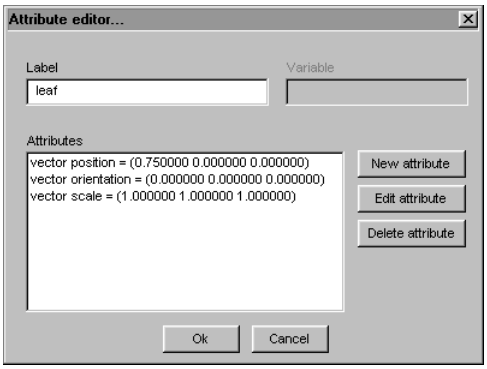


Abbildung 5-16: Dialog Darstellung eines Knotens

Die Geometrie des Objektes wird aus einer Liste ausgewählt, wobei Standardgeometrien und ein Eintrag mit Namen "User defined" angeboten wird. Wählt der Benutzer den "User defined" Eintrag, so wird der Browse-Button aktiviert und das Dateisystem kann nach der gewünschten Geometrie durchsucht werden. In der linken unteren Hälfte des Dialoges wird die gewählte Geometrie rotierend angezeigt.

Der Attributeditor wird über den Befehl "Edit attributes" aufgerufen.

a)



b)

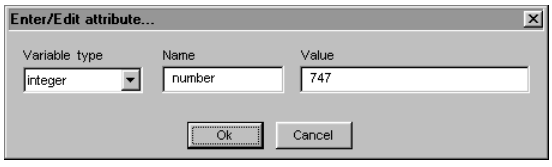


Abbildung 5-17: a) Attributeditor b) Editieren eines Attributes

Abbildung 5-17 a) zeigt den Dialog. Unter "Attributes" sind alle Attribute des Knoten bzw. der Kante aufgelistet. Dabei werden temporäre Attribute, die von der Programmierung benötigt wurden, sowie color und representation nicht aufgeführt

VR Interface

Das VR Interface benötigt viel Rechenzeit und kann so das flüssige Arbeiten gefährden, wenn zu viele Fenster geöffnet sind. Deshalb wurde die Möglichkeit geschaffen, das Interface ab- und anzuschalten. Dadurch müssen die Fenster nicht geschlossen werden, sondern können offenbleiben. Die Darstellung wird nur noch aktualisiert, falls dieses nötig ist.

5.4.2 Regeleditor

Der Regeleditor besteht aus zwei Grapheditoren, die in einem Fenster zusammengefaßt wurden (siehe Abbildung 5-18) Die Grapheditoren sind für die linke bzw. rechte Seite der Regel zuständig. Der Interface-Graph wird durch Zuordnung von Elementen gebildet. Dazu selektiert der Benutzer die entsprechenden Objekte und sie werden einander zugeordnet. Die gesamte Interaktion ist die gleiche wie bei dem Grapheditor. Der einzige Unterschied besteht darin, daß einige Aktionen nur für die linke bzw. rechte Seite gelten. So werden Knoten und Kanten in die aktive Seite eingefügt. Die aktive Seite wird durch einen grauen Button (LHS, RHS) festgelegt. Selektiert der Benutzer den Button Translation, so kann er in der linken, sowie der rechten Seite Verschiebungen vornehmen. Intern wird die jeweilige Seite aktiviert, was der Benutzer durch das Verändern der Buttons LHS und RHS mitverfolgen kann.

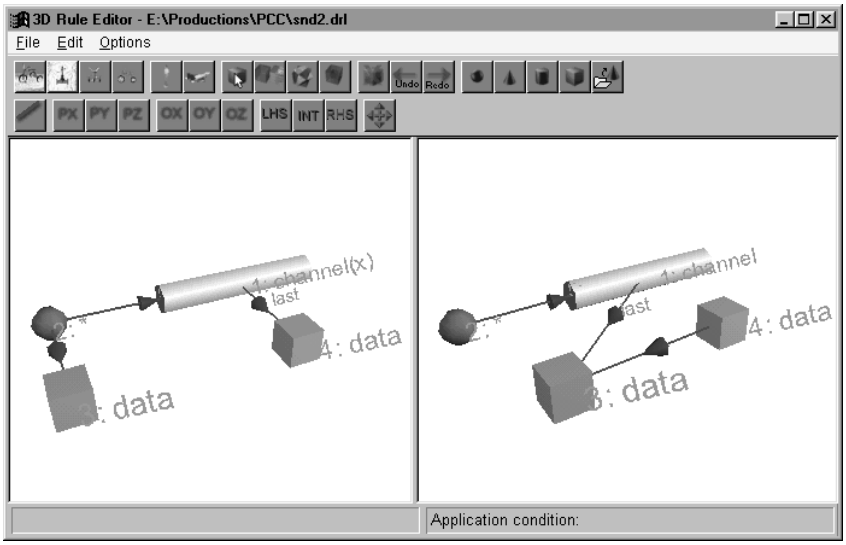


Abbildung 5-18: Regeleditor

5.4.3 Programmeditor

In der vorliegenden Version von GRACEland ist der Programmeditor ein einfacher Texteditor mit Syntaxhervorhebung. Die favorisierte Darstellung, in der Regeln als Bilder in den Text eingefügt werden, konnte mit der benutzten Bibliothek für den Editor bisher nicht realisiert werden.

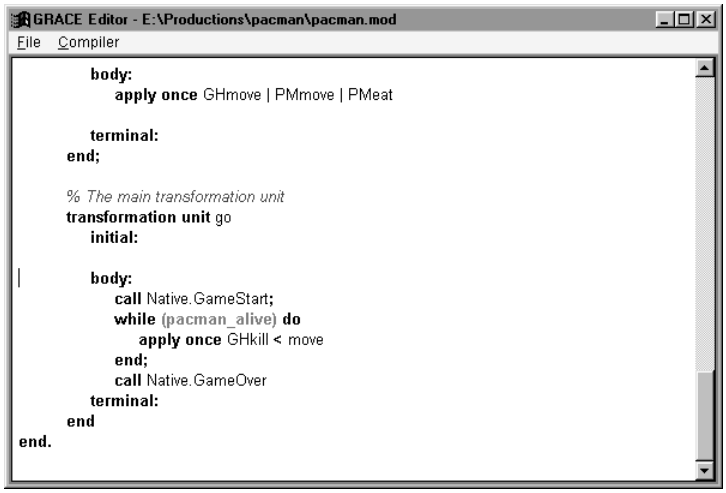


Abbildung 5-19: Programmeditor

GRACE Programme werden als reine Textdateien, d.h. ohne Syntaxhervorhebung gespeichert. Die Tabelle zeigt die Zuordnungen von lexikalischen Einheiten zum Textformat:

Lexikalische Einheiten	Textformat
Kommentar	*Grün und Kursiv*
Reservierte Schlüsselwörter	**Schwarz und Fett**
Zahlen	Rot
Graphausdrücke	**Grau und Fett**
Bezeichner	Schwarz

Die verschiedenen Textformatierungen sollen dem Benutzer helfen, die Struktur der Programme deutlicher zu erkennen.

Um zu prüfen, ob das Programm syntaktisch korrekt ist, kann das aktuell geladene Programm übersetzt werden. Fehler im Programm werden dabei angezeigt und können korrigiert werden. Soll das Programm gestartet werden, so kann der GRACE Interpreter aufgerufen werden.

Als alternative Darstellung besteht die Möglichkeit, ein Programm im HTML3 Format, mit Syntaxhervorhebung zu speichern, wobei Graphregeln als Bilder in den Text eingefügt werden (vgl. Abbildung 5-5, Seite 57). Die Bilder werden automatisch erzeugt und weisen daher nicht immer den optimalen Standpunkt und Blickrichtung auf, was dazu führen kann, daß Teile verdeckt oder abgeschnitten werden. Um die linke und die rechte Seite einer Graphregel auseinanderhalten zu können, werden verschiedene Hintergrundfarben benutzt.

5.5 GRACE Interpreter

Der GRACE Interpreter basiert auf dem Grapheditor, läßt aber keine Änderungen am Graphen durch den Benutzer zu. Die einzige Interaktivität für den Benutzer liegt in der Veränderung des Standpunktes und der Blickrichtung. Abbildung 5-20 zeigt einen Ausschnitt aus dem berechneten Fibonacci-Baum für fib(6).

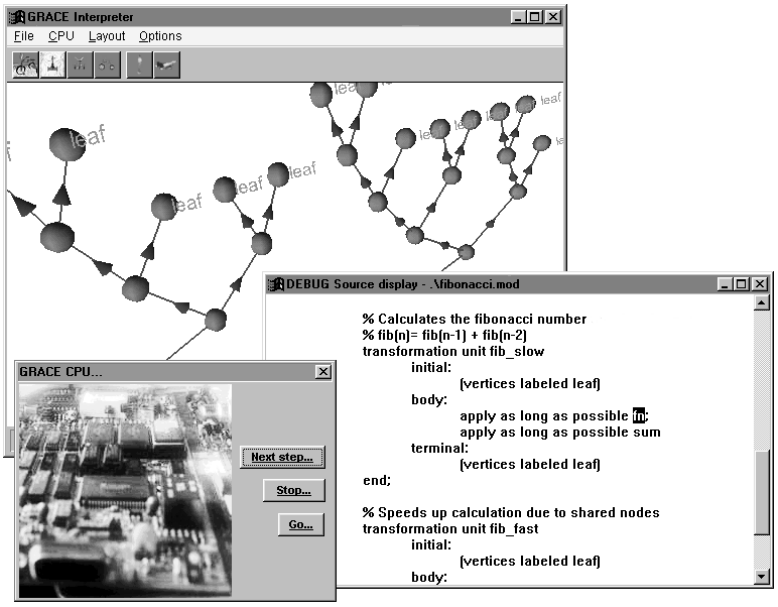


Abbildung 5-20: GRACE Interpreter

Der Ableitungsprozeß wird durch Start unter dem Menü CPU gestartet. Es erscheint eine Dialogbox (Abbildung 5-21 a)), welche die exportierten Transformationseinheiten der geladenen Module anzeigt (vgl. Klasse graceDB, Kapitel 4.4.5). Die angezeigten Namen setzen sich aus dem Modulnamen und dem der Transformationseinheit zusammen. Möchte der Benutzer eine andere Transformationseinheit starten, so kann das entsprechende Modul über den Button Load nachgeladen werden.

Nachdem eine Transformationseinheit ausgewählt und mit Ok bestätigt wurde, wird der Interpreter gestartet. Der Benutzer kann nun die Transformationen am Graphen mitverfolgen. Visualisiert werden der benutzte Ansatz einer Graphregel, die Veränderung am Graphen sowie Zwischenschritte einer Regelausführung und an welcher Stelle man sich im Sourcecode befindet. Oftmals laufen die Transformationen relativ schnell ab, so daß ein genaues Verfolgen nicht möglich ist. Dann ist es wünschenswert, Pausen zwischen den einzelnen Transformationen einlegen zu können. Unter dem Menü CPU befindet sich der Menüpunkt Properties, über den die Eigenschaften der Visualisierung eingestellt werden können (Abbildung 5-21 b)).

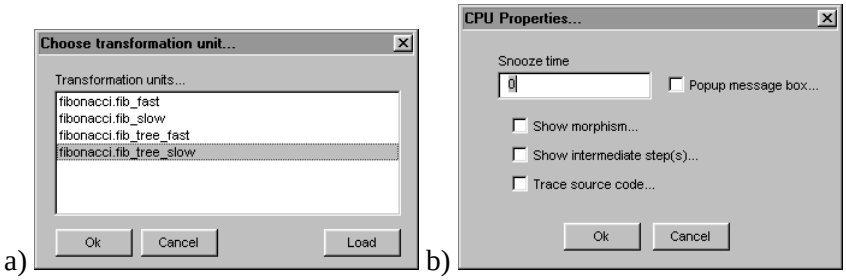


Abbildung 5-21: a) Auswahl der auszuführenden Transformationseinheit b) Eigenschaften der Visualisierung

Nach einer Regelanwendung besitzen neu hinzugekommene Knoten eine zufällige Position. Je öfter Regeln angewendet werden, desto wirrer wird die Darstellung des Graphen. Um hier Abhilfe zu schaffen, kann ein Layout-Algorithmus bestimmt werden, der nach jeder Regelanwendung ausgeführt wird.

5.6 Layout von Graphen

GRACEland definiert eine abstrakte Schnittstelle zu Layout-Algorithmen, die den Graphen entwirren und neu ausrichten sollen. Dazu muß für jeden Algorithmus eine dynamische Bibliothek vorhanden sein, welche die vorgegebenen Funktionen implementiert. Zur Laufzeit werden diese nachgeladen und entsprechende Funktionen aufgerufen.

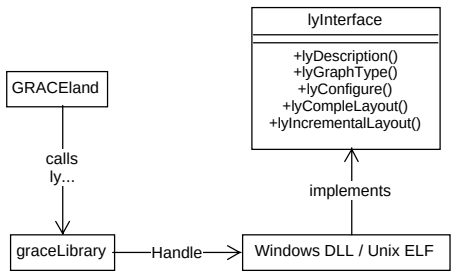


Abbildung 5-22: Schema des Layout-Interfaces

Die Schnittstelle umfaßt die Funktionen: lyDescription, lyGraphType, lyConfigure, lyCompleteLayout und lyIncrementalLayout.

Die Funktion lyDescription liefert für den Benutzer eine kurze Beschreibung des Algorithmus, da aus dem Namen dessen Funktionalität nicht immer ersichtlich ist.

Layout-Algorithmen werden für eine bestimmte Graphklasse geschrieben, d.h. man kann einen Algorithmus für gerichtete Graphen nicht auf ungerichtete anwenden. Deshalb muß die Graphklasse (lyGraphType) abgefragt werden können.

Darüber hinaus können die Algorithmen oftmals über Parameter an verschiedene Situationen angepaßt werden. Es muß deshalb eine Schnittstelle (lyConfigure) geben, über die der Benutzer die Parameter einstellen kann.

Die Interfacedefinition sieht zwei Funktionen zum Ausrichten von Graphen vor: lyCompleLayout und lyIncrementatlLayout. Die Funktion lyCompleteLayout richtet den gesamten Graphen neu aus, während lyIncrementalLayout nur einen Teil neu ausrichtet. Dazu bekommt diese Funktion als zusätzlichen Parameter die veränderten Knoten.

Die Funktion lyIncrementalLayout wird derzeit noch nicht benutzt, soll aber in naher Zukunft eingesetzt werden. Sie bietet sich für das Layouten eines Graphen nach der Anwendung einer Regel an, da nur Teile des Graphen verändert werden. Bei großen Graphen ließe sich dadurch eine hohe Geschwindigkeitssteigerung erzielen.

Der Benutzer spricht die Layout-Algorithmen über den folgenden Dialog an:

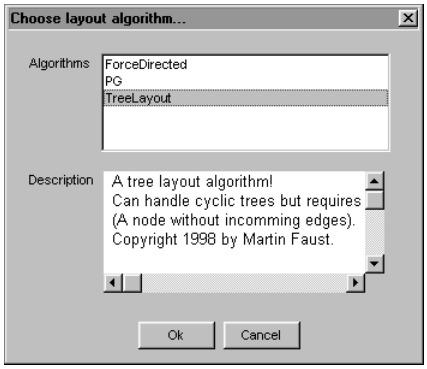


Abbildung 5-23: Auswahldialog von Layout-Algorithmen

Der Anwender sieht dabei nur die Namen und die Beschreibung der Algorithmen. Alles andere bleibt verborgen. Abbildung 5-24 zeigt die Anwendung des "ForceDirected" Layoutverfahrens auf einen Graphen.

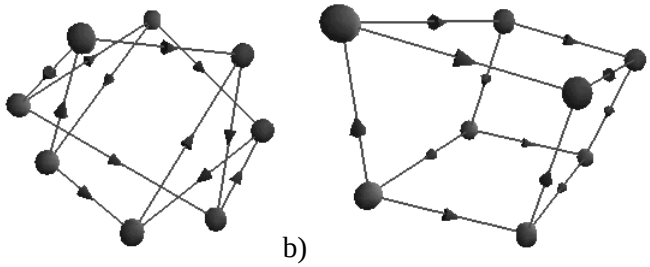


Abbildung 5-24: a) Graph vor dem Layouten b) Graph nach dem Layouten

6 ANWENDUNGEN

6.1 Übersicht

Graphen und Algorithmen auf diesen werden in vielen verschiedenen Applikationen eingesetzt. Dabei muß man zwischen Anwendungen unterscheiden, die "nur" Algorithmen auf Graphen benutzen, und Anwendungen, die Graphersetzungssysteme einsetzen. Im ersten Fall werden nur theoretische Erkenntnisse benutzt, um spezielle Algorithmen zu entwickeln, während im zweiten Fall alle Algorithmen durch Graphregeln implementiert werden.

Interessant ist in diesem Zusammenhang die Frage, ob die Algorithmen (aus Fall 1) sich auch in ein Graphersetzungssystem implementieren lassen. In einigen Fällen wird dieses an speziellen Anforderungen der Anwendung scheitern, da meistens noch auf andere Datenquellen zugegriffen werden muß. Die Lösung ist eine Kombination von imperativen Programmiersprachen und dem Graphersetzungssystem. Hier zeigt sich die Überlegenheit von GRACE und der vorgestellten Implementierung. Zum einen ist GRACE nicht auf einen spezifischen Graphersetzungsansatz festgelegt und zum anderen kann der Interpreter mit in ein Programm eingebunden werden und von dort aus gestartet werden. Dadurch hat man die Möglichkeit, komplexe Algorithmen auf Graphen elegant zu implementieren. Meistens muß kein neuer Ansatz implementiert werden, weil der Hauptteil der Anwendungen auf Standardsysteme, wie z.B. gerichtete Graphen, aufsetzen.

Die Palette der Einsatzgebiete von Graphersetzungssystemen ist sehr vielfältig und umfaßt unter anderem folgende Gebiete:

- **Programmiersprachen**
  - Optimierungen von Programmen ([Ass95])
  - Grundlage zur Implementierung von funktionalen Programmiersprachen ([ESP97])
  - Grundlage von visuellen Programmiersprachen ([Bar97])
- **Erkennungsprobleme**
  - Erkennung von mathematischen Ausdrücken ([GB95])
  - Erkennung von graphischen Symbolen ([Mes95])
  - Erkennung von Features in CAD/CAM ([Kla94])
  - Objekterkennung für Bildsuche ([Kla95])
- **Spezifikationen**
  - Prozeßspezifikation und -verifikation ([EHT+97])
  - Modellierung von verteilten dynamischen Systemen ([Tae96])

- Graphische Benutzungsschnittstellen ([Suc96], [GS96], [GTS98])

- **Grafikanwendungen**
  - L-Systeme ([Boe95], [DK98])
  - Collagen ([DK98])
  - Oberflächenrekonstruktion ([Men95]).

Im folgenden werden Beispiele vorgestellt, welche die Anwendungsmöglichkeiten von Graphersetzungssystemen und zugleich Facetten der Implementierung zeigen. Dabei wurde weniger Wert darauf gelegt, in jedem Fall ein komplettes System mit allen Extras zu implementieren, sondern die Beispiele sollen vielmehr die einzelnen Komponenten der Sprache GRACE vorstellen. Es wurden vereinfachende Annahmen gemacht, um die Beispiele nicht zu lang werden zu lassen. Programme können schnell über 30 Seiten lang werden, je mehr Eigenschaften sie berücksichtigen.

Insgesamt werden drei Beispiele aus den Bereichen Mustererkennung, Spezifikation und Simulation vorgestellt. Die verschiedenen Gebiete zeigen eindrucksvoll das Potential, das in Graphersetzungssystemen steckt. So konnten z.B. Petri-Netze um eine dynamische Komponente erweitert werden, die sich mit den vorhandenen Tools nicht simulieren läßt.

6.2 Spezifikation eines dynamischen Systems

Graphersetzungssysteme lassen sich auch als Spezifikationssprache einsetzen. Spezifikationen beschreiben ein System, wobei von einigen Details abstrahiert wird, unabhängig von einer späteren Implementierung. Die Spezifikation soll vor allem dabei helfen, das Problem zu verstehen und seine Eigenheiten zu erkennen. Verbreitet sind Methoden, wie UML, Booch und CSP ([Hoa5]).

Bei UML und Booch handelt es sich um graphische Notationen, die Objekte und ihre Relationen zueinander beschreiben. Sie bieten eine ausgezeichnete Unterstützung für statische Strukturen, aber nur eine unzureichende für dynamische, wie z.B. Interaktion zwischen Objekten. Bei CSP handelt es sich um eine Sprache, die kommunizierende sequentielle Prozesse beschreibt. Dabei steht vor allem die Interaktion zwischen den Prozessen im Vordergrund, d.h. das Senden und Empfangen von Nachrichten.

Graphersetzungssysteme sind eine Alternative zu herkömmlichen Spezifikationssprachen. Ihr Vorteil ist, daß sowohl statische als auch dynamische Aspekte beschrieben werden können. Graphen stellen den aktuellen Zustand des Systems dar, wobei Knoten Objekten des zu modellierenden Systems und Kanten Relationen zwischen diesen entsprechen. Markierungen und Attribute erweitern die Graphelemente um anwendungsspezifische Informationen oder Werte von Ressourcen. Mit Hilfe von Graphregeln wird das dynamische Verhalten des Systems beschrieben.

Mit diesen Einheiten ist ein formales Modell für Spezifikationen geschaffen worden, das auf eine fundierte und breite theoretische Grundlage zurückgreifen kann. Graphersetzungssysteme bieten eine ausgezeichnete Balance zwischen Ausdrucksstärke auf der einen Seite und Verständlichkeit auf der anderen. Die Ausdrucksstärke ist in den Graphregeln begründet, da

sich z.B. komplexe Anwendungsbedingungen einfach integrieren lassen. Die Verständlichkeit basiert auf der visuellen Darstellung. Durch den Einsatz von Tools, wie GRACEland, kann eine Spezifikation sofort ausgeführt werden, wodurch sich Fehler frühzeitig erkennen und beheben lassen. Darüber hinaus werden Methoden zur Dekomposition bereitgestellt (Module und Transformationseinheiten).

In der Praxis werden Graphersetzungssysteme selten eingesetzt, doch erste Erfolge haben sich bereits gezeigt. So konnte das Design einer graphischen Oberfläche entscheidend verbessert werden ([NPR+96]). Dabei wurde die graphische Oberfläche durch Graphen und Graph-Grammatiken beschrieben ([Suc96], [GS96]).

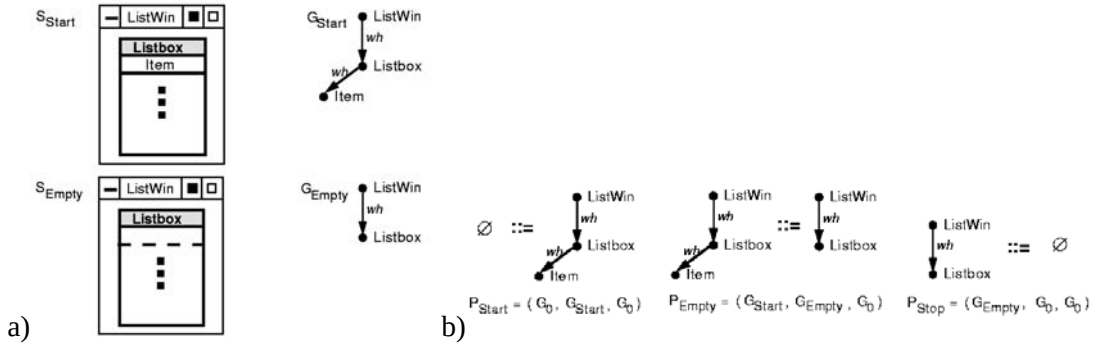


Abbildung 6-1: Spezifikation von graphischen Oberflächen (aus [Suc96], Seite 196)
a) Darstellung von GUI-Elementen als Graphen b) Graphregeln

Abbildung 6-1 a) zeigt die Darstellung von graphischen Oberflächen als Graphen am Beispiel eines Fensters mit einer Listbox. Die Knoten des Graphen stehen für Entitäten und Kanten kennzeichnen die Hierarchie der GUI-Elemente. Die Markierung wh steht dabei für "widget hierarchy", ein von Unix geprägter Begriff. In b) sind die Interaktionen beschrieben, die sich auf Öffnen des Fensters, Löschen des Inhaltes der Listbox und Schließen des Fensters beschränken.

Ein weiteres Beispiel für ein dynamisches System sind kommunizierende Prozesse, wie sie z.B. Client-Server-Systemen auftreten. Durch Graphen wird der statischen Anteil, d.h. der Aufbau des Netzwerkes, modelliert. Graphregeln geben an, wie sich der Zustand des Systems ändert.

Im folgenden wird ein einfaches dynamisches System, bestehend aus einem Drucker, einem Klienten und einem Übertragungskanal, modelliert. Das dynamische Verhalten des Systems besteht darin, daß der Klient Dokumente erzeugt und diese an den Drucker schickt. Die Übertragung wird dabei von einem Kanal übernommen, wobei vereinfachend angenommen wird, daß keine Fehler auftreten. Der Drucker empfängt die Dokumente vom Kanal und druckt sie aus.

Die folgende Abbildung zeigt die statische Struktur des Systems:

Knoten modellieren die im System vorhandenen Entitäten Klient, Kanel und Drucker. Kanten stellen logische Verknüpfungen zwischen den Elementen dar. Die Kante vom Klienten zum Drucker besagt, daß der Klient auf den Kanal senden kann. Ein Empfangen ist nicht vorgesehen, weshalb auch keine Verbindung vom Kanal zum Klienten existiert. Die unterschiedlichen Geometrien und Farben sollen dabei helfen die Übersichtlichkeit zu bewahren.

Wenn der Klient ein Dokument erzeugt, so wird dieses durch einen neuen Knoten mit der Markierung "data" dargestellt. Gleichzeitig wird eine Kante vom Besitzer, hier der Klient, zum Dokument erzeugt, die eine gehört-zu-Beziehung modelliert. Beim Senden und Empfangen wird nur die gehört-zu-Beziehung verändert. Wenn das Dokument beim Drucker angekommen ist, wird der Dokumentknoten gelöscht, was den Vorgang des Ausdruckens simuliert.

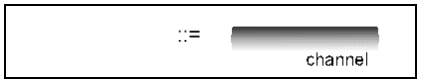
Insgesamt wird das System in vier Modulen spezifiziert, wovon drei das Verhalten des Klienten, des Kanals und des Druckers beschreiben. Das vierte Modul definiert globale Vorgänge. Zunächst erzeugt es die statische Struktur des Systems ("init", Seite 83) und startet dann die nebenläufigen Prozesse des Klienten und des Druckers. Da die aktuelle Implementierung von GRACEland keinen Prozeßbegriff besitzt, mußte dieses durch den nichtdeterministischen Aufruf der zwei Transformationseinheiten client.action und printer.print simuliert werden
(vgl. "go", Seite 83).

% The specification of the channel
module channel

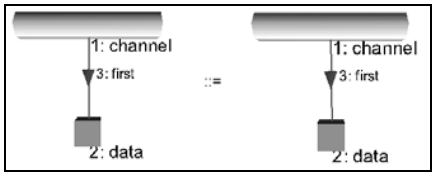
graph class: DirectedGraph

exports
transformation unit create**;**
transformation unit data_available**;**
transformation unit send**;**
transformation unit receive

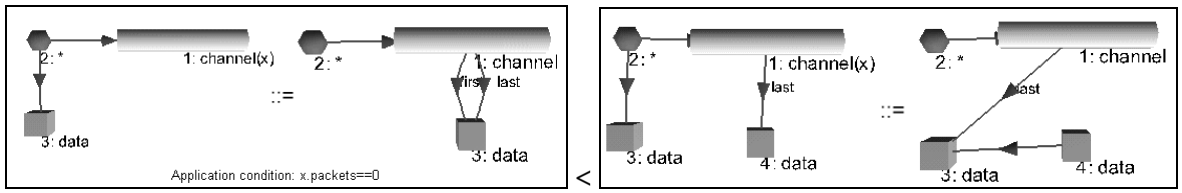
realized by

transformation unit create
initial:
body:
apply once 

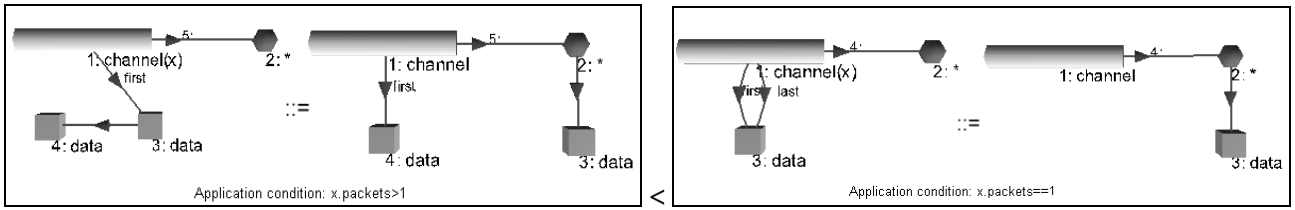
terminal:
end;

transformation unit data_available
initial:
body:
apply once 

terminal:
end;
% Sends data onto the channel

transformation unit send
initial:
body:
apply once 

terminal:
end;

% Receives data from the channel
transformation unit receive
initial:
body:
apply once 

terminal:
end
end.

% The specification of the client
module client

graph class: DirectedGraph

uses printer**,** channel**;**

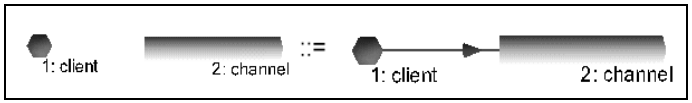
exports
transformation unit connect**;**
transformation unit create**;**
transformation unit action

realized by

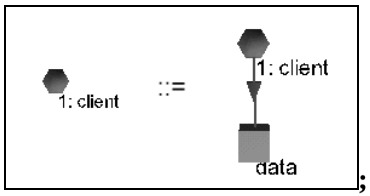
transformation unit create
initial:
body:
apply once

terminal:
end;

transformation unit connect
initial:
body:
apply once



terminal:
end;

```
    transformation unit action
        initial:
        body:
            apply once 

;
            apply once channel.send
        terminal:
    end
end.
```

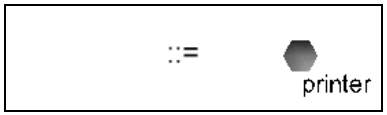
% The specification of the printer
module printer

graph class: DirectedGraph

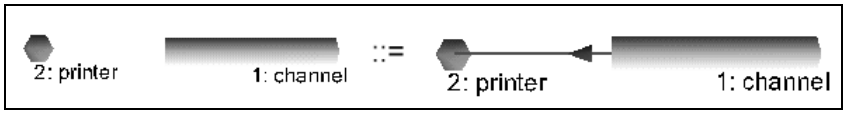
uses channel**;**

exports
transformation unit create**;**
transformation unit connect**;**
transformation unit print

realized by

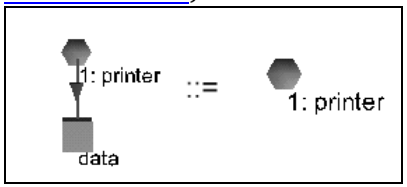
transformation unit create
initial:
body:
apply once


terminal:
end;

transformation unit connect
initial:
body:
apply once


terminal:
end;

```
        % Print a document
        transformation unit print
            initial:
            body:
                % Check, if a document is available
                if (channel.data_available) then
                    apply once channel.receive;
                    apply once
                end
            terminal:
        end
end.
```



```
% The specification of the whole system
module system

    graph class: DirectedGraph

    uses printer, client;

    exports
        transformation unit go
    realized by

        transformation unit init
            initial:
            body:
                apply once printer.create;
                apply once channel.create;
                apply once client.create;
                apply once printer.connect;
                apply once client.connect
            terminal:
        end;

        transformation unit go
            initial:
            body:
                apply once init;
                apply as long as possible client.action | printer.print
            terminal:
        end
end.
```

6.3 Petri-Netze

Petri-Netze sind ein Modell, mit dem man nebenläufige Prozesse und nichtdeterministische Vorgänge beschreiben und analysieren kann. Sie sind nach ihrem Erfinder C.A. Petri benannt, der sie 1962 entwickelt hat. Petri-Netze werden heute in vielen verschiedenen Anwendungen zur Spezifikation des Systems benutzt. Analysemöglichkeiten schließen u.a. Erreichbarkeit und Test auf Verklemmungsfreiheit von Transitionen mit ein.

Ein Petri-Netz ist ein gerichteter Graph, wobei die Knoten in Stellen und Transitionen unterteilt werden. Eine Stelle dient zur Speicherung von Daten, während eine Transition einen Verarbeitungsschritt beschreibt. Eine Kante darf dabei nur von Stellen (Eingabestellen) zu Transitionen bzw. von Transitionen zu Stellen (Ausgabestellen) führen. Üblicherweise werden Stellen als Kreise, Transitionen als Rechtecke und Token als gefüllte Kreise in den Stellen dargestellt. Der so aufgebaute Graph definiert die Struktur des Systems.

Dynamische Vorgänge werden durch Token simuliert, die durch das Netz wandern. Geregelt wird dieses durch Schaltregeln für Transitionen. Eine Transition schaltet, wenn auf jeder Eingabestelle mindestens ein Token ist. Dann wird von jeder Eingabestelle ein Token abgezogen und zu jeder Ausgabestelle einer hinzugefügt.

Das hier beschriebene Petri-Netz ist die Grundlage für verschiedene Erweiterungen, wie z.B. Zeit-Transitions-Netze. Eine ausführliche Einführung in die Theorie von Petri-Netzen findet man u.a. in [Rei92] und [Bau90].

Petri-Netze werden hauptsächlich eingesetzt, um nebenläufige Prozesse zu beschreiben. Das zu modellierende System muß allerdings statisch sein, d.h. die zugrundeliegende Struktur darf sich nicht ändern. Sollen allerdings Vorgänge modelliert werden, in denen sich die Basisstruktur ändert, so eignen sich Petri-Netze nicht. Sie sind durch einen statischen Graphen auf eine bestimmte Situation festgelegt. Ein Beispiel für diese Situation ist ein einfaches Client-Server-System mit gemeinsam benutzten Ressourcen. Solange eine feste Anzahl an Clients und Servern vorhanden ist, kann es modelliert werden. Wenn nun neue Clients oder Server hinzukommen oder Clients sich abmelden, schon nicht mehr. Diese Art von Systemen kommt häufig vor und ist nicht nur ein konstruiertes Beispiel. Als Lösung bietet sich der Einsatz von Graphersetzung an, da hiermit sowohl statische, als auch dynamische Vorgänge abgebildet werden können.

Im folgenden wird zunächst gezeigt, wie Petri-Netze sich in Graphersetzungssystemen abbilden lassen. Danach wird an einem Beispiel die Erweiterung um eine dynamische Komponente vorgestellt.

Abbildung von Petri-Netzen in Graphersetzungssysteme

Es existieren zwei verschiedene Möglichkeiten, Petri-Netze in Graphen und Graphregeln umzusetzen. Stellen, Transitionen und Kanten können direkt übernommen werden. Die zwei Darstellungen unterscheiden sich nur in der Repräsentation der Token. In der ersten Variante werden sie als Attribute den Stellen zugeordnet, während sie in der zweiten Variante als eigenständige Knoten mit einer Verbindung zu den Stellen behandelt werden.

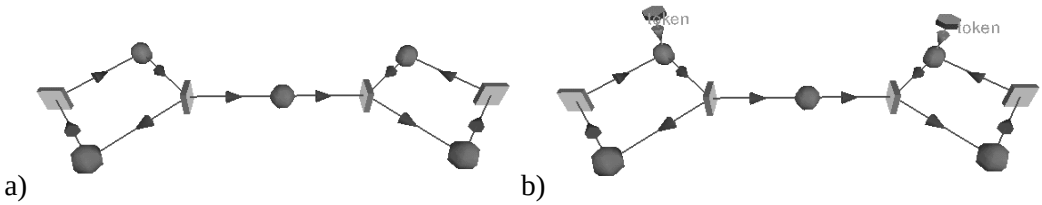


Abbildung 6-2: Unterschiedliche Repräsentationen von Petri-Netzen
a) Token als Attribute der Stellen b) Token als eigenständige Knoten

Der Vorteil der zweiten Lösung ist, daß die Token sichtbar sind und direkt in eine Graphregel integriert werden. In der ersten Lösung werden Token nur durch die Anwendungsbedingung bzw. durch Berechnungen auf Attributen benutzt. Dadurch fehlt die intuitive Darstellung, wie Token verändert werden. Mit einer speziellen Visualisierung und auf Petri-Netze zugeschnittene Editoren könnte dieses Problem umgangen werden. Es wird hier die zweite Lösung benutzt, da sie anschaulicher ist.

Das Schalten von Transitionen entspricht dem Anwenden einer Graphregel. Abbildung 6-3 zeigt eine entsprechende Graphregel. Die Anwendungsbedingung dient dazu, sicherzustellen, daß genau zwei Eingänge und ein Ausgang vorhanden sind. Ansonsten könnte sie z.B. bei Transitionen mit drei Eingängen und einem Ausgang angewendet werden.

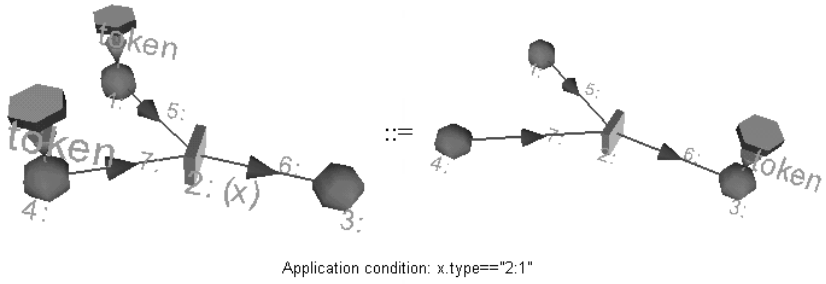


Abbildung 6-3: Schaltregel für eine Transition mit zwei Eingängen und einem Ausgang

Beispiel eines Client-Server Systems

Damit sind wir nun in der Lage, Standard-Petri-Netze durch Graphen und Graphregeln zu beschreiben. Im folgenden wird an einem Client-Server-System die Erweiterung um dynamische Vorgänge gezeigt. Server ist in diesem Beispiel ein Drucker, auf den die Klienten ihre Dokumente ausdrucken wollen.

Zunächst wird ein statisches System mit zwei Klienten und einem Drucker vorgestellt, welches im Anschluß um eine dynamische Komponente, daß an- und abmelden von Klienten, erweitert wird.

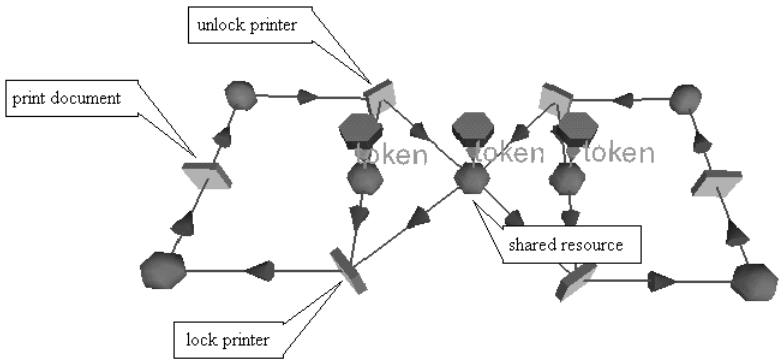


Abbildung 6-4: Petri-Netz-Modell des Systems mit zwei Klienten

Abbildung 6-4 zeigt das Petri-Netz-Modell des beschriebenen Systems. Bevor ein Dokument gedruckt werden kann, muß der alleinige Zugriff auf den Drucker gesichert sein. Dieses wird durch die Transition "lock printer" modelliert. Sie blockiert die "print document" Transition so lange, bis der Drucker freigegeben ist. Die Freigabe des Druckers wird durch die Stelle ("shared resource") in der Mitte modelliert. Enthält sie einen Token, so wird der Drucker momentan nicht benutzt, ansonsten ist er von einem Klienten belegt. Die Transition "unlock printer" gibt den Drucker frei, so daß auch andere Klienten drucken können.

Zur Beschreibung der Schaltvorgänge benötigen wir neben der Regel aus Abbildung 6-3 folgende zwei Graphregeln:

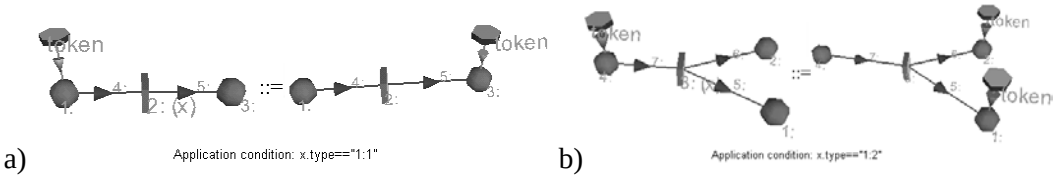


Abbildung 6-5: Schaltregeln für eine Transition mit
a) einem Eingang und einem Ausgang b) einem Eingang und zwei Ausgängen

Das GRACE-Programm besteht nur aus einer Transformationseinheit, in der die drei Graphregeln so lange wie möglich angewendet werden, die im folgenden mit rule1, rule2 und rule3 abgekürzt werden:

module Synchronisation

graph class: DirectedGraph

exports
transformation unit static

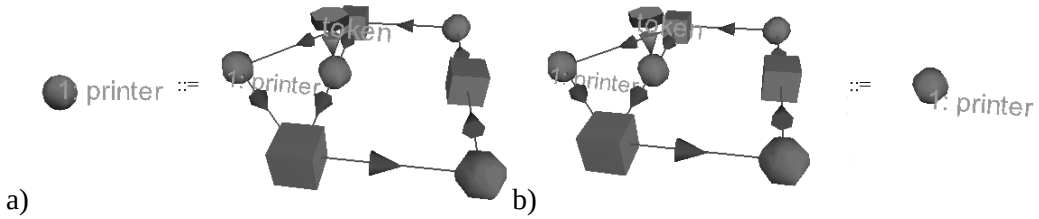
realized by

transformation unit static
initial:
(template TwoClients)

```
        body:
            apply as long as possible rule1 | rule2 | rule3
        terminal:
    end
end.
```

Die initiale Bedingung der Transformationseinheit static stellt sicher, daß der initiale Graph dem aus Abbildung 6-4 entspricht.

Realistischer wird die Modellierung, wenn man annimmt, daß neue Klienten sich anmelden und ihrerseits Druckaufträge abschicken oder aber, daß Klienten sich abmelden. Das An- und Abmelden von Klienten kann wie folgt modelliert werden:



a) b)

Abbildung 6-6: a) Anmelden neuer Klienten b) Abmelden von Klienten

Dabei mußte der zugrunde liegende Graph ein wenig geändert werden, damit ein eindeutiges Hinzufügen bzw. Entfernen von Klienten möglich ist. Die Änderung betrifft allerdings nur die Ressource, die den eindeutigen Namen "printer" bekommt. Als Konsequenz müssen auch die Regeln geändert werden, da sie leere Markierungen in Stellen und Transitionen fordern. Dieses geschieht am einfachsten, indem der Platzhalter "*" verwendet wird, der alle Markierungen zuläßt. Nun ist ein eindeutiges Hinzufügen und Entfernen möglich und geschieht entsprechend den Regeln aus Abbildung 6-6.

Der Startgraph enthält nur noch den freigegebenen Drucker. Das modifizierte GRACE-Programm lautet:

```
module Synchronisation

    graph class: DirectedGraph

    exports
        transformation unit dynamic

    realized by

        transformation unit dynamic
            initial:
                (template Clients)
            body:
                apply as long as possible rule1 | rule2 | rule3 | add | remove
            terminal:
        end
end.
```

Damit ist gezeigt, daß Graphersetzungssysteme mächtiger als Petri-Netze sind, da sie auch dynamische Vorgänge modellieren können. Durch das Modulkonzept von GRACE bietet sich darüber hinaus noch die Möglichkeit, Petri-Netze in kleine Teile aufzuspalten und zu strukturieren. Petri-Netze bieten die Möglichkeit, Analysen, wie z.B. Erreichbarkeit von Stellen, durchzuführen. Durch Transformationseinheiten lassen sich auch Analysemethoden integrieren. Reicht der Graphersetzungsansatz nicht aus, so können immer noch externe Module aufgerufen werden.

6.4 Erkennung von graphischen Symbolen

Erkennung von graphischen Symbolen ist eine Aufgabe, die eigentlich zum Gebiet der Mustererkennung gehört und man zunächst nicht mit Graphersetzungssystemen in Verbindung bringen würde. Die Aufgabe von Mustererkennung ist es, ein bestimmtes Muster in einem anderen zu erkennen und auszugeben. Das größte Problem sind Störungen der Daten, die sich z.B. in nicht korrekt ausgerichteten Objekten zeigt.

Graphersetzung bietet die Möglichkeit eine Rotations-, Translations- und Skalierungsinvariante Erkennung zu implementieren. Voraussetzung ist, daß die zu erkennenden Bilder aus Geradenstücken zusammengesetzt sind.

Die Grundidee ist es, jedes Objekt als Graph darzustellen und den Erkennungsprozeß als Ansatzsuche zu definieren.

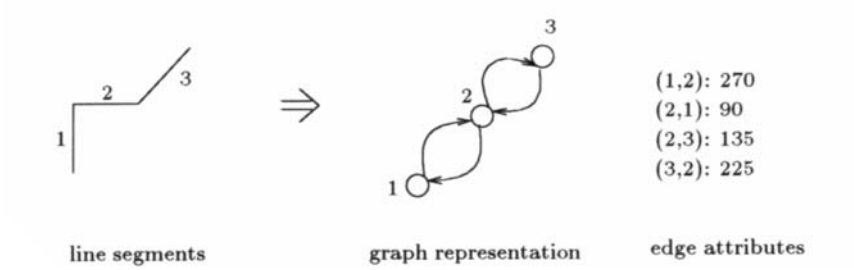


Abbildung 6-7: Graphrepräsentation von Linienzeichnungen ([Mes95], Seite 111)

Abbildung 6-7 zeigt die Graphrepräsentation von Linienzeichnungen. Jede Linie wird durch einen Knoten repräsentiert. Besitzen zwei Linien einen gemeinsamen Endpunkt, so werden zwei gerichtete Kanten zwischen den entsprechenden Knoten eingefügt. Einer Kante wird als Attribut der Winkel zwischen den Linien zugeordnet. Diese gewählte Darstellung von Objekten ist invariant gegenüber Translation, Rotation und Skalierung. Allerdings wird in dieser Form nicht auf unterschiedlich lange Liniensegmente eingegangen, so daß, wenn Linie 2 doppelt so lang wäre, derselbe Graph entstehen würde. Dieses kann unter Umständen zu Verwechslungen führen. Eine Lösung ist, als weiteres Attribut, die Länge einer Linie entweder den Knoten oder den Kanten zuzuordnen. Messmer differenziert an dieser Stelle nicht weiter, weshalb auch hier darauf verzichtet werden soll.

Die Erkennung ist als Ansatzsuche definiert. Für jedes zu erkennende Objekt existiert eine Graphregel, deren linke Seite die Graphrepräsentation des Objektes enthält und deren rechte

Seite beliebig ist. Eine Transformationsregel kann man als Ersetzung des Bildes eines Objektes z.B. durch ein Schaltungssymbol ansehen. Somit kann man Stück für Stück die Schaltung rekonstruieren.

Probleme bei der Erkennung ergeben sich, wenn ein Objekt Teil eines anderen ist. Wird zunächst das Teilobjekt erkannt und durch ein nicht identisches ersetzt, so kann das "größere" Objekt nicht mehr erkannt werden. Hier muß eine Strategie definiert werden, um diese Situation zu entschärfen. Die einfachste Strategie besteht in der Vergabe von Prioritäten für die Regelauswahl.

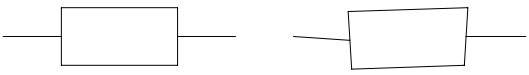
Die einzelnen Schritte für die Erkennung z.B. einer Schaltung sind:

1. Digitalisierung der Zeichnung,
2. Erzeugen der Graphrepräsentation,
3. Graphersetzungssystem starten,
4. Schaltung nachbearbeiten.

In der Theorie sieht dieses sehr gut aus, doch in der Praxis ist der Schritt 2 nicht immer eindeutig, da die Zeichnungen Fehler aufweisen können. So sind z.B. Linienzüge nicht immer geschlossen oder in kleine Einzelteile aufgespalten. Messmer definiert sechs Arten von Fehlern und wie sie behoben werden können. Darüber hinaus werden die einzelnen Operationen mit Kosten versehen, die empirisch ermittelt werden. Die Ansatzsuche wird durch eine fehlertolerante Suche ersetzt, deren Ziel es ist, die Kosten zu minimieren, die benötigt werden, um ein Graphenmorphismus zwischen Vorlage und Bild zu finden.

Um ein Beispiel in GRACEland definieren zu können, gehen wir hier davon aus, daß keine Fehler vorhanden sind. Der Vorschlag von Messmer läßt sich in GRACE integrieren, indem ein neuer Ansatz definiert wird, der die fehlertolerante Ansatzsuche implementiert. Um das Beispiel weiterhin klein zu halten, werden Schritt 1 und 2 übersprungen (siehe dazu [BB93]).

Die Implementierung von Graphregeln für die Erkennung von einzelnen Objekten stellt kein Hindernis dar. Wie aber können auftretende Schwankungen in den Winkeln der einzelnen Objekte mit in die Regel integriert werden (siehe Abbildung) ?



Dazu muß nur eine Abweichung der Winkel des Bildes zur Vorlage zugelassen werden. D.h. wenn alle Winkel innerhalb eines Toleranzbereiches liegen, gilt das Objekt als erkannt. Dieses läßt sich in die Graphregel durch Definition einer Anwendungsbedingung integrieren. Es muß nur für jeden Winkel abgefragt werden, ob er in dem Toleranzbereich liegt:

(v1.angle >265 && v1.angle < 275) && (...) && (v6.angle > 230 && v6.angle < 235).

Diese Methode ist zwar theoretisch anwendbar, führt aber zu endlosen Anwendungsbedingungen und ist somit nicht zu gebrauchen. Die Anzahl der Abfragen läßt sich reduzieren, indem wir eine externe Funktion aufrufen, die den Vergleich durchführt: $(pattern.tolerance)(v1.angle, 270). Nun benötigt man aber immer noch so viele Abfragen wie Kanten vorhanden sind. Hier läßt sich eine Eigenheit der Implementierung nutzen. In der aktuellen Umgebung des Ausdruckshändlers sind nur Attribute von Knoten und Kanten der

aktuellen Regel vorhanden, die eine Variable deklariert haben. D.h. man kann die gesamte Überprüfung der Toleranzen auf die Funktion übergeben. Damit sieht die Anwendungsbedingung wie folgt aus: $(pattern.match)("angle", 5). Der folgende Ausschnitt zeigt, wie die Anwendungsbedingung berechnet wird:

```
forall(pVar, plVariables)
{
  if (pVar->Name.SearchString(".angle") != -1)
  {
    if (pVar->Name.SearchString(".org") != -1)
        name = pVar->Name(0, pVar->Name.Length()-5);
    else
        name = pVar->Name + ".org";

    pV1 = (exprValueReal *) pEnvironment->Get(name);
    pV2 = (exprValueReal *) pVar->pValue;

    if (fabs(fabs(pV1->GetValue()) - fabs(pV2->GetValue())) > tolerance)
        return new exprValueBoolean(exprFALSE);
  }
}

return new exprValueBoolean(exprTRUE);
```

Dabei treten zwei Fälle auf: Entweder liegt ein Attribut des Originals oder ein Attribut des Bildes vor. Je nachdem wird das entsprechend andere gesucht und diese miteinander verglichen. Dabei werden die Attribute doppelt überprüft, was kaum Zeitverluste zur Folge hat, da die Anzahl der Einträge in der Umgebung relativ gering ist.

Am Beispiel einer elektrischen Schaltung, bestehend aus Widerständen, Lampen und Meßgeräten, soll das Konzept demonstriert werden. Die rechten Seiten der Graphregeln sind so geschrieben, daß die Ein- und Ausgänge des erkannten Objektes erhalten bleiben und gleichzeitig ein Knoten mit der Symbolbezeichnung eingefügt wird. Die Regel für den Widerstand sieht wie folgt aus:

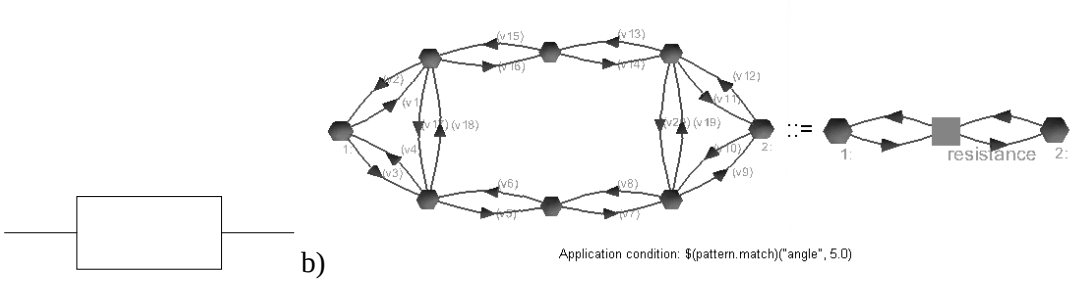


Abbildung 6-8: a) Symbol Widerstand b) Graphregel zur Erkennung eines Widerstandes

Abbildung 6-9 a) zeigt die zu erkennende Schaltung. Sie liegt als Vektorzeichnung vor und wird automatisch in die Graphrepräsentation umgewandelt, da eine manuelle Konvertierung zu aufwendig ist. Um die, bei der Digitalisierung, auftretenden Störungen zu simulieren, kann als zusätzlicher Parameter ein Fehlerwert angegeben werden, der das Attribut Winkel der Kanten zufällig ändert. Abbildung 6-9 b) zeigt den Graphen nach der Graphersetzung. Es wurden alle

Objekte einwandfrei erkannt. Bei den verbleibenden Knoten und Kanten handelt es sich um Verbindungen zwischen Objekten, die von der Graphersetzung nicht erfaßt werden. Diese Elemente müssen entweder manuell entfernt oder durch weitere Graphregeln transformiert werden.

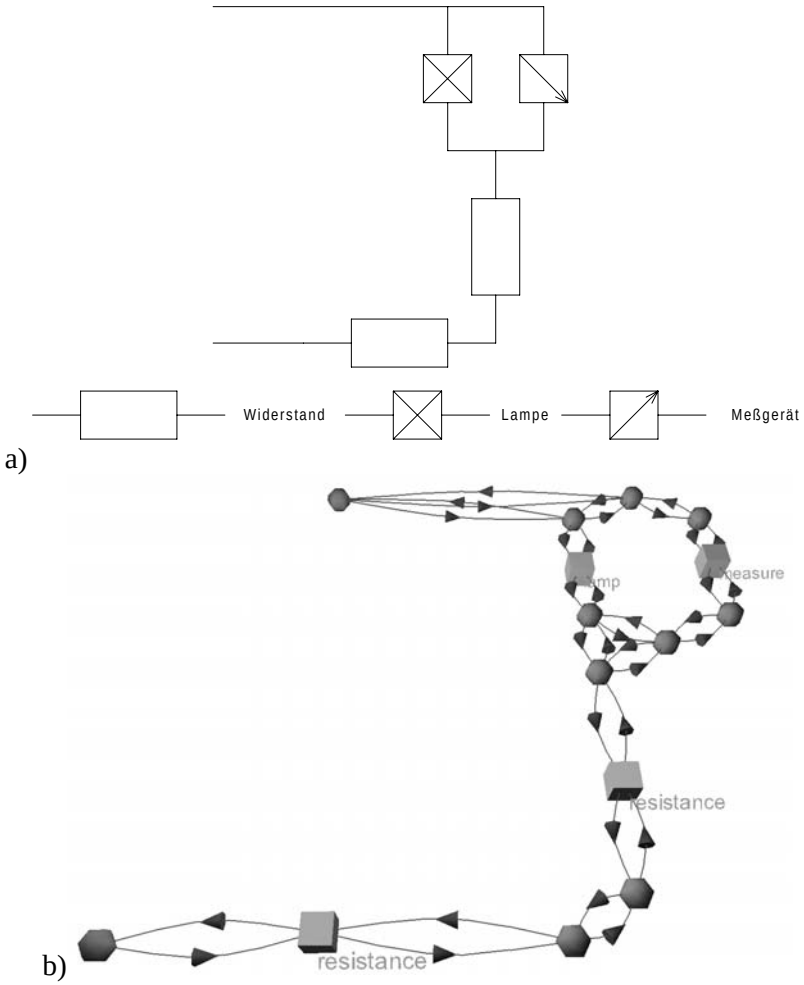


Abbildung 6-9: a) Zu erkennende Schaltung b) Erkannte Schaltung

Abschließend sei erwähnt, daß dieses Konzept auch einen Nachteil hat. So müssen die zu erkennenden Symbole zusammenhängend sein, wodurch man in der Symbolauswahl eingeschränkt wird. Eine Erweiterung auf nicht zusammenhängende Objekte muß den geometrischen Abstand der Knoten mit einbeziehen, was die Komplexität der Graphregeln erhöht.

7 ZUSAMMENFASSUNG UND AUSBLICK

Graphersetzungssysteme können als neues High-Level-Paradigma für Programmiersprachen in die Reihe bestehender eingegliedert werden. Mit dieser Arbeit wurde ein Rahmen für die maschinelle Interpretation des Graphersetzungssystems GRACE erarbeitet. Um die gesamte Mächtigkeit und Intuition von GRACE nutzen zu können, wurde die visuelle Entwicklungsumgebung GRACEland entwickelt. Abschluß bildeten Anwendungen, die den Einsatz zeigten.

Welche Ziele wurden erreicht ?

Ziel war u.a. die Implementierung von GRACE, welche nur wenig von der Theorie abweichen sollte. Haupteigenschaften der Sprache sind: eine regelbasierte Graphersetzung, Unterstützung verschiedener Graphklassen, Ansatzunabhängigkeit, Kontrollbedingungen und Strukturierungs-konzepte. All diese Features finden sich in der Implementierung wieder.

Realisiert wurde eine regelbasierte Graphersetzung dadurch, daß nur Regeln den Zustand des Graphen ändern können. Verschiedene Graphklassen werden durch das Modul xgGraph zugelassen, welches u.a. einen abstrakten Graphbegriff zur Verfügung stellt. Gleichzeitig wird Ansatzunabhängigkeit zugesichert, da der Operator $\Rightarrow$ abstrakt definiert ist und in abgeleiteten Klassen überschrieben werden muß. Somit können verschiedene Ansätze benutzt werden.

Anders sieht dieses bei den Kontrollbedingungen aus, die von der Theorie abweichend implementiert wurden. Die in der Theorie vorliegende abstrakte Definition wurde zugunsten von einheitlichen Strukturen aufgegeben. Alle GRACE-Programme besitzen die gleichen Kontrollbedingungen, wodurch eine größtmögliche Konsistenz zwischen verschiedenen Graphersetzungsansätzen erreicht wurde.

Hervorheben möchte ich die Strukturierungskonzepte von GRACE, die einzigartig sind. Transformationseinheiten erlauben eine Abstraktion von Berechnungen, indem sie diese mit einem Namen versehen. Ein ähnliches Konzept weist auch PROGRES auf, dort wird es als Produktion bezeichnet. AGG weist keine entsprechenden Mechanismen auf. GRACE bietet darüber hinaus Module an, die Transformationseinheiten zu Einheiten zusammenfassen. Im Beispiel aus Kapitel 6.2 zeigte sich der Vorteil der Strukturierung in Module, wodurch die Lesbarkeit um ein Vielfaches erhöht wurde.

Die visuelle Entwicklungsumgebung GRACEland hat die Erwartungen auch erfüllt. Das Editieren von Graphen und Regeln ist effizient. Die Entscheidung für eine dreidimensionale Darstellung hat sich als positiv erwiesen. So konnte dadurch u.a. die Übersichtlichkeit in den Beispielen gewahrt werden. Der einzige Kritikpunkt ist der Programmeditor, der bisher nicht entsprechend der im Kapitel 5.2.3 vorgestellten Visualisierung realisiert werden konnte. Dieser Nachteil konnte durch den HTML Exportfilter ausgeglichen werden.

Erweiterungsvorschläge

Als ungünstig hat sich im Spezifikationsbeispiel das Fehlen eines Operators zur Definition von Nebenläufigkeit erwiesen. So mußten parallele Prozesse sequentialisiert werden, um mit GRACEland definiert werden zu können. Dadurch entfernt sich die Modellierung von der Realität und könnte verfälscht werden.

Bei der Integration von Nebenläufigkeit ist zu beachten, daß es keinen eindeutigen Vorgängerzustand gibt. Damit kann kein Backtracking-Schritt durchgeführt werden (vgl. Kapitel 4.4.1). Bei einem Fail-Stop-Interpreter kann Nebenläufigkeit ohne weiteres integriert werden. Der Befehl könnte z.B. so aussehen: apply parallel process1 | process2.

Beim Ableiten muß durch den Interpreter zugesichert werden, daß bei der Anwendung einer Regel nur ein Prozeß zur Zeit auf den Graphen zugreift. Ein Ausschluß kann dabei auf Graph- oder Morphismusebene geschehen. Auf Graphebene wird der komplette Graph gesperrt, während bei der Morphismusebene nur Knoten und Kanten des Morphismus gesperrt werden.

Ein weiteres Konzept sind parametrisierte Graphregeln. Man kann sie mit Templates von C++ vergleichen. Bei der Definition werden Platzhalter eingesetzt, die später ausgefüllt werden. Dadurch können in einigen Situationen Regeln wiederverwendet werden, so daß keine Regel mehrmals definiert werden muß.

Als letzte Erweiterung sei die Ausnahmebehandlung (engl. exception handling) vorgestellt, mit der der Benutzer in Fehlerfällen den Kontrollfluß steuern könnte. Dieses würde eine vom Interpreter vorgegebene Behandlung ablösen, auf die kein Einfluß genommen werden kann.

Um dieses zu realisieren, müßte der Umfang der Sprache für die Kontrollbedingungen erweitert werden, damit der Benutzer steuernd eingreifen kann. Die Erweiterung könnte z.B. Sprungbefehle, Wiederholungen oder Abbruchbefehle beinhalten. Ferner muß geklärt werden, auf welchen Ebenen eine Ausnahmebehandlung möglich sein soll: Transformationsebene und/oder Befehlsebene.

Darüber hinaus existieren weitere Konzepte, wie z.B. Parameterübergabe, auf die hier nicht näher eingegangen werden. Bei einer Erweiterung sollte die Priorität auf die Integration von Nebenläufigkeit liegen, damit parallele Vorgänge angemessen beschrieben werden können.

Was hat man in der Zukunft zu erwarten ?

In Zukunft werden Graphersetzungssysteme vermehrt eingesetzt werden, um Algorithmen auf Graphen zu beschreiben. Ziel ist dabei nicht die Ersetzung von "herkömmlichen" Systemen, sondern durch Kombination deren Vorteile zu nutzen. Meiner Meinung nach, bieten sie in einigen Bereichen (z.B. Petri-Netze) eine Alternative zu bestehenden Werkzeugen. Die steigende Anzahl von Arbeiten, die Graphersetzungssysteme einsetzen, festigt diese These.

Mit GRACE steht ein System zur Verfügung, das, durch sein offenes Konzept, genügend Potential für neue Anforderungen besitzt. Die Leistungsfähigkeit beruht auf dem abstrakten Graphersetzungsansatz, der beliebige konkrete erfaßt und man somit nicht auf einen festgelegt ist.

Bisher kennen nur wenige das Potential von Graphersetzungssystemen, so daß ein Ziel sein muß, den Bekanntheitsgrad zu erhöhen, um den Sprung von der reinen Forschung in die Praxis zu schaffen.

"Let's do it soon, and let's do it with GRACE(land)!"

(Frei nach [HHK98], Seite 7)

8 ANHANG

8.1 Operationelle Semantik

8.1.1 Notation

In der Definition der Semantik werden folgende Konventionen benutzt:

- Mit c, c_i, ... werden die Komponenten eines Sprachkonstruktes bezeichnet. C ist ein Platzhalter für eine Regel r oder eine Transformationseinheit t. Diese Abkürzung macht es einfacher die Semantik zu definieren, da mit einer Regel mehrere Fälle abgedeckt werden können.
- Mit t, t_i, ... wird eine Transformationseinheit bezeichnet.
- Mit r, r_i, ... wird eine Regel bezeichnet.
- G, G', ... stehen für Graphen.
- Mit spitzen Klammern wird der Code S in Bezug zu einem Kontext G gesetzt: <S, G>. Der Kontext ist der aktuelle Graph, auf denen die Transformationen durchgeführt werden. Der Kontext wird benötigt, um z.B. die Anwendbarkeit von Regeln zu testen.
- Mit $\perp$ ($\perp \notin \mathcal{G}$) wird ein Fehler bezeichnet, der zu einem Halten des Interpreters führt. In der Implementierung wird an dieser Stelle eine Ausnahme (engl. exception) signalisiert, die hier allerdings nicht aufgeführt sind.

8.1.2 Transitionsregeln

Ein zentraler Bestandteil in der Definition ist das Prädikat applicable, mit dem geprüft werden kann, ob eine Regel bzw. eine Transformationeinheit in der aktuellen Konfiguration anwendbar ist oder nicht.

Eine Transformationseinheit t ist im Kontext G anwendbar, wenn $< initial_t, \mathrm{G} >$ zu $< \lambda, \mathrm{G}' >$ ausgewertet wird, d.h. $< initial_t, \mathrm{G} > \xrightarrow{*} < \lambda, \mathrm{G}' >$ gilt. Andernfalls ist t nicht anwendbar.

$$\mathrm{applicable}(t, \mathrm{G}) = \begin{cases} \mathrm{true} & if < initial_t, \mathrm{G} > \xrightarrow{*} < \lambda, \mathrm{G}' > \\ \mathrm{false} & otherwise \end{cases}$$

Eine Regel t ist im Kontext G anwendbar, wenn ein Morphismus m existiert und es gilt:

$$\text{applicable}(r,\text{G}) = \begin{cases} \text{true} & \textit{if } \exists \textit{morphism } m : \neg\text{dangling}(m,r) \wedge \text{appcond}(m,r) \\ \text{false} & \textit{otherwise} \end{cases}$$

$$\text{dangling}(m,r) = \exists e \in \text{G}_\text{E} : e_{source} \in (r_{LHS,V} - r_{INT,V}) \vee e_{target} \in (r_{LHS,V} - r_{INT,V})$$

$$appcond(m,r) = expression(r_{appcond}, m)$$

Für eine Komponente c gilt:

$$\text{applicable}(c,\text{G}) = \begin{cases} \text{applicable}(r,\text{G}) & \textit{if } c \textit{ is a rule } r \\ \text{applicable}(t,\text{G}) & \textit{if } c \textit{ is a transformation unit } t \end{cases}$$

Eine Menge von Komponenten ist reduziert, wenn keine Komponente im aktuellen Kontext angewendet werden kann:

$$\text{reduced}(\{c_1,...,c_n\},\text{G}) = \neg\exists c_i \in \{c_1,...,c_n\} : \text{applicable}(c_i,\text{G})$$

Nun folgen die Transitionsregeln für die verschiedenen Konstrukte der Sprache:

(1) $\langle$*apply once* $c_1 \mid ... \mid c_n$, G$\rangle \rightarrow \langle c_i$, G$\rangle$, applicable($c_i$, G)
(2) $\langle$*apply once* $c_1 \mid ... \mid c_n$, G$\rangle \rightarrow \langle\lambda, \perp\rangle$, reduced({$c_1$, ... , c_n}, G)

(3) $\langle$*apply once* $c_1 < ... < c_n$, G$\rangle \rightarrow \langle c_i$, G$\rangle$, applicable($c_i$, G) $\wedge \neg\exists j<i$: applicable(c_j)
(4) $\langle$*apply once* $c_1 < ... < c_n$, G$\rangle \rightarrow \langle\lambda, \perp\rangle$, reduced({$c_1$, ... , c_n}, G)

(5) $\langle$*apply as long as possible* $c_1 \mid ... \mid c_n$, G$\rangle \rightarrow \langle c_i$*; apply as long as possible* $c_1 \mid ... \mid c_n\rangle$, applicable($c_i$, G)
(6) $\langle$*apply as long as possible* $c_1 \mid ... \mid c_n$, G$\rangle \rightarrow \langle\lambda$, G$\rangle$, reduced({$c_1$, ... , c_n}, G)

(7) $\langle$*apply as long as possible* $c_1 <..< c_n$, G$\rangle \rightarrow \langle c_i$*; apply as long as possible* $c_1 <..< c_n$, G$\rangle$, applicable(c_i, G) $\wedge \neg\exists j<i$: applicable(c_j)
(8) $\langle$*apply as long as possible* $c_1 < ... < c_n$, G$\rangle \rightarrow \langle\lambda$, G$\rangle$, reduced({$c_1$, ... , c_n}, G)

(9) $\langle$*apply k times* $c_1 \mid ... \mid c_n$, G$\rangle \rightarrow \langle c_i$*; apply k-1 times* $c_1 \mid ... \mid c_n$, G$\rangle$, applicable(c_j, G) $\wedge$ k>0
(10) $\langle$*apply 0 times* $c_1 \mid ... \mid c_n$, G$\rangle \rightarrow \langle\lambda$, G$\rangle$
(11) $\langle$*apply k times* $c_1 \mid ... \mid c_n$, G$\rangle \rightarrow \langle\lambda, \perp\rangle$, reduced({$c_1$, ... , c_n}, G) $\wedge$ k>0

(12) $\langle$*apply k times* $c_1 < ... < c_n$, G$\rangle \rightarrow \langle c_i$*; apply k-1 times* $c_1 < ... < c_n$, G$\rangle$, applicable(c_i, G) $\wedge \neg\exists j<i$: applicable(c_j) $\wedge$ k>0
(13) $\langle$*apply 0 times* $c_1 < ... < c_n$, G$\rangle \rightarrow \langle\lambda$, G$\rangle$
(14) $\langle$*apply k times* $c_1 < ... < c_n$, G$\rangle \rightarrow \langle\lambda, \perp\rangle$, reduced({$c_1$, ... , c_n}, G) $\wedge$ k>0

(15) $\langle$*call library.method*, G$\rangle \rightarrow \langle\lambda$, G‘$\rangle$

(16) $\langle S_1;S$, G$\rangle \rightarrow \langle S_2;S$, G‘$\rangle$, $\langle S_1$, G$\rangle \rightarrow \langle S_2$, G‘$\rangle$
(17) $\langle\lambda; S$, G$\rangle \rightarrow \langle S$, G$\rangle$

(18) $\langle r$, G$\rangle \rightarrow \langle\lambda$, G‘$\rangle$, $\text{G} \underset{\text{r}}{\Rightarrow} \text{G}'$

(19) $\langle t$, G$\rangle \rightarrow \langle\lambda$, X$\rangle$, $\langle body_t;terminal_t$, G$\rangle \rightarrow \langle\lambda$, X$\rangle \Leftrightarrow \text{G} \underset{\text{t}}{\Rightarrow} \text{X}$

(20) <*if* condition *then* S *end*, G> → <S, G>, <condition, G> → <λ, G'>
(21) <*if* condition *then* S *end*, G> → <λ, G>, <condition, G> → <λ, ⊥>
(22) <*if* condition *then* S1 *else* S2 *end*, G> → <S1, G>, <condition, G> → <λ, G'>
(23) <*if* condition *then* S1 *else* S2 *end*, G> → <S2, G>, <condition, G> → <λ, ⊥>

(24) <*while* condition *do* S *end*, G> → <S, G>, <condition, G> → <λ, G'>
(25) <*while* condition *do* S *end*, G> → <λ, G>, <condition, G> → <λ, ⊥>

8.1.3 Graphausdrücke

Die Graphausdrücke beziehen sich auf die Klasse der gerichteten Graphen ("DirectedGraph"), die in der dynamischen Bibliothek DirectedGraph.gc implementiert sind.

(1) <*vertices labeled* $l_1, \ldots, l_n$, G> → <λ, G>, $\forall v \in V_G$: label(v)∈ $\{l_1, \ldots, l_n\}$
(2) <*vertices labeled* $l_1, \ldots, l_n$, G> → <λ, ⊥>, $\exists v \in V_G$: label(v)∉ $\{l_1, \ldots, l_n\}$

(3) <*vertices not labeled* $l_1, \ldots, l_n$, G> → <λ, G>, $\forall v \in V_G$: label(v)∉ $\{l_1, \ldots, l_n\}$
(4) <*vertices not labeled* $l_1, \ldots, l_n$, G> → <λ, ⊥>, $\exists v \in V_G$: label(v)∈ $\{l_1, \ldots, l_n\}$

(5) <*vertices unlabeled*, G> → <λ, G>, $\forall v \in V_G$: label(v) = ε[12]
(6) <*vertices unlabeled*, G> → <λ, ⊥>, $\exists v \in V_G$: label(v) ≠ ε

(7) <*edges labeled* $l_1, \ldots, l_n$, G> → <λ, G>, $\forall e \in E_G$: label(e)∈ $\{l_1, \ldots, l_n\}$
(8) <*edges labeled* $l_1, \ldots, l_n$, G> → <λ, ⊥>, $\exists e \in E_G$: label(e)∉ $\{l_1, \ldots, l_n\}$

(9) <*edges not labeled* $l_1, \ldots, l_n$, G> → <λ, G>, $\forall e \in E_G$: label(e)∉ $\{l_1, \ldots, l_n\}$
(10) <*edges not labeled* $l_1, \ldots, l_n$, G> → <λ, ⊥>, $\exists e \in E_G$: label(e)∈ $\{l_1, \ldots, l_n\}$

(11) <*edges unlabeled*, G> → <λ, G>, $\forall e \in E_G$: label(e) = ε
(12) <*edges unlabeled*, G> → <λ, ⊥>, $\exists e \in E_G$: label(e) ≠ ε

(13) <*template G'*, G> → <λ, G>, G = G'
(14) <*template G'*, G> → <λ, ⊥>, G ≠ G'

(15) <t, G> → <λ, X>, <bodyt; terminalt, G> →* <λ, X>

[12] Epsilon (ε) steht für eine leere Markierung.

8.2 Dateiformat 3rd

Das Dateiformat wurde speziell für das VR Toolkit VLE entwickelt, um Objekte und Szenen zu beschreiben.

Am Anfang der Datei befindet sich eine Kennung, die das Format identifiziert. Danach können Materialdefinitionen folgen, in der ein Material mit einem Namen versehen wird. Bei der Definition von Polygonen können sie über die Namen referenziert werden. Der Vorteil ist, daß nur eine gemeinsam benutzte Instanz erzeugt wird und damit der Speicherverbrauch gering bleibt. Wird hingegen das Material direkt beim Polygon angegeben, so wird jedesmal eine neue Instanz erzeugt. Es werden zwei Arten vom Materialien unterstützt: Farbenmaterialien und Texturen (im Portable Pixmap "PPM" Format).

Die Geometrie der Objekte wird durch eine Knoten- und eine Flächenliste beschrieben. Die Knotenliste definiert die Eckpunkte durch Angabe der X-, Y- und Z-Koordinaten (vgl. Abbildung 5-7, Seite 59). Jeder Knoten besitzt eine Indexnummer (startend bei 0), die ihm seine Position in der Liste zuordnet. In der Flächenliste werden die Polygone des Objektes beschrieben. Ein Polygon wird durch Angabe der Anzahl der Eckpunkte und deren Indexnummern definiert (vgl. Beispiel). Dabei wird vorausgesetzt, daß die Knoten entgegen den Uhrzeigersinn angegeben werden. Sind die Punkte im Uhrzeigersinn angegeben, so muß dieses durch "cw" (clock wise) gekennzeichnet sein. Abgeschlossen wird die Polygondefinition durch die Angabe eines Materials.

Objekte werden mit einem Namen versehen, der als Referenz für den Aufbau einer Hierarchie dient. Darüber hinaus kann die Position, Orientierung, Skalierung und der Pivot-Punkt[13] angegeben werden. Für die Wurzel der Hierarchie hat allerdings nur der Pivot-Punkt eine Bedeutung, da die anderen Parameter überschrieben werden. Nach dem Laden werden die Geometrien so transformiert, daß der Mittelpunkt des Objektes zum neuen Ursprung wird. Positionsangaben beziehen sich damit immer auf den Mittelpunkt des Objektes.

Die Grammatik der Dateien lautet:

```
TRD       ::= "3rd v1" [{MTDEF}] OBJECT {OBJECT}.
OBJECT    ::= IDENTIFIER ["child" IDENTIFIER]
              "{" PLACE VERTICES FACES "}".
PLACE     ::= ["position" POINT] ["orientation" POINT]
              ["scale" POINT] ["pivot" POINT].
VERTICES  ::= "vertices" NUMBER {VERTEX}.
VERTEX    ::= NUMBER NUMBER NUMBER.

FACES     ::= "faces" NUMBER {FACE}.
FACE      ::= NUMBER {NUMBER} ["cw"] MATERIAL.
MATERIAL  ::= ("color" RGB) | ("texture" STRING)
             |("material" IDENTIFIER).
```

[13] Der Pivot-Punkt definiert den Rotationspunkt des Objektes in Objektkoordinaten. Normalerweise liegt dieser im Mittelpunkt des Objektes und hat die Position (0,0,0).

```
MTDEF     ::= "material" IDENTIFIER "{" (MTSIMPLE | MTTEXTURE) "}".
MTSIMPLE  ::= COLOR | (AMBIENT DIFFUSE SPECULAR EMISSION SHININESS).
COLOR     ::= "color" RGB.
AMBIENT   ::= "ambient" RGB.
DIFFUSE   ::= "diffuse" RGB.
SPECULAR  ::= "specular" RGB.
EMISSION  ::= "emission" RGB.
SHININESS ::= "shininess" NUMBER.
MTTEXTURE ::= "texture" STRING.
RGB       ::= NUMBER NUMBER NUMBER.
```

Beispiel: Definition einer Pyramide

```
* Formatidentifizierung
3rd v1

* Definiere ein grünes Material
material green { color 0.0 0.75 0.0 }

* Beschreibung des Objekte 'Pyramide'
Pyramide
{
  * Knotenliste des Objektes
  vertices 5
    -0.5  0.0  0.5 * Grundfläche
     0.5  0.0  0.5 *    --"--
     0.5  0.0 -0.5 *    --"--
    -0.5  0.0 -0.5 *    --"--
     0.0  1.0  0.0 * Spitze der Pyramide

  * Polygonliste des Objektes
  faces 5
    4  3 2 1 0 material green  * Grundfläche
    3  0 1 4   material green  * Seitenfläche
    3  1 2 4   material green  *    --"--
    3  2 3 4   material green  *    --"--
    3  3 0 4   material green  *    --"--
}
```

8.3 Abbildungsverzeichnis

8.4 Definitionen

8.5 Literaturverzeichnis

[AB92] S. Aukstakalnis and D. Blatner. *Sillicon Mirage. The Art and Science of Virtual Reality*. Peachpit Press, 1992

[AEH+96] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr and G. Taentzer. *Graph transformation for specification and programming*. Technical Report 7/96, Universität Bremen, 1996

[AO94] K. R. Apt and E.-R. Olderog. *Programmverifikation - Sequentielle, parallele und verteilte Programme*. Springer-Verlag, Berlin, 1994

[Ass95] U. Assman. *Generierung von Programmoptimierungen mit Graphersetzungs-systemen*. Dissertation, Universität Karlsruhe, 1995

[Bar97] R. Bardohl. *Application of Graph Transformation to Visual Languages - State of the Art and Further Ideas* -. Technical Report 97-10, Technical University Berlin, 1997

[Bau90] B. Baumgarten. *Petri-Netze. Grundlagen und Anwendungen*. BI-Wissenschaftsverlag, Mannheim, 1990

[BB93] H. Bässmann und P. W. Besslich. *Bildverarbeitung Ad Oculos*. 2. Auflage, Springer-Verlag, Berlin, 1993

[BET+94] G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis. *Algorithms for Drawing Graphs: an Annotated Bibliography*. In Computational Geometry: Theory and Applications, vol. 4, no. 5, 1994, pp. 235-282

[BFG96] D. Blostein, H. Fahmy and A. Grbavec. *Issues in the Pracitcal Use of Graph Rewriting*. Lecture Notes in Computer Science, Vol. 1073, Springer-Verlag, 1996, pp. 38-55

[BH98] D. Blostein and L. Haken. *Using Diagram Generation Software to Improve Recognition: A Case Study of Music Recognition*. Submitted for publication. http://www.qucis.queensu.ca/~blostein/recgen.ps

[Boe95] E. J.W. Boers. *Using L-Systems as Graph Grammar: G2L-Systems*. Technical Report 95-30, Dept. of Computer Science, Leiden University, October 1995

[Boo94] G. Booch. *Object-oriented analysis and design with applications*. Addison-Wesley Reading, Massachusetts, 1994

[Bro95] F. P. Brooks, Jr.. *The mytical man-month: essays on software engineering*. Anniversary ed., Addison Wesley, Reading, Massachusetts, 1995

[BS97] D. Blostein and A. Schürr. *Computing with Graphs and Graph Rewriting*. Aachener Informatik-Berichte AIB 97-8, RWTH Aachen, Fachgruppe Informatik,

1997

[Bur97] R. Burkhardt. *UML-Unified Modelling Language, Objektorientierte Modellierung für die Praxis*. Addison-Wesley, Bonn, 1997

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe. *Algebraic approaches to graph transformation. Part I: Basic concepts and double pushout approach*. In [Roz97], pp. 163-245

[CSH+92] D. B. Conner, S. S. Snibbe, K. P. Herndon, D. C. Robbins, R. C. Zeleznik and A. van Dam. *Three-Dimensional Widgets*. Computer Graphics (Proceedings of the 1992 Symposium on Interactive 3D Graphics), 25(2), ACM SIGGRAPH, March, 1992, pp. 183-188

[DK98] F. Drewes and H.-J. Kreowski. *Picture Generation*. European School on Graph Transformation, Bremen, March 2-7, 1998

[DOL94] A. Davison, Paul Otto and David Lau-Kee. *Visual Programming*. In [DV94], 1994, pp. 27-41

[Dör95] H. Dörr. *Efficient graph rewriting and ist implementation*. Lecture Notes in Computer Science Nr. 922, Springer-Verlag, Berlin, 1995

[Dud93] H. Engesser (Hrsg.). *Duden Informatik: ein Sachlexikon für Studium und Praxis*. 2. Auflage, Dudenverlag, Mannheim, 1993

[DV94] L. Mac Donals and J. Vice (eds.). *Interacting with Virtual Environments*. John Wiley & Sons, Chichester, 1994

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini. *Algebraic approaches to graph transformation. Part II: Single pushout approach and comparison with double pushout approach*. In [Roz97] pp. 247-312

[EHT+97] G. Engels, R. Heckel, G. Taentzer and H. Ehrig. *A View-Oriented Approach to System Modelling based on Graph Transformation*. To appear in Proceedings of ESEC/FSE'97, Zürich, 1997
http://www.cs.tu-berlin.de/~gabi/gEHTE97.ps.gz

[ESP97] M. van Eekelen, S. Smetsers and R. Plasmeijer. *Graph Rewriting Semantics for Functional Programming Languages*. In Proc. of CSL '96, Fifth Annual conference of the European Association for Computer Science Logic (EACSL), Utrecht, Dirk van Dalen Ed., Springer-Verlag, LNCS 1258, 1997, pp. 106-128

[ET96] H. Ehrig and G. Taentzer. *Computing by Graph Transformation, A Survey and Annotated Bibliography*. Technical Report, TU-Berlin, No. 96-21

[Fau97] M. Faust. *Virtual Reality Träume mit OpenGL*. In Linux Magazin Nr. 12, 1997, pp. 11-17

[Fau98] M. Faust. *GRACEland Homepage.*
http://www.informatik.uni-bremen.de/grp/ag-ti/GRACEland

[FDF+96] J. D. Foley, A. van Dam, S. K. Feiner and J. F. Hughes. *Computer Graphics, Principles and Practice*. Second Edition in C, Addison-Wesley, Reading, Massachusetts, 1996

[Fel92] W. D. Fellner. *Computer-Grafik*. BI-Wissenschaftsverlag, Mannheim, 1992

[Fis95] P. A. Fishwick. *Simulation Model Design and Execution, Building Digital Worlds*. Prentice-Hall, New Jersey, 1995, pp. 216-231

[FLM+97] A. S. Forsberg, J. J. LaViola, Jr., L. Markosian and R. C. Zeleznik. *Projects in VR, Seamless Interaction in Virtual Reality*. In IEEE Computer Graphics and Applications, November/Dezember 1997, pp. 6-9

[FLM94] A. Frick, A. Ludwig and H. Mehldau. *A Fast Adaptive Layout Algorithm for Undirected Graphs*. In Proceedings of Graph Drawing'94, LNCS 894, Springer-Verlag, 1995

[FR91] T. M. J. Fruchterman and E. M. Reingold. *Graph Drawing by Force-directed Placement*. In Software–Practice and Experience, Vol 21(11), November 1991, pp. 1129-1164

[Fra97] G. Franck. *Graph Visualizer 3D*. http://www.omg.unb.ca/hci/projects/gv3d, October 1997

[FSS97] C. Faisstnauer, D. Schmalstieg and Zsolt Szalavári. *Device-Independent Navigation and Interaction in Virtual Environments*. Technical Report 186-2-97-15, Technische Universität Wien, 1997

[FW94] G. Franck and C. Ware. *Representing Nodes and Arcs in 3D Networks*. In Conference Proceedings of IEEE Conference on Visual Languages, 1994, pp. 189-190

[GB95] A. Grbavec and D. Blostein. *Mathematics Recognition Using Graph Rewriting*. Third International Conference on Document Analysis and Recogniton, Montreal, Canada, August 1995, pp. 417-421

[Gog96] M. Gogolla. *Skript zur Vorlesung Datenbanksysteme I*. Universität Bremen, 1996

[Gog97] M. Gogolla. *Skript zur Vorlesung Datenbanksysteme II*. Universität Bremen, 1997

[GS96] M. Goedicke and B. E. Sucrow. *Towards a Formal Specification Method for Graphical User Interfaces using Modularized Graph Grammars*. In: Proceedings of the 8th International Workshop on Software Specification and Design; IEEE Computer Society Press, March, 22-23; Schloß Velen, Germany; 1996, pp. 56-65

[GTS98] M. Goedicke, P. Tröpfner und B. Sucrow. *Hierarchical Specification of Graphical User Interfaces using a Graph Grammar Approach*. Accepted for Publication in Proceedings of the Third World Conference on Integrated Design & Process Technology, Berlin, Germany, June, 1998
http://

[Han97] C. Hand. *A Survey of 3D Interaction Techniques*. In Computer Graphics Forum, Volume 16 Nr. 5, 1997, pp. 269-281

[HHK98] R. Heckel, B. Hoffmann and S. Kuske. *Simple Modules for GRACE*. Internal Paper, January 1998
http://www.informatik.uni-bremen.de/grp/ag-ti/~kuske

[HK92] A. Habel und H.-J. Kreowski. *Formale Graph-Sprachen*. Skript zur Veranstaltung

Formale Graph-Sprachen, Universität Bremen, 1992

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985

[HP79] M. C. B. Hennessy and G. D. Plotkin. *Full abstraction for a simple programming language*. In Proceedings of Mathematical Foundations of Computer Science, New York, 1979, pp. 108-120

[Hub98] G. Hubona. *Perceptual and Cognitive Issues for the Design of 3D Visual Interfaces and for Virtual Environment (VE) Technology*. http://groucho.gsfc.nasa.gov/eve/HTML/cogdoc.html, Juni 1998

[HZR+92] K. P. Herndon, R. C. Zeleznik, D. C. Robbins, D. B. Conner, S. S. Snibbe and A. van Dam. *Interactive Shadows*. Proceedings of UIST '92, ACM SIGGRAPH, November, 1992, pp. 1-6

[Jun94] D. Jungnickel. *Graphen, Netzwerke und Algorithmen*. BI Wissenschaftsverlag Mannheim, Leipzig, Wien, Zürich, 1994

[KK98] H.-J. Kreowski and S. Kuske. *Graph Transformation Units with Interleaving Semantics*. European School on Graph Transformation, Bremen, March 2-7, 1998 http://www.informatik.uni-bremen.de/grp/ag-ti/~kuske/papers/SemAsp.ps.gz

[KKS97] H.-J. Kreowski, S. Kuske and A. Schürr. *Nested graph transformation units*. In International Journal on Software Engineering and Knowledge Engineering 7(4), 1997, pp. 479-502

[Kla94] C. Klauck. *Eine Graphgrammatik zur Repräsentation und Erkennung von Features in CAD/CAM*. Dissertation, infix-Verlag, DISKI Nr.66, 1994

[Kla95] C. Klauck. *Graph-Grammar-Based Object Recognition for Image Retrieval*. In Proceedings of the 2nd Asian Conference on Computer Vision (ACCV'95), Singapore, December 1995

[Kon94] G. Kondrak. *A Theoretical Evaluation of Selected Backtracking Algorithms*. Technical Report 94-10, Dept. of Computing Science, University of Alberta, 1994

[Kum92] V. Kumar. *Algorithms for Constraint Satisfaction Problems: A Survey*. In AI Magazine 13(1), 1992, pp. 32-44

[Kus95] S. Kuske. *Semantic Aspects of the Graph and Rule Centered Language GRACE*. In Francesc Rossello and Gabriel Valiente (eds.). Proceedings Colloquium on Graph Transformation and its Application in Computer Science, Technical Report B-19, Universitat de les Illes Balears, 1995, pp. 63-69

[KY93] H. Koike and H. Yoshihara. *Fractal Approaches for Visualizing Huge Hierarchies*. In Proceedings of the 1993 IEEE Symposium on Visual Languages, 1993, pp. 55-60

[Lou94] K. C. Louden. *Programmiersprachen, Grundlagen, Konzepte, Entwurf*. Thomson

Publishing, Bonn, 1994

[LS93] I. Lemke, G. Sander. *Visualization of Compiler Graphs*. Design report D 3.12.1-1, USAAR-1025-visual, ESPRIT Project #5399 Compare, Universität des Saarlandes, FB 14 Informatik, 1993

[MB95] B. T. Messmer and H. Bunke. *Subgraph Isomorphism in Polynomial Time*. Technical Report IAM-95-003, Research Group on Computer Vision and Artificial Intelligence, University of Bern, 1995

[Men95] R. Mencel. *A Graph-Theoretic Approach to Surface Reconstruction*. Forschungsbericht Nr. 568 / 1995, Universität Dortmund, März 1995

[Mes95] B.T. Messmer. *PhD Thesis*. Rearach Group on Computer Vision and Artificial Intelligence, University of Bern, 1995
ftp://iamftp.unibe.ch/pub/Papers/MessmerPhdThesis.ps.gz

[MG96] T. Mazuryk and M. Gervautz. *Virtual Reality. History, Applications, Technology and Future.* Technical Report TR-186-2-96-06, Institute of Computer Graphics, Visualisation and Animation Group, Vienna Institute of Technology, 1996

[Nor94] S. C. North. *Applications of Graph Visualisation*. AT&T Bell Labs, 1994
http://www.research.att.com/sw/tools/graphviz/Gl94.ps.gz

[NPR+96] U. Nowak, U. Pöhle, R. Roitzsch and B. E. Sucrow. *Formal Specification of the ZIB-GUI Using Graph Grammars*. In W. Mackens and S. Rumpf (eds.), Software Engineering in Scientific Computing, Vieweg, July 1996, pp. 290-296

[Oli95] D. Olivastro. *Das chinesische Dreieck: die kniffligsten mathematischen Rätsel aus 10000 Jahren*. Droemer Knauer, München, 1995

[Plu98] D. Plump. *Termination of graph rewriting is undecidable*. In Fundamenta Informaticae XX, IOS Press, 1998, pp. 1-8

[Rei92] W. Reisig. *A primer in Petri Net Design*. Springer-Verlag Berlin, 1992

[Rep95] A. Repenning. *Bending the Rules: Steps Toward Semantically Enriched Graphical Rewrite Rules*. In Proceedings of the 11th International Symposium on Visual Languages, VL95, Darmstadt, Germany, 1995

[Roz97] G. Rozenberg (Ed.). *Handbook of graph grammars and computing by graph transformation, Volume 1 - Foundations*. World Scientific, Singapore, 1997

[Rud97] M. Rudolf. *Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation*. Diplomarbeit, Technische Universität Berlin, Dezember 1997

[San94] G. Sander: *Graph Layout through the VCG Tool*. Technical Report A03-94, Universität des Saarlandes, FB 14 Informatik, 1994

[SB92] M. Sarkar and M. H. Brown. *Graphical Fisheye Views of Graphs*. SRC Research Report 84a, Digital Systems Research Center, 1992

[Sch88] A. Schürr. *Modellierung und Simulation komplexer Systeme mit PROGRESS*. In W. Ameling (Hrsg.): Proc. 5. Symp. Simulationstechnik, Aachen, Germany, Sept.

1988, Informatik-Fachberichte 179, Springer-Verlag, Berlin, 1988, pp. 84-91

[Sch94a] A. Schürr: Rapid Programming with Graph Rewrite Rules, in: USENIX Symp. Proc. Very High Level Languages (VHLL), Santa Fee, New Mexico, Okt. 1994, USENIX Association, 1994, pp. 83-100

[Sch94b] A. Schürr. *PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems*. Technical Report AIB 94-11, RWTH Aachen, Germany, 1994

[Sch97] A. Schulz. *Graphenanalyse hydraulischer Schaltkreise zur Erkennung von hydraulischen Achsen und deren Kopplung*. Diplomarbeit, Universität Paderborn, September 1997

[Sch98] A. Schürr. *Pattern matcher for simple mathematical expressions*. PROGRES Spezifikation, 1998
http://www-i3.informatik.rwth-aachen.de/private/andy/GraphRewriting/index.html

[Suc96] B. E. Sucrow. *Formal Specification of Human-Computer Interaction by Graph Grammars under Consideration of Information Resources*. In: Proceedings of The Second World Conference on Integrated Design & Process Technology; December 1-4, Austin, Texas, 1996, pp. 194-201

[SW95] A. Schürr, A. Winter and A. Zündorf. *Spezifikation und Prototyping graphenbasierter Systeme*. GI-Fachtagung Softwaretechnik, Braunschweig, 1995

[SZH96] D. Stalling, M. Zöckler and H.-C. Hege. *Fast Display of Illuminated Field Lines*. Report SC 96-58, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1996

[Tae96] G. Taentzer. *Modeling Dynamic Distributed Object Structures by Graph Transformation*. In on-line magazine Objects Currents, Vol. 1, no. 12, 1996
http://www.cs.tu-berlin.de/~gabi/oc9701.f.taentzer.html

[UML97a] *Unified Modelling Language - UML Notation Guide*. Version 1.1, September 1997
ftp://ftp.omg.org/pub/docs/ad/97-08-05.pdf

[UML97b] *Unified Modelling Language - UML Semantics*. Version 1.1, September 1997
ftp://ftp.omg.org/pub/docs/ad/97-08-04.pdf

[Wat93] A. Watt. *3D Computer Graphics*. Addison-Wesley Workingham, England, 1993

[WF94] C. Ware and G. Franck. *Viewing a Graph in a Virtual Reality Display is Three Times as Good as a 2D Diagramm*. In Proceedings of IEEE Conference on Visual Languages, 1994, pp. 182-183

[WHF93] C. Ware, D. Hui and G. Franck. *Visualizing Object Oriented Software in Three Dimensions*. In Conference Proceedings of CASCON'93, 1993, pp. 612-620

[Wir86] N. Wirth. *Compilerbau*. Teubner-Studienbücher : Informatik, 1986

[WS97] A. Winter and A. Schürr. *Modules and Updatable Graph Views for PROgrammed Graph Rewriting Systems*. Aachener Informatik Berichte 97-3, RWTH Aachen, Fachgruppe Informatik, 1997

[ZPR+93] R. C. Zeleznik, K. P. Herndon, D. C. Robbins, N. Huang, T. Meyer, N. Parker and J. F. Hughes. *An Interactive 3D Toolkit for Constructing 3D Widgets*. *In Computer Graphics Proceedings*. Annual Conference Series, 1993, pp. 81-84

[Zün93] A. Zündorf. *A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES*. Technical Report AIB 93-5, RWTH Aachen, Germany, 1993