# Über generisches visuelles Programmieren *

Berthold Hoffmann[1] und Mark Minas[2]

[1] Fachbereich Mathematik/Informatik, Universität Bremen
Postfach 33 04 40, 28334 Bremen
hof@informatik.uni-bremen.de
[2] Lehrstuhl für Programmiersprachen, Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen
minas@informatik.uni-erlangen.de

**Zusammenfassung** Graphen sind nützlich für die abstrakte Darstellung visueller Spezifikationen. In diesem Papier umreißen wir eine Sprache, die geschachtelte Graphen unterstützt sowie regelbasiert und objektorientiert ist. Insbesondere gehen wir auf die *Generizität* der Sprache ein. Dadurch lassen sich Graphen als verschiedene Arten von Diagrammen visualisieren; Programme können so Notationen für die Ein-Ausgabe verwenden, wie sie im jeweiligen Anwendungsgebiet gebräuchlich sind. Die Sprache ist selbst visuell und benutzt den Diagramm-Editor-Generator DiaGen um Generizität zu unterstützen, sowie zur Implementierung ihrer Benutzungsschnittstelle und zur Typüberprüfung.

## 1 Einleitung

Graphen sind nützlich für die abstrakte Darstellung visueller Spezifikationen. Deshalb ist die regelbasierte Transformation von Graphen [20, 7] ein interessantes Berechnungsmodell für visuelle Programmiersprachen. Existierende Graphtransformationssprachen wie Progres [21] waren in dieser Anwendung jedoch nicht so erfolgreich wie erwartet werden konnte. Wir glauben, daß dies hauptsächlich zwei Gründe hat: Zum einen wird die *Schachtelung* von Graphen (analog derer von *Objekten*) und damit auch die *objektorientierte Programmierung* nicht unterstützt; außerdem ist es nur schwer möglich, die "Standard"-Notation von Graphen an die visuellen Notationen bestimmter Anwendungsgebiete *anzupassen*.

Für das visuelle Programmieren sind schon einige objektorientierte Sprachen entwickelt worden [2], insbesondere *Prograph* [4]. Aber auch alle visuellen Sprachen, die wir kennen, sind mehr oder weniger an eine Diagrammnotation gebunden.

In diesem Papier geht es um eine Programmiersprache, die auf Graphtransformation basiert, geschachtelte Graphen und objektorientiertes Programmieren unterstützt und visuell (*graph*-isch) und regelbasiert ist. Wir diskutieren insbesondere die *Generizität* der Sprache, dank derer die intern verarbeiteten Graphen

* Diese Arbeit wurde von der ESPRIT-Arbeitsgruppe *Applications of Graph Transformation* (Appligraph) unterstützt.

in einer Vielfalt von Diagrammnotationen visualisiert werden können. In dieser Sprache implementierte Systeme können dadurch unmittelbar die Darstellungsweisen nutzen, die im jeweiligen Anwendungsgebiet gebräuchlich sind. Die konzipierte Programmier- und Ausführungsumgebung gestattet es dem Programmierer und dem späteren Anwender, anwendungsspezifische Diagramme sowohl in der Eingabe als auch der Ausgabe des Systems zu verwenden. Damit unterscheidet sich die Programmiersprache mit ihrer zugehörigen Umgebung deutlich von anderen Systemen (wie Smalltalk- oder Prolog-Umgebungen), die zwar auch graphische Ein- und Ausgaben erlauben, die aber getrennt von den intern verwendeten Datenstrukturen zu programmieren sind.

Das Papier gliedert sich wie folgt: In Abschnitt 2 umreißen wir exemplarisch einige grundlegende Konzepte einer graph- und regelbasierten Programmiersprache; auf die anwendungsspezifische Visualisierungen der verwendeten Graphen wird dabei noch nicht eingegangen. Dies ist Thema von Abschnitt 3, das rekapituliert, wie Graphen als Datenstruktur bei der Analyse von Diagrammen benutzt werden können, und das erklärt, wie dies ausgenutzt wird, um das Programmieren mit Graphen *generisch* an viele verschiedene Diagrammnotationen anzupassen. In Abschnitt 4 beschreiben wir, wie so eine generische visuelle Programmiersprache implementiert werden kann. Abschließend machen wir in Abschnitt 5 einige Bemerkungen zu verwandten und zukünftigen Arbeiten.

Schon aus Platzgründen kann unsere Darstellung nur informell sein. Wir erklären die Konzepte an einem Beispiel, in dem es um die Darstellung von Listen als Graphen geht.

## 2 Programmieren mit Graphen und Regeln

In diesem Abschnitt beschreiben wir exemplarisch, wie auf der Basis eines Modells *geschachtelter Graphen* regelbasiert und objektorientiert programmiert werden kann. Eine ausführlichere Darstellung der hier umrissenen Programmierkonzepte findet sich in [10].

### 2.1 Graphen

*Graphen* repräsentieren Relationen zwischen Größen (*entities*) als *Kanten* zwischen *Knoten*. Üblicherweise verbinden Kanten zwei Knoten eines Graphen und repräsentieren zweistellige Relationen. Wir erlauben dagegen, daß Kanten *beliebig* viele Knoten verbinden, und unterscheiden verschiedene *Kantentypen* durch Markierungen, so dass in einem Graphen verschiedene Relationen beliebiger Stelligkeit repräsentiert werden können. Außerdem zeichnen wir eine Folge von Knoten als *Punkte* aus, an denen Graphen miteinander verknüpft werden können. Solche Graphen sind in der Literatur als *punktierte Hypergraphen* bekannt [5].

Üblicherweise sind Graphen flach: Sie können keine *geschachtelten* Graphen enthalten. Wir dagegen zeichnen eine Teilmenge der Kanten in einem Graphen $H$ als *Rahmen* (*frames*) aus: Jeder Rahmen $f$ *enthält* einen Untergraphen $H_f$, der

selbst geschachtelt sein kann. Solche Graphen werden in [6] *hierarchisch* genannt und sind sind für das objektorientierte Programmieren mit Graphen unerläßlich.

Abbildung 1 zeigt, wie Listen als Graphen dargestellt werden können: Ein

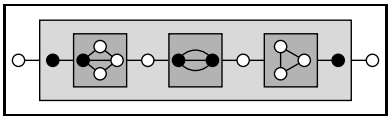

**Abbildung 1.** Ein Listenrahmen mit drei geschachtelten Elementrahmen

zweistelliger *Listenrahmen* enthält eine Kette von ebenfalls zweistelligen Elementrahmen. Jeder *Elementrahmen* enthält einen *Elementgraphen.*

Knoten werden als Kreise gezeichnet und sind schwarz, wenn es sich um Punkte handelt. Kanten werden als Kästen um ihre Markierung gezeichnet, und mit Knoten durch Linien verbunden. (Die Kästen einfacher binärer Kanten "verschwinden", so daß sie als Linien zwischen zwei Knoten gezeichnet werden.) Rahmen sind Kästen (wie normale Kanten auch), in die der Inhalt hineingezeichnet wird. Wir unterscheiden Listen- und Elementrahmen durch ihren Grauton und lassen dafür ihre Markierungen weg. Der Graph in Abbildung 1 enthält einen Listenrahmen, der drei Elementrahmen enthält, deren Inhalt in diesem Beispiel keine besondere Bedeutung hat.

Kanten über Rahmengrenzen hinweg, wie andere Hierarchiebegriffe [8] sie erlauben, sind in diesen Graphen *verboten*, damit sie modular transformiert werden können. Die Verbindungen eines Rahmens, und die Punkte seines Inhalts können jedoch eine *indirekte Beziehung* zwischen dem Inhalt und der Umgebung des Rahmens ausdrücken. Der Listenrahmen in Abbildung 1 drückt so eine indirekte Beziehung zwischen den Punkten des in ihm enthaltenen Listengraphen und seinen Verbindungen aus.

Die hier betrachteten Graphen sind *attributiert*: Ihre Knoten und Kanten können mit Werten assoziiert werden, die mittels semantischer Funktionen berechnet werden. Attribute sind größtenteils einfache Werte wie Zahlen (von Bildschirmkoordinaten u. dgl.). Attribute werden jedoch der Einfachheit zuliebe in diesem Papier nicht weiter behandelt und tauchen daher auch nicht in den Beispielen auf.

### 2.2 Abstraktion

Die Abbildungen 2 und 3 zeigen *Prädikate*, die Graphen transformieren. Die Definition des Prädikats remove in Abbildung 2 enthält zwei Graphen: Der linke ist ein *Muster* (*pattern*), das in dem zu transformierenden *Wirtsgraphen* gesucht wird (*matching*). Dabei können die *Variablen* $X$ und $L$ an beliebige Untergraphen gebunden werden. Das Muster mit der remove-Kante und dem Listenrahmen wird an den Punkten aus dem Wirtsgraphen *herausgeschnitten*, und eine

Kopie des rechten Graphen dafür *eingeklebt*, wobei dessen Variable $L$ durch den an sie gebundenen Untergraphen *instanziiert* wird.

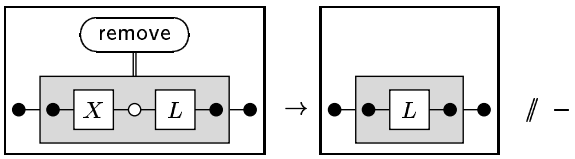


**Abbildung 2.** Das Prädikat remove

Ein Prädikat kann durch mehrere Ersetzungsregeln definiert werden, die sich nicht ausschließen müssen; für remove reicht eine. Falls diese Regel nicht angewendet werden kann, legt die *otherwise*-Definition "// −" fest, daß das Prädikat *scheitert* und *Backtracking* eingeleitet werden soll. Das Prädikat remove löscht also den ersten Elementgraphen $X$ in einem Listenrahmen und scheitert, wenn es auf den leeren Listengraphen angewendet wird.

Im allgemeinen können die rechten Seiten von Regeln in Prädikaten weitere (oval gezeichnete) Prädikatkanten enthalten, mit denen weitere Prädikate aufgerufen werden. Die *Auswertung* eines Prädikats verlangt es daher im allgemeinen, zunächst eine seiner Regeln anzuwenden, und dann rekursiv alle dadurch eingefügten Prädikataufrufe auszuwerten.

Die Verbindungen einer Prädikatkante geben *Parameter* an; sie sind als Doppellinien dargestellt. Ein Prädikatparameter kann einfach ein *Knoten*, aber auch eine *Kante* sein. Insbesondere kann diese Kante ein Rahmen sein, der einen *Graphparameter* angibt (wie in diesem Beispiel), oder ein *Prädikatparameter*, wie im nächsten Beispiel.

### 2.3 Steuerung

Abbildung 3 zeigt den Kombinator normalize, der auf einen mit der Variablen $T$ bezeichneten *Prädikatparameter* angewendet wird und diesen solange auswertet, bis dies fehlschlägt. Für den Fall signalisiert ein "+" in der *otherwise*-Definition, daß normalize *gelingt*, und die Auswertung weiter gehen kann.

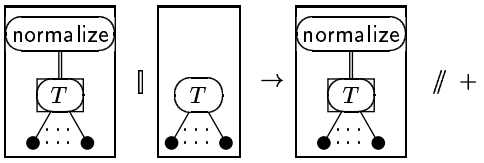


**Abbildung 3.** Der Kontrollkombinator normalize

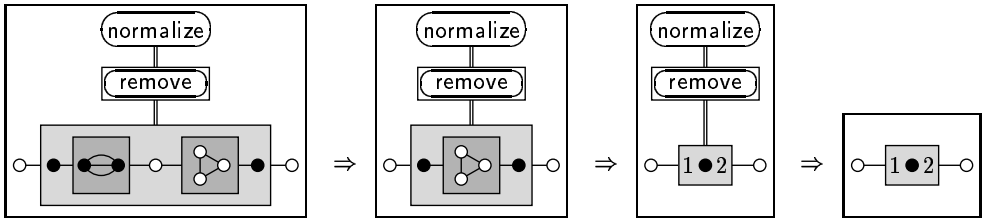


**Abbildung 4.** Eine Auswertung von normalize

Das Prädikat ist mit einer *bedingten Regel* der Form $P \,[\![\, A \to R$ definiert. Die *Anwendbarkeitsbedingung* $A$ "bewacht" die Regel: Nur wenn sie vollständig ausgewertet werden kann, ohne zu scheitern, wird die Regel angewendet.

Da $T$ ein Platzhalter für Prädikate mit beliebig vielen Parametern sein soll, benutzen wir Punkte, um eine veränderliche Anzahl von Knoten anzudeuten. Wo die $T$-Kante als Prädikatparameter benutzt wird, wird sie außerdem als einfache Kante *getarnt*, indem ein Kasten um das Oval gezogen wird. Das verhindert, dass sie ausgewertet wird, während sie "umhergetragen" wird (nämlich im Muster- und Ersetzungsgraph der Regel).

Abbildung 4 zeigt die Auswertung von normalize mit einem (getarnten) Aufruf von remove. Jede Anwendung von normalize entfernt einen Elementrahmen, indem remove als Anwendbarkeitsbedingung ausgewertet wird, bis der Listenrahmen leer ist, und remove fehlschlägt. (Der leere Listengraph besteht aus einem einzelnen Knoten, wobei die Ziffern 1 und 2 angeben, dass dieser Knoten gleichzeitig Start- und Endpunkt des leeren Listengraphen ist.)

Prädikate stellen also ähnliche Steuerungsmechanismen zur Verfügung wie funktionale und logische Sprachen: *Pattern matching* und *otherwise*-Definitionen erlauben Fallunterscheidungen; mit Anwendbarkeitsbedingungen kann die Reihenfolge von Prädikatanwendungen gesteuert werden. Zusammen mit Rekursion und Kombinatorprädikaten wie normalize reicht das schon aus, um in der Sprache Steuerung zu spezifizieren.

### 2.4 Kapselung

Wir sehen eine datenorientierte Art der Modularisierung vor. Abbildung 5 zeigt die globale Sicht einer Klasse $\mathsf{L}_\tau$. Die Klasse $\mathsf{L}_\tau$ ist eine Vorlage (*template*) für Listenrahmen. Ihre Definition enthält öffentliche Methoden enter oder remove. (In diesem einfachen Fall gibt es keine verborgenen Methoden.)

Wir fassen Listenrahmen als *Objekte* auf, die als Instanzen der Klasse $\mathsf{L}_\tau$ gebildet werden und auf *Nachrichten* (Aufrufe der öffentlichen Prädikate enter oder remove) reagieren. Die Struktur der Listengraphen ist jedoch nur in der Klasse, für deren Methoden sichtbar. Andere Objekte, die einen Listenrahmen $l$ enthalten, können auf dessen Inhalt nur zugreifen, indem sie eine Nachricht enter oder remove an $l$ schicken. So wird das Prinzip der *Datenabstraktion* bewahrt.

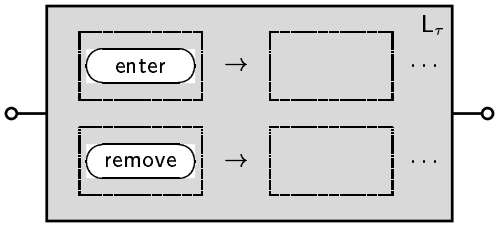


**Abbildung 5.** Die Listen-Klasse

### 2.5 Typisierung

Typen klassifizieren die Objekte eines Programms und geben Regeln für deren Benutzung vor. Über die übliche Typisierung von Graphen durch Markierung und Beschränkung des Grades von Knoten und Kanten hinaus erlauben wir, die *Struktur* (*shape* [9]) der in Rahmen enthaltenen Graphen zu definieren.

Abbildung 6 spezifiziert die Struktur von Listengraphen. Nur so aufgebaute

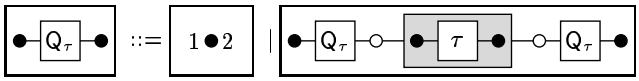


**Abbildung 6.** Die Struktur von Listengraphen

Graphen dürfen in Listenrahmen enthalten sein. Der Typ $\mathsf{L}_\tau$ der Listengraphen ist *polymorph*. Die *Typvariable* $\tau$ kann mit beliebigen Typen instanziiert werden. In unseren Beispielen hat die Variable $L$ den Typ $\mathsf{L}_\tau$, und die Variable $X$ den Typ $\tau$, der in den Transformationen von Abbildung 3 mit dem Typ der Elementgraphen instanziiert wird.

Jede Variable, jede Klasse, jede Regel und jeder Prädikatparameter hat einen so spezifizierten Typ, und es wird überprüft, ob alle Variablensubstitutionen, alle Rahmeninhalte, und die Muster- und Ersetzungsgraphen von Regeln diesen Typen *entsprechen*. Dann sind alle Graphen eines Programms wohlstrukturiert.

Die Strukturdefinitionen sind gut untersucht als *kontextfreie Graphtransformationen* [5]. Typüberprüfung bedeutet deshalb *kontextfreies Graphparsieren*, wie es in DiaGen implementiert ist (vgl. Abschnitt 3). Auch wenn Graphparsieren teuer ist, wird es *vor* Ausführung des Programms gemacht, und verringert somit den Suchraum für das *Graphmatching* zur Laufzeit.

## 3 Diagramme

Dieser Abschnitt ruft kurz in Erinnerung, wie Diagramme als Graphen repräsentiert werden können (beispielsweise in Diagrammeditoren), und wie dieses Modell dazu benutzt werden kann, Graphen *generisch* als Diagramme darzustellen.

### 3.1 Repräsentation und Analyse von Diagrammen

Andries *et al.* [1] haben ein Modell für die Repräsentation von Diagrammen mir Graphen vorgeschlagen, das modifiziert auch in DiaGen realisiert ist, einem Werkzeug zur Erzeugung von Diagrammeditoren, die sowohl freies als auch strukturiertes Editieren unterstützen [12, 14, 15]. Abbildung 7 zeigt die Ebenen der Diagramm-Repräsentation und die Schritte, in denen ein Diagramm analysiert wird. (Beachte die Analogie zur Analyse textueller Sprachen [26].)

Der *Scanner* baut einen *Layoutgraph* (*spatial relationship graph* [1]) auf, um die lexikalische Struktur des Diagramms festzustellen. Dieser Schritt benutzt Kanten, um die *Komponenten* einer Diagrammsprache zu repräsentieren (wie Kreise, Kästen, Pfeile und Text). Die Komponentenkanten sind mit Knoten verbunden, die die *Anknüpfungsbereiche* der Komponenten repräsentieren (wie *Rand* und *Fläche* von Kreisen und Kästen, *Quell-* und *Zielenden* von Pfeilen).

Die Komponenten können in ihren Anknüpfungsbereichen verbunden sein. Dies wird explizit durch binäre *Anordnungskanten* repräsentiert. Die Art einer Anordnungskante berücksichtigt die Typen der verbundenen Anknüpfungsbereiche und die Art ihrer Verbindung (wie *Überschneidung* oder *Enthaltensein*). Beispielsweise können sich *Quell-* und *Zielbereiche* von Pfeilen mit den *Randbe-*

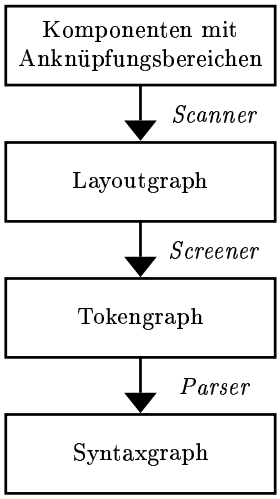


**Abbildung 7.** Diagrammrepräsentation und -erkennung in DiaGen

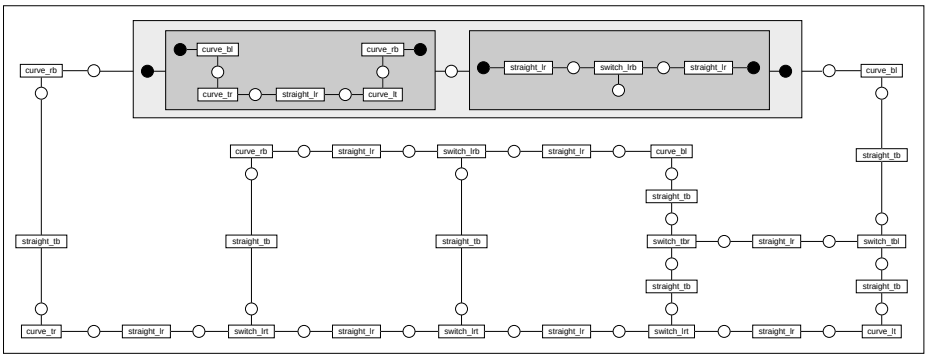


**Abbildung 8.** Der Syntaxgraph eines Schienendiagramms

*reichen* von Kreisen und Kästen überschneiden, was zu einer bestimmten Anordnungskante zwischen den betreffenden Knoten der Komponentenkanten führt.

Die weitere Analyse vollzieht sich in zwei Schritten: Der Layoutgraph wird zuerst vom *Screener* in einen kompakteren *Tokengraphen* überführt, indem Kantengruppen zu Einzelkanten zusammengefasst werden. Der Tokengraph wird dann gemäß der Syntax der Diagrammsprache *parsiert*. Diese zweistufige Analyse transformiert den Layoutgraph in die logischen Relationen des (*abstrakten*) *Syntaxgraphen* des Diagramms. Die Syntax von Diagrammen wird mit graphischen Regeln beschrieben, wie sie auch für die Typisierung von Graphen in Abschnitt 2.5 verwendet werden. Abbildung 8 repräsentiert den Layoutgraphen des Diagramms in Abbildung 9.

Mit diesem Modell können viele verschiedene Arten von Diagrammen repräsentiert und analysiert werden: endliche Automaten und Kontrollflußdiagramme [16], Nassi-Shneiderman-Diagramme (Struktogramme) [13], *message sequence charts* und *visual expression diagrams* [15], *sequential function charts* [12], und Kontaktpläne aus der SPS-Programmierung [14]. Derzeit sind uns keine Diagrammsprachen bekannt, die so nicht modelliert werden können.

### 3.2 Generische Visualisierung von Graphen

Da das oben beschriebene Modell zur Repräsentation und Analyse von Diagrammen Graphen benutzt, können wir es umgekehrt auch dazu benutzen, die Daten einer graphbasierten Programmiersprache als Diagramme zu visualisieren. Graphen sind zwar auch eine visuelle Darstellung, die aber für viele Anwendungen nicht angemessen ist. Statt dessen können wir für die *externe* Darstellung von Graphen beliebige visuelle Darstellungen nehmen. Die Benutzungsschnittstelle eines Programms kann so an die visuelle Notation seines Anwendungsgebietes angepasst werden. Damit kann die in hier vorgestellte Programmiersprache als *generische visuelle Programmiersprache* benutzt werden. Indem Graphen als verschiedene Diagrammarten repräsentiert werden, können verschiedene Stile visu-

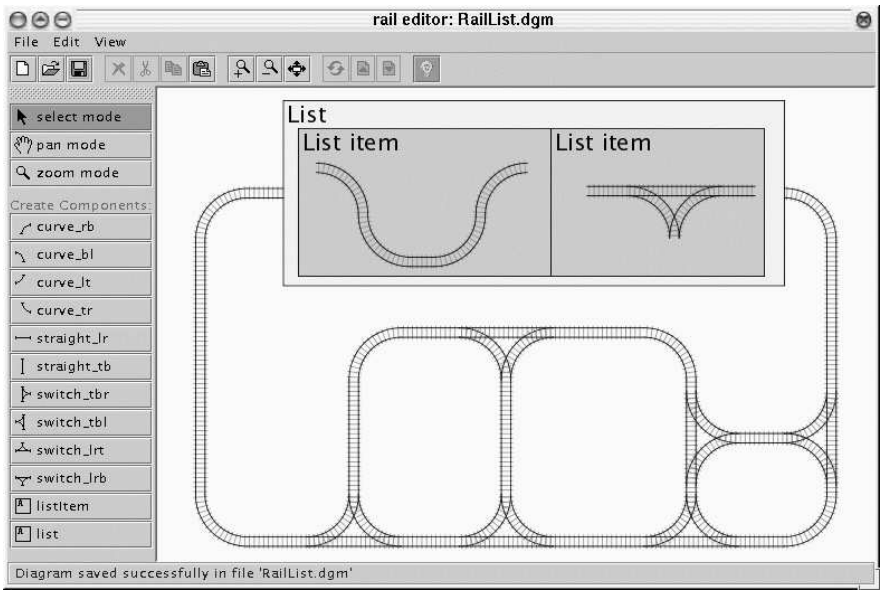


**Abbildung 9.** Ein Schienendiagramm mit einem Listendiagramm

eller (Programmier-) Sprachen beschrieben und implementiert werden. Naheliegende Beispiele wären *Pictorial Janus* [11] (dessen Agenten und *ports* unserem Begriff von typisierten Kanten direkt entsprechen) oder *KidSim* [22].

Abbildung 9 illustriert, wie die Generizität des Diagrammmodells zum Bau einer *KidSim*-artigen Umgebung benutzt werden kann. Der Bildschirmausschnitt zeigt einen Diagramm-Editor, der mit DiaGen erzeugt wurde: Datenstrukturen sind Eisenbahnschienen, erweitert um abstrakte Datentypen (Listen). Der Editor ist Teil der Benutzungsschnittstelle zu einem (noch nicht realisierten) Programm, das es erlaubt, Schienennetze zu bearbeiten. Abstrakte Datentypen (Listen etc.) werden dabei unter anderem benötigt, den Vorrat an Schienen oder bestimmte Konfigurationen (z. B. Bahnhöfe) verwalten zu können. Das Programmierwerkzeug (das derzeit noch nicht Bestandteil des Editors ist), manipuliert diese Eisenbahnschienen visuell. Intern, aber für den Benutzer nicht sichtbar, arbeitet das Werkzeug auf der Graph-Repräsentation in Abbildung 8. DiaGen benutzt ohnehin Graphen als interne Repräsentation, und ist daher für die generische Implementierung derartiger Umgebungen besonders gut geeignet.

## 4 Implementierung

Die in diesem Papier vorgestellte Programmiersprache ist noch nicht fertig entworfen. Ihre Implementierung steht erst ganz am Anfang und kann deshalb hier nur skizziert werden (vgl. Abbildung 10).

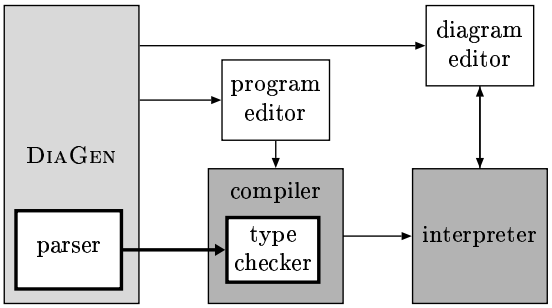


**Abbildung 10.** Systemarchitektur

- Der *Interpreter* führt Programme der Sprache aus, indem in einer Schleife – gesteuert vom Benutzer – ein Eingabegraph gelesen, gemäß dem Programm transformiert, und der modifizierte Graph wieder dargestellt wird.
- Der *Compiler* liest Programme und formt sie so um, dass die vom Interpreter leicht und effizient ausgeführt werden können, und überprüft, ob das Programm irgendwelche Regeln der Sprache verletzt. Dabei kommt der Typisierung (wie in Abschnitt 2.5 diskutiert) eine besondere Rolle zu: Die Typüberprüfung soll mithilfe des Graphparsers von DiaGen [13] realisiert werden.
- Mit einem interaktiven *Diagramm-Editor* werden die Eingabedaten für das Programm konstruiert. Dieser Editor soll mit DiaGen erzeugt werden, damit die intern verarbeiteten Graphen flexibel als Diagramme dargestellt werden können. Ein solcher Editor für eine *KidSim*-artige Diagrammnotation wurde bereits als Prototyp realisiert (vgl. Abschnitt 3.2). Abbildung 9 zeigt einen Bildschirmausschnitt.
- Die Programme werden mit einem interaktiven *Programm-Editor* für die visuelle Syntax der Sprache konstruiert. Auch dieser Editor soll mit DiaGen erzeugt werden.

Alles in allem muss also nur das Herz des Systems – Übersetzer und Interpreter – neu implementiert werden, während wir für die Erzeugung der Benutzungsschnittstelle ganz auf DiaGen bauen.

DiaGen muss noch in einigen Punkten ergänzt werden, um den Anforderungen dieser Anwendung gerecht zu werden. Insbesondere braucht das System selbst einen visuellen Editor. Es ist sicher keine Überraschung, dass auch dafür DiaGen benutzt werden soll.

Die Implementierung von Übersetzer und Interpreter ist eine anspruchsvolle Aufgabe. Auch wenn wir den Leser hoffentlich davon überzeugt haben, dass alle vorgeschlagenen Sprachkonzepte implementierbar sind, heißt das lange noch nicht, dass dies auch *effizient* geht, und zu *effizienten Systemen* führt. Die Effizienz der Implementierungen logischer und funktionaler Sprachen müsste erreicht werden können, weil die Fragen der Benutzungsschnittstelle vom Kern des Systems abgekoppelt werden können.

## 5 Schlußbetrachtungen

In diesem Papier haben wir eine auf geschachtelten Graphen basierende Programmiersprache vorgestellt, die regelbasiert, typisiert und objektorientiert ist. Wir haben insbesondere ihre Generizität in Hinblick auf Diagrammnotationen diskutiert.

Geschachtelte oder *hierarchische* Graphen wurden im Zusammenhang mit Graphtransformation bereits in [19, 8] vorgeschlagen, und auch in graphbasierten Datenbanksprachen [18] und in Systemmodellierungssprachen [25]. Graph-*Strukturen* gibt es in *Structured Gamma* [9] (für ungeschachtelte Graphen).

Wir kennen jedoch keine andere Sprache (auch keinen Entwurf), die geschachtelte Graphen, Transformationen, Typisierung und Klassen mit Generizität kombiniert.

Die präzise Definition der in diesem Papier genannten Konzepte wurde in [6] begonnen, und muss weiter geführt werden. Einige weitere Konzepte, wie *Nebenläufigkeit* und *Verteiltheit*, sollen noch untersucht werden. Da diese Konzepte schon in [23] und [24] in ähnlichem Zusammenhang studiert wurden, gibt es berechtigte Hoffnung darauf, dass sie sich auf unsere Sprache übertragen lassen.

## Literatur


1. Andries, M., G. Engels und J. Rekers: *How to Represent a Visual Specification.* In: Marriott, K. und B. Meyer (Hrsg.): *Visual Language Theory*, Kap. 8, S. 245–260. Springer, New York, 1998.
2. Burnett, M. M., A. Goldberg und T. G. Lewis (Hrsg.): *Visual Object-Oriented Programming.* Manning, Greenwich, CT, 1994.
3. Corradini, A. und U. Montanari (Hrsg.): *Proc. Joint* CompuGraph/Semagraph *Workshop on Graph Rewriting and Computation*, Nr. 2 in *Electronic Notes in Theoretical Computer Science*, `http://www.elsevier.nl/locate/entcs`, 1995. Elsevier.
4. Cox, P. T., F. R. Giles und T. Pietrzykowski: *Prograph.* In: Burnett, M. M. et al. [2], Kap. 3, S. 45–66.
5. Drewes, F., A. Habel und H.-J. Kreowski: *Hyperedge Replacement Graph Grammars.* In: Rozenberg, G. [20], Kap. 2, S. 95–162.
6. Drewes, F., B. Hoffmann und D. Plump: *Hierarchical Graph Transformation.* In: Tiuryn, J. (Hrsg.): *Foundations of Software Science and Computation Structures (FOSSACS 2000)*, Nr. 1784 in *Lecture Notes in Computer Science*, S. 98–113. Springer, März 2000.
7. Engels, G., H. Ehrig, H.-J. Kreowski und G. Rozenberg (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Specification and Programming.* World Scientific, Singapore, 1999.
8. Engels, G. und A. Schürr: *Encapsulated Hierachical Graphs, Graph Types, and Meta Types.* In: Corradini, A. und Montanari [3].
9. Fradet, P. und D. L. Métayer: *Structured Gamma.* Science of Computer Programming, 31(2/3):263–289, 1998.
10. Hoffmann, B.: *From Graph Transformation to Rule-Based Programming with Diagrams.* In: Nagl, M. und Schürr [17].

11. KAHN, K. M. und V. SARASWAT: *Complete Visualizations of Concurrent Programs and Their Executions.* In: *Proc. IEEE Workshop on Visual Languages (VL'90)*, S. 7–15, 1990.
12. KÖTH, O. und M. MINAS: *Generating Diagram Editors Providing Free-Hand Editing as well as Syntax-Directed Editing.* In: *Proc. International Workshop on Graph Transformation (GRATRA 2000), Berlin*, März 2000.
13. MINAS, M.: *Diagram Editing with Hypergraph Parser Support.* In: *Proc. 13th IEEE International Symposium on Visual Languages (VL'97), Capri, Italy*, S. 230–237. IEEE Computer Society Press, 1997.
14. MINAS, M.: *Creating Semantic Representations of Diagrams.* In: NAGL, M. und SCHÜRR [17].
15. MINAS, M.: *Hypergraphs as a Uniform Diagram Representation Model.* In: EHRIG, H., G. ENGELS, H.-J. KREOWSKI und G. ROZENBERG (Hrsg.): *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, Nr. 1764 in *Lecture Notes in Computer Science*, S. 281–295. Springer, 2000.
16. MINAS, M. und G. VIEHSTAEDT: *DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams.* In: HAARSLEV, V. (Hrsg.): *Proc. 11th IEEE International Symposium on Visual Languages (VL'95), Darmstadt, Germany*, S. 203–210. IEEE Computer Society Press, 1995.
17. NAGL, M. und A. SCHÜRR (Hrsg.): *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'99), Selected Papers*, Lecture Notes in Computer Science. Springer, Mai 2000.
18. POULOVASSILIS, A. und M. LEVENE: *A Nested-Graph Model for the Representation and Manipulation of Complex Objects.* ACM Transactions on Information Systems, 12(1):35–68, 1994.
19. PRATT, T. W.: *Pair Grammars, Graph Languages and String-to-Graph Translations.* Journal of Computer and System Sciences, 5:560–595, 1971.
20. ROZENBERG, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations.* World Scientific, Singapore, 1997.
21. SCHÜRR, A., A. WINTER und A. ZÜNDORF: *The* PROGRES *Approach: Language and Environment.* In: ROZENBERG, G. [20], Kap. 13, S. 487–550.
22. SMITH, D. C., A. CYPHER und J. SPOHRER: *KidSim: Programming Agents Without a Programming Language.* Communications of the ACM, 37(7):54–67, 1994.
23. TAENTZER, G.: *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* Dissertation, TU Berlin, 1996. Shaker Verlag.
24. TAENTZER, G. und A. SCHÜRR: *DIEGO, another step towards a module concept for graph transformation systems.* In: CORRADINI, A. und MONTANARI [3].
25. TAPKEN, J.: *Implementing Hierarchical Graph Structures.* In: FINANCE, J.-P. (Hrsg.): *Proc. Formal Aspects of Software Engineering (FASE'99)*, Nr. 1577 in *Lecture Notes in Computer Science.* Springer, 1999.
26. WILHELM, R. und D. MAURER: *Übersetzerbau — Theorie, Konstruktion, Generierung.* Springer, 2. Aufl., 1992.