Weitere Konzepte für Graphtransformationssysteme – Multiregeln und Kontrollfluss

Jens Kosiol

13. Mai 2024

Überblick

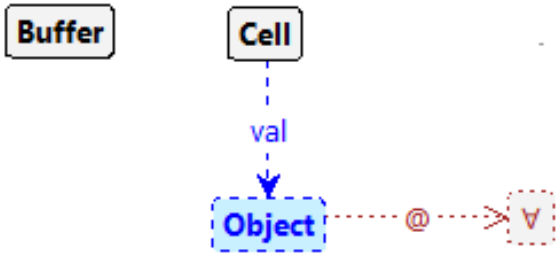
- Wie lässt sich eine for-each-Semantik kompakt in Graphtransformationsregeln integrieren?
  - *Multiobjekte als einfacher Spezialfall*
  - *Kernregel und Multiregeln*
- Kontrollfluss für Graphtransformationssysteme
  - *Graphprogramme*
  - *Steuerung durch Ebenen*
  - *Steuerung durch Priorisierung*
  - *Ein- und Ausgabeparameter*

Regeln mit Multiobjekten

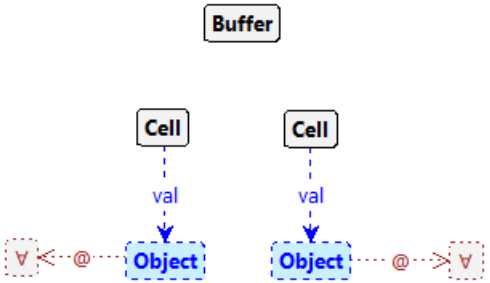
- Eine Multiregel kann Multiobjekte enthalten.
- Ein Multiobjekt ist ein zu löschender Knoten oder eine zu löschende Kante, der/die für mehrere Knoten/Kanten steht und so oft wie möglich angewendet wird.
  - *Kanten, die an Multiknoten starten oder enden, sind Multikanten.*
  - *Multiknoten sind unabhängig voneinander, also nicht durch Kanten verbunden*
- Kernregel: Alle Objekte der Multiregel, die nicht Multiobjekte sind.
- Anwendung einer Multiregel:
  - *Finden eines Ansatzes für die Kernregel (liefert Kernansatz)*
  - *Kernansatzerweiterung durch alle möglichen Ansätze der Multiobjekte (liefert mehrere Matches der Multiregel)*
  - *Anwendung der Multiregel (gleichzeitig mit Kernregel) an allen gefundenen Ansätzen (sogenannte amalgamierte Regel)*

Beispielregeln mit Multiobjekten

- In Groove sind Multiobjekte durch einen Zeiger auf einen ∀-markierten Knoten gekennzeichnet.
- Unterschiedliche Zeiger markieren unterschiedliche Multiknoten.
- *deleteObjects1* löscht alle Objekte aus einer Zelle.
- *deleteObjects2* löscht alle Objekte aus zwei Zellen.



deleteObjects1

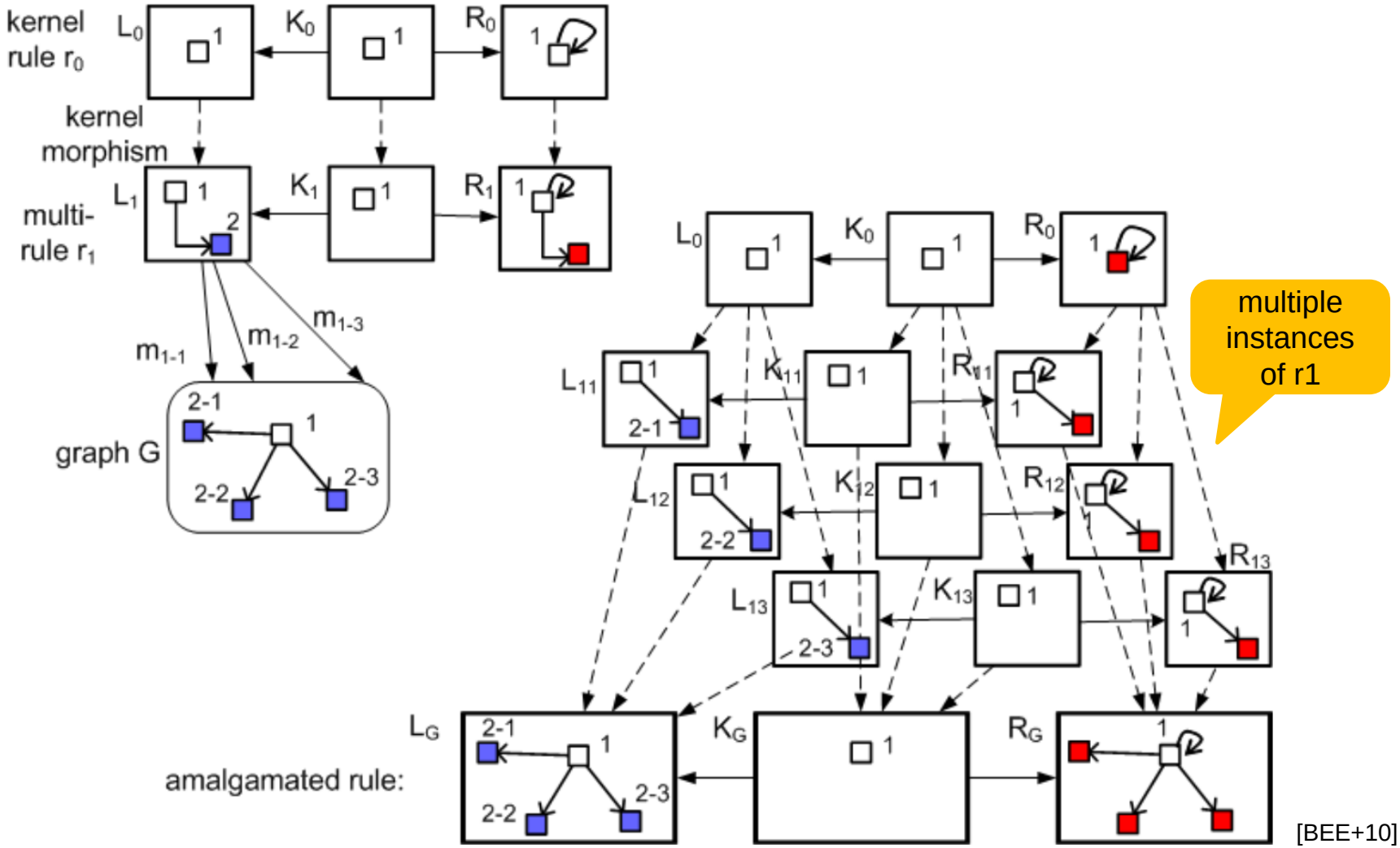


deleteObjects2

Multiregeln allgemein

- Eine Kernregel kann durch mehrere Multiregeln erweitert werden.
- Jede Multiregel kann die Kernregel durch zusammenhängende Strukturen und um jede Art von Element erweitern.
- Anwendung einer Multiregel:
  - *Finden eines Ansatzes für die Kernregel (liefert Kernansatz)*
  - *Kernansatz so oft wie möglich um Ansätze für Multiregeln erweitern, sodass diese (bei einmaliger Anwendung der Kernregel am Kernansatz) gleichzeitig anwendbar sind*
  - *Gleichzeitige Anwendung der Multiregeln (bei einmaliger Anwendung der Kernregel) an allen gefundenen Ansätzen (sogenannte amalgamierte Regel)*

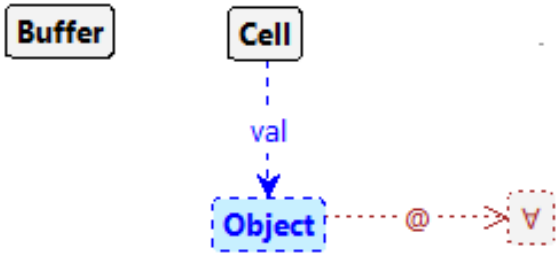
Amalgamierte Regel als Vereinigung



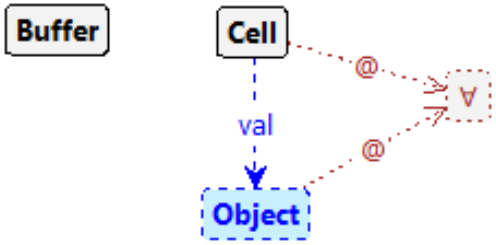
[BEE+10]

Beispiel Multiregeln

- In Groove sind Multiobjekte durch einen Zeiger auf einen ∀-markierten Knoten gekennzeichnet.
- *deleteObjects1* löscht alle Objekte aus einer Zelle.
- *deleteObjects3* löscht alle Objekte aus allen Zellen.



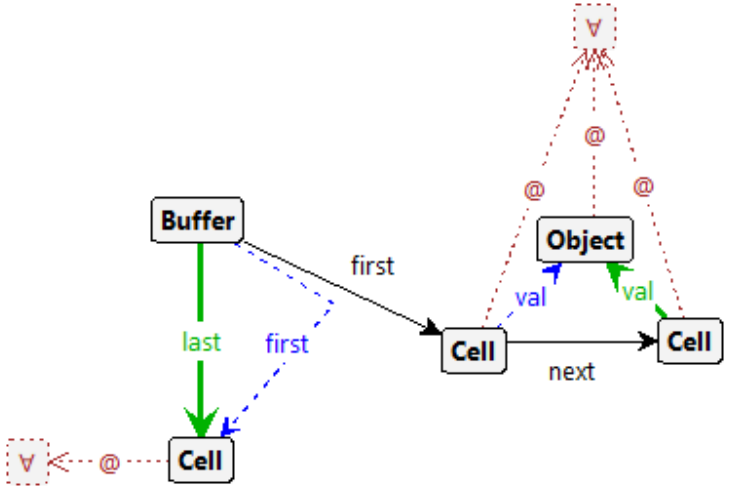
deleteObjects1



deleteObjects3

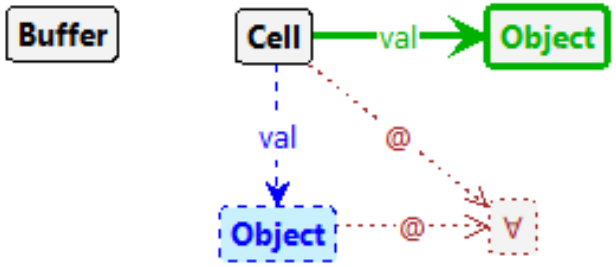
Beispiel Indeterminismus

- Jede first-Kante kann nur einmal gelöscht werden
- Jede Wahl, eine first-Kante zu löschen oder die dort gespeicherten Objekte weiterzuschieben, führt zu einer validen Transformation

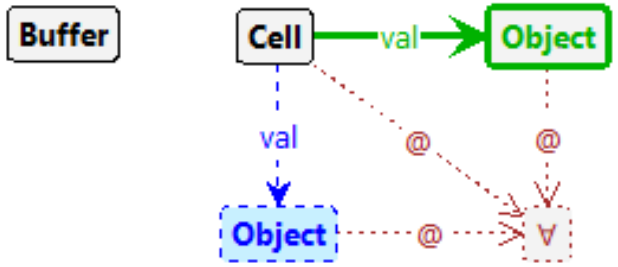


moveFirst

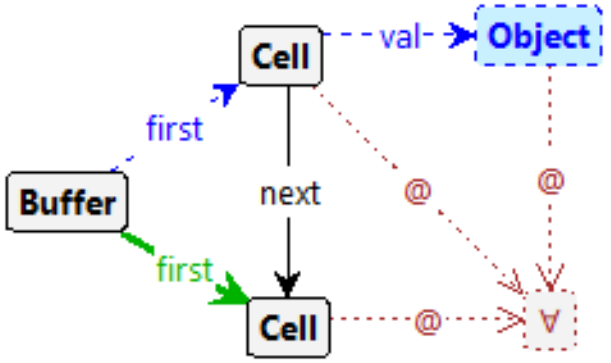
Welche Semantik haben die folgenden Multiregeln?



deleteObjects4



deleteObjects5



get2

Überblick

- Wie lässt sich eine for-each-Semantik kompakt in Graphtransformationsregeln integrieren?
  - *Multiobjekte als einfacher Spezialfall*
  - *Kernregel und Multiregeln*
- Kontrollfluss für Graphtransformationssysteme
  - *Graphprogramme*
  - *Steuerung durch Ebenen*
  - *Steuerung durch Priorisierung*
  - *Ein- und Ausgabeparameter*

Graphprogramme

- Atomare Operation ist die Anwendung von Graphtransformationsregeln
- Zusätzlicher Kontrollfluss durch
  - *Sequentielle Anwendung von Programmen*
  - *Iteration von Programmen*
- Programme sind in der Regel nichtdeterministisch
- Ohne Attribute kann auf Parameter verzichtet werden

Definition: Graphprogramm

Gegeben ist ein Typgraph TG (oder TGI). Ein (Graph)Programm über TG ist wie folgt induktiv definiert:

1. Jede endliche Menge R von über TG getypten Regeln ist ein Programm (elementares Programm).
2. Sind P_1 und P_2 Programme, so ist $< P_1; P_2 >$ ein Programm (sequentielle Komposition).
3. Ist P ein Programm, so ist $P\downarrow$ ein Programm (Iteration).

Semantik von Graphprogrammen

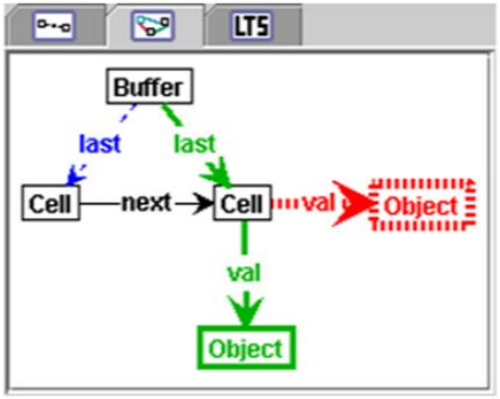
- Semantik $\rightarrow_P$ eines Programms P ist eine Relation auf über TG getypten Graphen (Input-Output-Paare)
  - $G \rightarrow_P H$ bedeutet, dass das Programm *P* für den Eingabegraphen G den Graphen H berechnen *kann*
- $\rightarrow_P$ ist im Allgemeinen *nicht* linkstotal, d.h., für einen Graphen G muss kein Graph H existieren mit $G \rightarrow_P H$.
  - *P kann ohne Ergebnis abbrechen*
  - *P muss nicht terminieren*
- $\rightarrow_P$ ist im Allgemeinen *nicht* rechtseindeutig, d.h., es kann $H_1 \neq H_2$ geben, sodass $G \rightarrow_P H_1$ und $G \rightarrow_P H_2$.

Definition: Programmsemantik

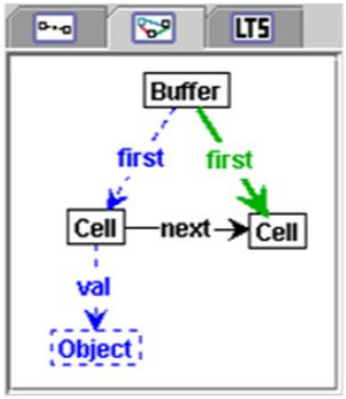
Gegeben ein Graphprogramm P über dem Typgraphen TG ist die Semantik $\rightarrow_P$ von P wie folgt induktiv definiert:

1. Ist P ein elementares Programm R, so gilt $G \rightarrow_P H$ genau dann, wenn $G \Rightarrow_R H$ gilt, also, wenn eine Regel $r \in R$ und ein Ansatz m existieren, sodass $G \Rightarrow_{r,m} H$.
2. Ist $P = P_1; P_2$ eine sequentielle Komposition von Programmen, dann gilt $G \rightarrow_P H$ genau dann, wenn ein Graph X existiert mit $G \rightarrow_{P_1} X$ und $X \rightarrow_{P_2} H$.
3. Ist $P = Q!$ eine Iteration, so gilt $G \rightarrow_P H$ genau dann, wenn $G \rightarrow_Q^* H$ (reflexiv-transitiver Abschluss) und kein Graph X existiert mit $H \rightarrow_Q X$.

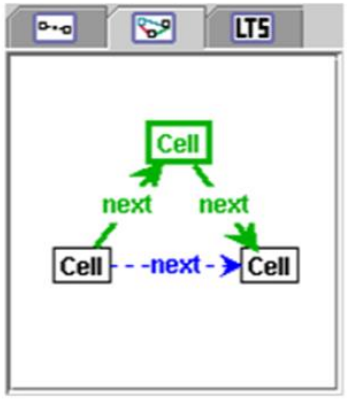
Beispielprogramme



(a) put rule.

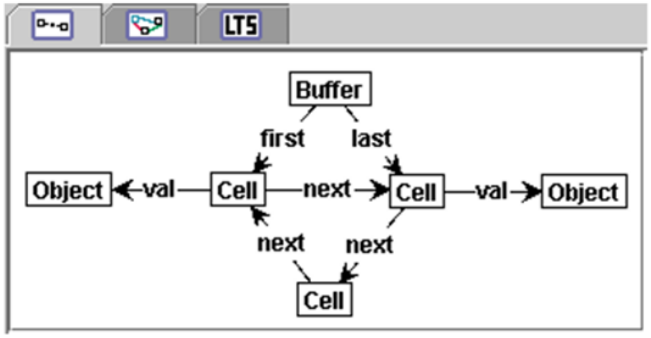


(b) get rule.



(c) extend rule.

[KR06]



Was berechnen die folgenden Programme für den linken Graphen G als Input? D.h., welche Menge beschreibt die Semantik eingeschränkt auf G als erste Komponente?

- put↓
- put;put;put
- extend↓
- {put, get};{put, get}
- {}

Der leere Graph

- Der *leere Graph* $\emptyset$ ist der eindeutige Graph, dessen Knoten- und Kantenmenge jeweils die leere Menge ist.
- Anpassung für erweiterte Graphen: Die Menge der Attributierungskanten ist leer, die Datenknoten bilden die sogenannte *initiale Algebra*.
- *Universelle Eigenschaft*: Für jeden (erweiterten) Graphen G gibt es genau einen Morphismus $\emptyset \to G$.
  - *Insbesondere können wir für jeden beliebigen Typgraphen TG annehmen, dass $\emptyset$ über TG getypt ist.*

Skip-Semantik für Programme

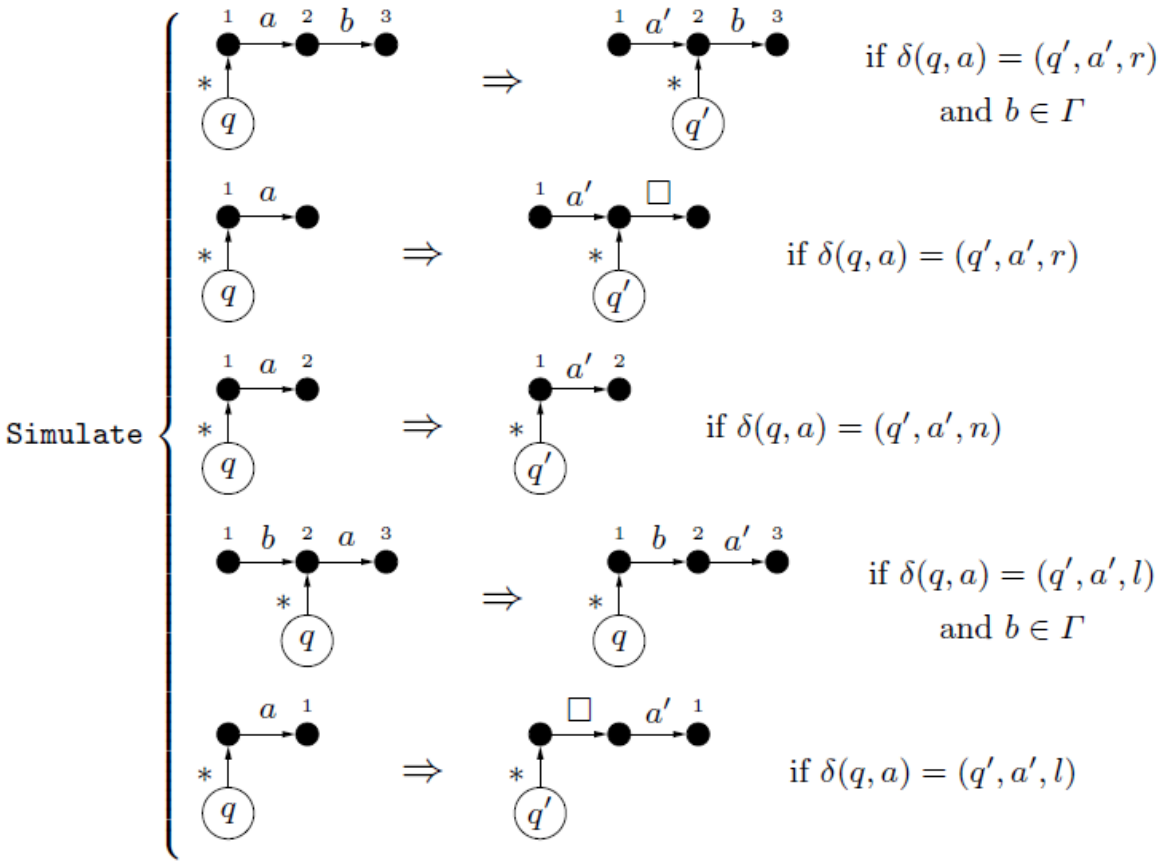
- Die *leere Regel* ist die Regel $\emptyset \Rightarrow \emptyset$
  - *Die leere Regel ist auf jeden (getypten, attributierten) Graphen G anwendbar mit Ergebnis G.*
  - *Ansatz ist der eindeutige Morphismus $\emptyset \to G$*
- Leere Regel in elementaren Programmen:
  - *Ist $P = R$ ein elementares Programm und $\emptyset \Rightarrow \emptyset \in R$, so gilt $G \to_P G$ für jeden Graphen G.*
  - *Ist $P = R$ ein elementares Programm und $R = \{\emptyset \Rightarrow \emptyset\}$, so gilt $\to_P = \{(G, G) \mid G \text{ ist über } TG \text{ getypt}\}$, wobei TG der gegebene Typgraph ist.*

Formale Eigenschaften von Graphprogrammen

Graphprogramme wie oben definiert sind *vollständig* und *minimal*:

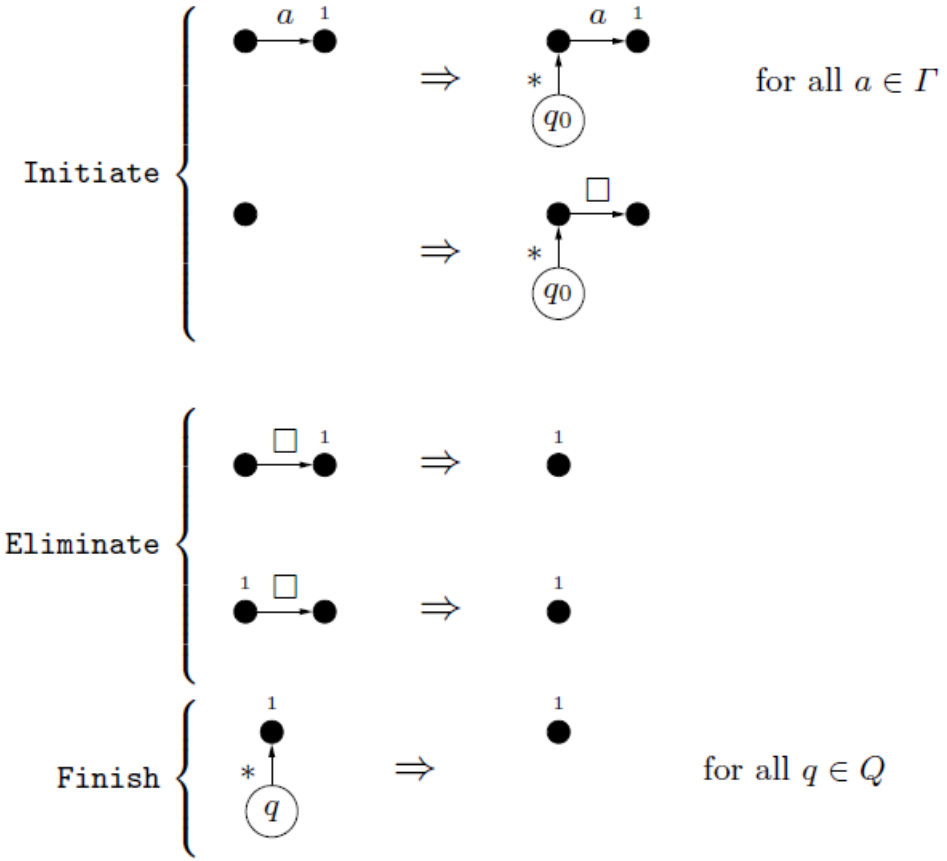
- Es gibt eine Übersetzung toGr(_) von Strings in Graphen, sodass es für jede berechenbare partielle Funktion $f: \Sigma_1^* \to \Sigma_2^*$ zwischen Wörtern über Alphabeten Σ_1 und Σ_2 einen Typgraphen TG und ein Programm P_f gibt, sodass für alle $w \in \Sigma_1^*$ und alle über TG getypten Graphen G gilt: $\mathrm{toGr}(w) \to_{P_f} G \Leftrightarrow G = \mathrm{toGr}\big(f(w)\big)$. [Theorem 1, HP01]
- Es gibt eine Übersetzung toStr von Graphen in Strings, sodass für jede partielle Funktion f zwischen Graphen, deren Übersetzung toStr(f) berechenbar ist, ein Programm P existiert, das f berechnet. [Theorem 2, HP01]
- Die Menge der Programme ohne Iteration oder sequentielle Komposition ist nicht vollständig. [Theorem 3, HP01]

Simulation von Turingmaschinen



[HP01]

Simulation von Turingmaschinen II



[HP01]

Graphprogramme in Groove

- Per *Edit* → *New* → *New Control* wird eine neue Kontrollstruktur für eine Grammatik erzeugt
  - *Gehört eine aktivierte Kontrollstruktur zu einer Grammatik, wird diese automatisch bei der Exploration des Suchraums berücksichtigt*
- Elementare Programme: Regelnamen verbunden per |
- Sequentielle Komposition per Semikolon
- Iteration: *alap*()
- Unterstützt Parameter und Konstrukte wie *while*, *if-then-else*, *try*, … (syntactic sugar)

Regelpriorisierung

- Leichtgewichtige Kontrollstruktur in Groove
  - *Über Rechtsklick auf Regelnamen einstellbar (*set, lower, *und* raise priority*)*
- Semantik Regelpriorisierung:
  - *Solange wie möglich wird von den Regeln mit der höchsten Priorität zufällig eine der anwendbaren gewählt und angewendet.*
  - *Ist keine der Regeln mit der höchsten Priorität anwendbar, wird zufällig eine der Regeln der nächst niedrigeren Priorität angewendet usw.*
  - *Terminiert, sobald auf keine Regel mehr anwendbar ist.*
- Vermeidet tiefe Schachtelungen von if und/oder try

Kontrollfluss durch Ebenen

- Das Programm besteht aus einer Sequenz von Ebenen $E_1, \ldots, E_n$
  - *Jede Ebene ist eine Menge von Regeln*
  - *Nacheinander werden die Regeln einer Ebene so lange angewendet wie möglich. Sobald keine Regel der Ebene mehr anwendbar ist, wird zur nächsten Ebene übergegangen*
  - *Innerhalb der Anwendung einer Ebene: Immer zufällige Auswahl einer der anwendbaren Regeln*
- Als Graphprogramm: $E_1 \downarrow; E_2 \downarrow; \ldots; E_n \downarrow$
- Praktisch oft zur Übersetzung von Graphen benutzt

Eingabeparameter

Ein Eingabeparameter ist ein Teil der Regel, der nicht randomisiert gematcht wird, sondern gemäß externer Informationen.

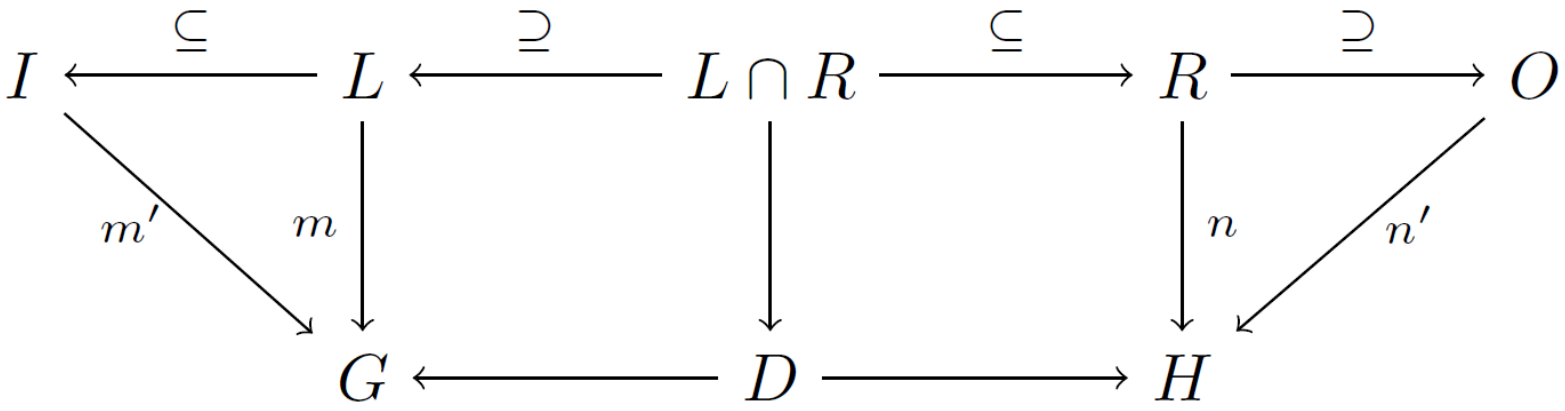
- Formal: Die Eingabeparameter bilden einen Untergraphen *I* von *L* (der linken Regelseite)
- Die Regel braucht einen „Teilansatz“ $m' : I \to G$, um auf einen Graphen *G* angewendet werden zu können (Binden der Parameter)
- Eine Regelanwendung erfolgt an einem Ansatz $m : L \to G$, sodass m und m' auf *I* übereinstimmen

Ausgabeparameter

Ein Ausgabeparameter ist Information, die eine Regel über ihre Anwendung zurückgibt.

- Formal: Die Ausgabeparameter bilden einen Untergraphen *O* von *R* (der rechten Regelseite)
- Die Regelanwendung liefert einen „Teilkoansatz“ $n' : O \to H$, der auf *O* mit dem Koansatz $n : R \to H$ übereinstimmt

Regelanwendung mit Parametern schematisch



Transaktionen

- Für Outputparameter O der ersten Regel und Inputparameter I der zweiten Regel gilt $I = O$. Dann Transaktion durch: Teilkoansatz n' der Anwendung der ersten Regel liefert Teilansatz m' für Anwendung der zweiten Regel.
- Erweiterungen:
  - *Es ist erlaubt, dass O größer als I ist (überflüssige Eingaben werden ignoriert)*
  - *Es ist erlaubt, dass I größer als O ist (fehlende Eingaben werden zufällig gematcht)*
  - *Verbindung zwischen I und O nicht über Gleichheit von Elementen, sondern über (partielle) Graphmorphismen*

Zusammenfassung

- Multiregeln ergänzen Graphtransformationsregeln um die Möglichkeit, ausgewählte Operationen so oft wie möglich durchzuführen.
  - *Zu löschende Multiobjekte bilden einen einfachen Spezialfall*
- Graphprogramme steuern die Ausführung von Graphtransformationsregeln
  - *Zufällige Auswahl aus Regelmenge, sequentielle Komposition und Iteration sind ausreichend*
  - *Tools wie Groove bieten zusätzliche Kontrollstrukturen*
- Priorisierung oder Einteilung in Ebenen werden praktisch auch häufig genutzt

Literatur und Links

- Reiko Heckel, Gabriele Taentzer: Graph Transformation for Software Engineers, Springer, 2020, Kapitel 2 und 3
- Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, Gabriele Taentzer: Fundamentals of Algebraic Graph Transformation, Springer, 2006
- Groove: Graphs for Object-Oriented Verification, groove.cs.utwente.nl
- [BEE+10] E. Biermann, H. Ehrig, C. Ermel, U. Golas, G. Taentzer: Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation. In “Graph Transformations and Model-Driven Engineering“, Springer, 2010: 121–140
- [HP01] A. Habel, D. Plump: Computational Completeness of Programming Languages Based on Graph Transformation. FOSSACS 2001: 230–245