Diplomarbeit

Graphsprachen für die Spezifikation von Invarianten bei verteilten und dynamischen Systemen

am Lehrstuhl für Theoretische Informatik
Prof. Dr. Barbara König
Universität Duisburg-Essen

von

Christoph Blume

Betreuer: Prof. Dr. Barbara König,
Dr. H. J. Sander Bruggink

Tag der Anmeldung: 09. Mai 2008
Tag der Abgabe: 09. November 2008

Die Mathematik ist das Instrument, welches die Vermittlung bewirkt zwischen Theorie und Praxis, zwischen Denken und Beobachten: sie baut die verbindende Brücke und gestaltet sie immer tragfähiger. Daher kommt es, daß unsere ganze gegenwärtige Kultur, soweit sie auf der geistigen Durchdringung und Dienstbarmachung der Natur beruht, ihre Grundlage in der Mathematik findet.

(David Hilbert)

Danksagung

An dieser Stelle möchte ich mich ganz herzlich bei Frau Prof. Dr. Barbara König und Herrn Dr. H.J. Sander Bruggink für ihre Unterstützung während der Anfertigung dieser Arbeit und das Wecken meiner Begeisterung für diesen Teil der theoretischen Informatik bedanken. Sie haben mit vielen interessanten und hilfreichen Diskussionen dazu beigetragen, dass diese Arbeit in der vorliegenden Form überhaupt entstehen konnte.

Außerdem möchte ich meinen Eltern Dank aussprechen, da sie mir dieses Studium überhaupt erst ermöglicht und mich stets unterstützt haben.

Mein besonderer Dank gilt auch Anke-Susann Dabels und Romina Schulz dafür, dass sie nicht nur während der Zeit der Diplomarbeit, sondern während meines gesamten Studiums eine große Hilfe für mich waren und mir den nötigen Rückhalt gegeben haben.

Schließlich möchte ich mich insbesondere bei Özlem Acikel, Daniel Deja und Martin Friedrich bedanken, die mir beim Korrigieren dieser Arbeit geholfen und auf so manchen Fehler hingewiesen haben.

Inhaltsverzeichnis

Kurzfassung

Die vorliegende Diplomarbeit befasst sich mit erkennbaren Graphsprachen und der Frage wie diese für die Spezifikation von Invarianten verwendet werden können. Die grundlegende Ansatz hierbei ist, dass viele Systeme, darunter beispielsweise verteilte Systeme, die aus mehreren untereinander verbundenen Komponenten bestehen, in ihrer statischen Form als Graphen beschrieben werden können. Die Dynamik solcher Systeme kann durch Graphtransformationsregeln spezifiziert werden, die sowohl das Entstehen und Verschwinden von Komponenten, als auch die Umstrukturierung der Topologie beschreiben können.

Ausgehend von den regulären Wortsprachen in Kombination mit Zeichenersetzungsregeln wird das Konzept der Invarianten auf Graphsprachen in Zusammenhang mit Graphtransformationsregeln erweitert. Dazu wird das Theorem, welches besagt, dass eine Wortsprache genau dann regulär ist, wenn sie bezüglich einer monotonen Wohlquasiordnung nach oben abgeschlossen ist, entsprechend für erkennbare Graphsprachen verallgemeinert. Aufbauend auf diesen Ergebnissen wird zunächst ein Algorithmus zur Überprüfung von Invarianten bei Wortsprachen entwickelt, der anschließend als Grundlage für einen entsprechenden Algorithmus dient, der auf Graphsprachen anwendbar ist. Anhand dieses Algorithmus' können erkennbare Graphsprachen daraufhin untersucht werden, ob sie invariant unter der Anwendung einer Graphtransformation sind.

Für die Untersuchung der praktischen Realisierbarkeit wird parallel zu den theoretischen Überlegungen ein Programm entwickelt, welches den Algorithmus zum Überprüfen von Invarianten bei erkennbaren Graphsprachen implementiert. Mit Hilfe dieses Programms werden anhand eines Fallbeispiels, bei dem die Zugriffsregeln eines Mehrbenutzersystems verifiziert werden sollen, die Einsatzmöglichkeiten der vorgestellten Theorien aufgezeigt.

Kapitel 1

Einleitung

1.1 Motivation

In vielen Bereichen der Informatik und sowie in anderen Wissenschaften werden Graphen zur Modellierung von komplexen Systemen und Sachverhalten eingesetzt. Zu den bekannten Beispielen aus der (theoretischen) Informatik zählen unter anderem endliche Automaten, welche einfache, sequentielle Abläufe modellieren sollen, sowie Petrinetze, mit denen auch nebenläufige Systeme beschrieben werden können. Mit Hilfe von Graphen wird dabei die statische Form dieser Systeme modelliert. Dynamische Änderungen des daraus resultierenden Modells können durch Graphtransformationsregeln [23] spezifiziert werden, die aus linker und rechter Seite bestehen, und lokale Änderungen auf dem Graphen beschreiben. Eine (potentiell unendliche) Menge solcher graphisch dargestellten Systeme bildet insbesondere eine Graphsprache.

Unter den Wortsprachen stellen die regulären Sprachen auf Grund ihrer Eigenschaften eine besondere Sprachklasse dar. Neben unterschiedlichen Charakterisierungen, wie zum Beispiel durch endliche Automaten, in Form von regulären Grammatiken bzw. Ausdrücken oder in Bezug auf Kongruenzrelationen, verfügt diese Sprachklasse auch über zahlreiche Abschlusseigenschaften, wodurch die regulären Sprachen in vielen Anwendungen, wie zum Beispiel zur Verifikation, benutzt werden. Wie im Fall von Wortsprachen ist es auch möglich Graphsprachen in verschiedene Sprachklassen einzuteilen. Von besonderer Bedeutung für diese Arbeit ist dabei die Klasse der erkennbaren Graphsprachen, die vor allen von Courcelle untersucht wurden [7]. Hierbei ist Erkennbarkeit analog zur Regularität im Falle von Wortsprachen zu verstehen. Es gibt Charakterisierungen erkennbarer Sprachen, die mittels der Kategorientheorie ausgedrückt werden können, und die an reguläre Wortsprachen angelehnt sind. Genannt seien hier beispielsweise die Definition in Bezug auf Kongruenzrelationen [11] bzw. durch eine bestimmte Klasse von Automaten, sogenannten Automatenfunktoren [6]. Zahlreiche wichtige Eigenschaften, wie zum Beispiel Färbbarkeit, Planarität, Existenz oder Nicht-Existenz bestimmter Subgraphen, können als erkennbare Graphsprachen spezifiziert werden.

Eine wichtige Fragestellung besteht darin, festzustellen, ob eine gegebene Menge von Graphen bzw. eine Graphsprache invariant unter dem dynamischen Verhalten, das heißt unter Anwendung der Graphtransformationen, ist. Dieser Ansatz kann zur Verifikation der modellierten Systeme genutzt werden. Somit ist es einerseits möglich nachzuweisen, dass bestimmte erwünschte Systemeigenschaften auch bei dynamischen Änderungen des Systems erhalten bleiben. Andererseits kann gezeigt werden, dass bestimmte unerwünschte Systemeigenschaften nicht durch das Verhalten des Systems hervorgerufen werden können. Diese Erkenntnis ist beispielsweise wichtig, wenn es darum geht die Sicherheit eines (verteilten) Mehrbenutzersystems

in Bezug auf den gemeinsam lesenden sowie schreibenden Zugriff auf Ojekte zu beweisen.

1.2 Problemstellung

Im Rahmen dieser Diplomarbeit soll zum einen überprüft werden, ob Eigenschaften, die Invarianten dynamischer und verteilter Systeme bilden, als erkennbare Graphsprachen beschrieben werden können. Aufbauend auf diesem Ergebnis soll dann zum anderen ein Algorithmus entwickelt werden, mit dessen Hilfe automatisch überprüft werden kann, ob eine gegebene Menge von Graphtransformationen, die das Verhalten des Systems beschreiben, eine bestimmte Invariante erhalten.

Um diese Untersuchungsziele zu erreichen, soll zunächst das Konzept der Invarianten für den Fall von regulären Wortsprachen untersucht werden. Für diese wird darauf aufbauend ein Algorithmus entwickelt, mit dem überprüft werden kann, ob bestimmte (zeichenbasierte) Ersetzungsregeln die jeweiligen Invarianten erhalten. Nachfolgend sollen die erzielten Ergebnisse zuerst auf Pfeilsprachen verallgemeinert werden. Anschließend wird die Theorie auf Graphsprachen – als Spezialfall von Pfeilsprachen – angewandt, um einen Algorithmus zur Überprüfung von Invarianten bei Graphsprachen zu erhalten und somit das Hauptziel dieser Arbeit zu erreichen.

Um die praktische Realisierbarkeit zu zeigen, wird parallel zu den oben genannten, theoretischen Überlegungen ein Programm entwickelt. Dieses soll einerseits einen Automatenfunktor zur Erkennung aller Graphen, die einen bestimmten Subgraphen enthalten, implementieren und andererseits den zuvor entwickelten Algorithmus realisieren. Abschließend werden mit Hilfe des Programms Zugriffsregeln für ein Mehrbenutzersystem verifiziert.

1.3 Aufbau der Arbeit

Kapitel 2 – Grundlagen der Ordnungstheorie:
Im zweiten Kapitel werden zunächst grundlegende Definitionen und Sätze der Ordnungstheorie eingeführt, da hierauf die gesamte Arbeit aufbaut.

Kapitel 3 – Wortsprachen:
Das dritte Kapitel führt in die Theorie der Wortsprachen ein, wobei insbesondere verschiedene Charakterisierungen der Klasse der regulären Sprachen angegeben werden. Für diese Arbeit ist dabei das Theorem, dass eine Sprache genau dann regulär ist, wenn sie bezüglich einer monotonen Wohlquasiordnung nach oben abgeschlossen ist, von besonderer Bedeutung. Abschließend wird ein Algorithmus entwickelt, der automatisch prüft, ob eine gegebene reguläre Sprache eine Invariante bezüglich bestimmter Ersetzungsregeln ist.

Kapitel 4 – Kategorientheorie und Pfeilsprachen:
Zunächst wird im vierten Kapitel eine Einführung in die Kategorientheorie gegeben und der Begriff der erkennbaren Pfeilsprache mit Hilfe der Kategorientheorie eingeführt. Im Anschluss daran werden die Ergebnisse aus Kapitel drei auf erkennbare Pfeilsprachen verallgemeinert, um eine Grundlage zur Entwicklung eines Algorithmus' zur Prüfung von Invarianten bei Graphsprachen zu schaffen.

Kapitel 5 – Graphsprachen:
Im fünften Kapitel werden erkennbare Graphsprachen und sogenannte Graphoperationen mittels der Kategorie der Cospans von Graphen eingeführt. Aufbauend auf den Ergebnissen des vorherigen Kapitels wird abschließend ein Algorithmus zur Prüfung von Invarianten hergeleitet, dessen Grundlagen im vorherigen Kapitel gelegt wurden. Anschließend wird ein Automatenfunktor definiert, die die Sprache aller Graphen erkennt, die einen bestimmten Subgraphen enthalten.

Kapitel 6 – Implementierung:
Das sechste Kapitel beschreibt die Implementierung des Programms zur Simulation des im vorherigen Kapitels definierten Automatenfunktors sowie des dort vorgestellten Algorithmus' zur Prüfung von Invarianten.

Kapitel 7 – Verifikation von Zugriffsregeln für ein Mehrbenutzersystem:
Im siebten Kapitel wird erläutert, wie mit Hilfe des in dieser Arbeit entwickelten Algorithmus' unter anderem Zugriffsregeln für Mehrbenutzersysteme verifiziert werden können. Dazu werden sicherheitskritische Systemzustände als (Sub-)Graphen modelliert. Mit Hilfe des in Kapitel sechs vorgestellten Programms kann anschließend überprüft werden, ob durch die Zugriffsregeln ein sicherheitskritischer Zustand zu erreichen ist.

Kapitel 8 – Fazit und Ausblick:
Das achte Kapitel gibt ein abschließendes Fazit und zeigt mögliche Anknüpfungspunkte für weitergehende Arbeiten auf.

Kapitel 2

Grundlagen der Ordnungstheorie

In diesem Kapitel werden zunächst einige grundlegende Begriffe aus der mathematischen Ordnungstheorie eingeführt, die im weiteren Verlauf für die Charakterisierung der betrachteten Sprachen und für die Untersuchung von Invarianten benötigt werden. Weitere relevante Definitionen werden im konkreten Zusammenhang mit den betreffenden Stellen eingeführt. Die Definitionen und Sätze dieses Abschnittes sind vor allem aus [18] entnommen.

2.1 Quasiordnungen

Definition 2.1 (Quasiordnung)**.** *Eine* Quasiordnung (QO) *ist eine zweistellige Relation $\sqsubseteq$ auf einer Menge M, welche folgende Bedingungen erfüllt:*

- $\sqsubseteq$ *ist* reflexiv*: Für alle $m \in M$ ist $m \sqsubseteq m$,*
- $\sqsubseteq$ *ist* transitiv*: Für alle $m_1, m_2, m_3 \in M$ folgt aus $m_1 \sqsubseteq m_2$ und $m_2 \sqsubseteq m_3$ auch $m_1 \sqsubseteq m_3$.*

Man nennt $(M, \sqsubseteq)$ eine quasigeordnete Menge *oder ebenfalls kurz eine* Quasiordnung*. Eine endliche oder unendliche Folge $m_1, m_2, m_3, \ldots$ von Elementen aus M wird* aufsteigend *bzw.* absteigend *genannt, falls $m_i \sqsubseteq m_{i+1}$ bzw. $m_{i+1} \sqsubseteq m_i$ für alle $i \in \mathbb{N}$ gilt.*

Eine Teilmenge $N \subseteq M$, die die folgende Bedingung

$$(m \in N \wedge m \sqsubseteq n) \implies n \in N.$$

erfüllt, heißt nach oben abgeschlossen *bezüglich $\sqsubseteq$.*

Außerdem soll für eine Quasiordnung $\sqsubseteq$ auf M mit $\sqsubseteq^{-1}$ die zu $\sqsubseteq$ *inverse Quasiordnung* bezeichnet werden. Diese ist für zwei beliebige Elemente $m_1, m_2 \in M$ wie folgt definiert:

$$m_1 \sqsubseteq^{-1} m_2 \text{ gdw. } m_2 \sqsubseteq m_1.$$

Definition 2.2. *Eine Quasiordnung $\sqsubseteq$ heißt* total*, wenn für zwei Elemente $m_1, m_2 \in M$ immer $m_1 \sqsubseteq m_2$ oder $m_2 \sqsubseteq m_1$ gilt.*

Definition 2.3 (Kette)**.** *$K \subseteq M$ heißt* Kette*, wenn $\sqsubseteq$ eingeschränkt auf $K \times K$ total ist. Eine Kette K heißt* aufsteigende Kette*, wenn für jede endliche oder unendliche Folge $k_1, k_2, k_3, \ldots$ von Elementen aus K gilt, dass $k_i \sqsubseteq k_{i+1}$ für alle $i \in \mathbb{N}$ ist.*

Definition 2.4 (Aufsteigende Kettenbedingung)**.** *Eine Quasiordnung $\sqsubseteq$ erfüllt die* aufsteigende Kettenbedingung *(engl.: Ascending Chain Condition), wenn es für jede (unendliche) aufsteigende Kette $k_1, k_2, k_3, \ldots$ einen Index m gibt mit $k_i = k_{i+1}$ für $i \geq m$.*

Falls M nicht nur eine Menge sondern sogar ein Monoid ist, das bedeutet, auf der Menge M ist eine binäre Operation $\circ$ definiert, die *assoziativ* ist, das heißt für alle $m_1, m_2, m_3 \in M$ gilt:

$$(m_1 \circ m_2) \circ m_3 = m_1 \circ (m_2 \circ m_3),$$

und für die ein *neutrales Element* $e \in M$ existiert, so dass für alle $m \in M$ gilt:

$$e \circ m = m \circ e = m,$$

kann man Folgendes definieren:

Definition 2.5. *Eine Quasiordnung $\sqsubseteq$ auf einem Monoid M heißt* links- *bzw.* rechts-monoton*, falls für alle $w, x, y \in M$ gilt:*

$$x \sqsubseteq y \implies wx \sqsubseteq wy \quad \textit{bzw.} \quad x \sqsubseteq y \implies xw \sqsubseteq yw.$$

Ist $\sqsubseteq$ sowohl links- als auch rechts-monoton, so wird $\sqsubseteq$ auch monoton *genannt.*

2.2 Wohlquasiordnungen

Für diese Arbeit ist vor allem die folgende Klasse von Quasiordnungen von besonderem Interesse, da die Klasse der regulären Wortsprachen bzw. der erkennbaren Pfeilsprachen in Bezug auf Wohlquasiordnungen definiert werden können, wie in den Kapiteln 3 und 4 gezeigt wird.

Definition 2.6 (Wohlquasiordnung)**.** *Eine Quasiordnung auf M wird* Wohlquasiordnung (WQO) *genannt, wenn für jede unendliche Folge $m_1, m_2, \ldots$ von Elementen aus M gilt, dass Indizes i, j existieren, so dass $0 < i < j$ und $m_i \sqsubseteq m_j$ ist.*

Außer der soeben vorgestellten gibt es noch weitere Definitionen, die eine Wohlquasiordnung charakterisieren. Diese sind aber zu der obigen Definition äquivalent, wie der folgende Satz zeigt:

Satz 2.7. *Sei $(M, \sqsubseteq)$ eine quasigeordnete Menge, dann sind die folgenden Bedingungen äquivalent:*

1. *$\sqsubseteq$ ist eine Wohlquasiordnung.*
2. *Die aufsteigende Kettenbedingung gilt für abgeschlossene Teilmengen von M (geordnet durch Mengeninklusion).*
3. *Jede unendliche Folge von Elementen aus M hat eine unendlich aufsteigende Teilfolge.*
4. *Es existieren weder eine unendliche, streng absteigende Folge in M, noch unendlich viele paarweise unvergleichbare Elemente aus M.*

Beweis. Siehe [14]. □

Mit Hilfe des obigen Satzes lassen sich die beiden folgenden Lemmata beweisen, die im weiteren Verlauf für Beweise benötigt werden:

Lemma 2.8. *Seien $\sqsubseteq_1, \sqsubseteq_2$ Quasiordnungen auf einer Menge M, so dass $\sqsubseteq_1 \subseteq \sqsubseteq_2$ gilt. Wenn $\sqsubseteq_1$ eine Wohlquasiordnung ist, dann ist auch $\sqsubseteq_2$ eine Wohlquasiordnung.*

Beweis. Da $\sqsubseteq_1$ eine Wohlquasiordnung ist, hat nach Satz 2.7 jede unendliche Folge $m_1, m_2, \ldots$ von Elementen aus M eine unendliche aufsteigende Teilfolge $n_1, n_2, \ldots$, das heißt es gilt $n_i \sqsubseteq_1 n_j$ für $i < j$. Wegen $\sqsubseteq_1 \subseteq \sqsubseteq_2$ folgt aber aus $n_i \sqsubseteq_1 n_j$ auch $n_i \sqsubseteq_2 n_j$. Damit existiert für jede unendliche Folge von Elementen aus M eine unendliche aufsteigende Teilfolge bzgl. $\sqsubseteq_2$. Damit ist $\sqsubseteq_2$ eine Wohlquasiordnung. □

Lemma 2.9. *Seien M und N Mengen sowie $\sqsubseteq_1$ und $\sqsubseteq_2$ Wohlquasiordnungen auf M bzw. N. Dann ist das direkte Produkt $\sqsubseteq_1 \times \sqsubseteq_2$, welches definiert ist durch*

$$(m_1, n_1) \sqsubseteq_1 \times \sqsubseteq_2 (m_2, n_2) \text{ gdw. } m_1 \sqsubseteq_1 m_2 \wedge n_1 \sqsubseteq_2 n_2,$$

eine Wohlquasiordnung auf der Menge $M \times N$.

Beweis. Sei $(m_1, n_1), (m_2, n_2), \ldots$ eine unendliche Folge von Elementen aus der Menge $M \times N$. Weil $\sqsubseteq_1$ eine Wohlquasiordnung ist, existiert nach Satz 2.7 eine Teilfolge $(m'_1, n'_1), (m'_2, n'_2), \ldots$, so dass für $i < j$ gilt: $m'_i \sqsubseteq_1 m'_j$. Da $\sqsubseteq_2$ ebenfalls eine Wohlquasiordnung ist, existiert wiederum eine Teilfolge $(m''_1, n''_1), (m''_2, n''_2), \ldots$ (von $(m'_1, n'_1), (m'_2, n'_2), \ldots$), so dass für $i < j$ gilt: $n''_i \sqsubseteq_2 n''_j$. Insgesamt gilt daher für $i < j$: $(m'_i, n''_i) \sqsubseteq_1 \times \sqsubseteq_2 (m'_j, n''_j)$. Also ist $\sqsubseteq_1 \times \sqsubseteq_2$ eine Wohlquasiordnung. □

2.3 Äquivalenzrelationen

Um nachfolgend einige Zusammenhänge zwischen Wohlquasiordnungen und Äquivalenzrelationen aufzeigen zu können, wird zunächst die Definition der Äquivalenz- und der Kongruenzrelation benötigt:

Definition 2.10 (Äquivalenzrelation, Kongruenzrelation)**.** *Eine* Äquivalenzrelation *ist eine zweistellige Relation $\sim$ auf einer Menge M, welche folgende Bedingungen erfüllt:*

- *$\sim$ ist* reflexiv*: Für alle $m \in M$ ist $m \sim m$,*
- *$\sim$ ist* symmetrisch*: Für alle $m_1, m_2 \in M$ folgt aus $m_1 \sim m_2$ auch $m_2 \sim m_1$,*
- *$\sim$ ist* transitiv*: Für alle $m_1, m_2, m_3 \in M$ folgt aus $m_1 \sim m_2$ und $m_2 \sim m_3$ auch $m_1 \sim m_3$.*

Falls $\sim$ außerdem monoton ist, wird $\sim$ Kongruenz(-relation) *genannt.*

Ist $\sim$ eine Äquivalenzrelation auf einer Menge M, so nennt man für ein $a \in M$ die Teilmenge

$$[a]_\sim = \{x \in M \mid x \sim a\}$$

die $\sim$-Äquivalenzklasse *von* a. *Ist aus dem Kontext klar, dass Äquivalenzklassen bezüglich* $\sim$ *gebildet werden, schreibt man statt* $[a]_\sim$ *kürzer* $[a]$. *Elemente einer Äquivalenzklasse werden ihre* Vertreter *oder* Repräsentanten *bezeichnet.*

Die Anzahl der verschiedenen Äquivalenzklassen von $\sim$, *also die maximale Anzahl von Elementen* $m_1, \ldots, m_n, \ldots$ *mit* $m_i \not\sim x_j$ *für* $i \neq j$, *wird mit* $Index(\sim)$ *bezeichnet. Falls* $Index(\sim) < \infty$ *gilt, hat* $\sim$ *einen* endlichen Index.

Zum Beispiel ist die *Identität auf* M

$$\mathsf{id}_M = \{(m, n) \mid m = n\}$$

eine Äquivalenzrelation, ebenso wie eine Quasiordnung $\sqsubseteq$ auf M eine Äquivalenzrelation definiert, falls für alle $m_1, m_2 \in M$ gilt:

$$m_1 \sqsubseteq m_2 \implies m_2 \sqsubseteq m_1.$$

Außerdem gilt für jede Quasiordnung $\sqsubseteq$, dass der Schnitt mit der inversen Quasiordnung, also $\sqsubseteq \cap \sqsubseteq^{-1}$ ebenfalls eine Äquivalenzrelation induziert.

Falls nun sowohl $\sqsubseteq$ als auch $\sqsubseteq^{-1}$ Wohlquasiordnungen sind, gilt für die induzierte Äquivalenzrelation das folgende Lemma:

Lemma 2.11. *Seien* $\sqsubseteq$ *eine Wohlquasiordnung auf einer Menge* M *und* $\sim$ *eine Äquivalenzrelation auf einer Menge* M *mit* $\sim = (\sqsubseteq \cap \sqsubseteq^{-1})$. *Dann hat* $\sim$ *einen endlichen Index, genau dann, wenn* $\sqsubseteq^{-1}$ *eine Wohlquasiordnung ist.*

Beweis.

(⇒) Sei $m_1, m_2, \ldots$ eine unendliche Folge von Elementen aus M. Dann existieren Indizes $0 < i < j$, so dass $m_i \sim m_j$ gilt. Dies folgt sofort aus der Tatsache, dass $\sim$ einen endlichen Index hat und daher nur endlich viele Äquivalenzklassen besitzt. Damit ist $\sim$ eine Wohlquasiordnung. Aus $m_i \sim m_j$ folgt umgehend $m_i \sqsubseteq^{-1} m_j$ und daher gilt $\sim \subseteq \sqsubseteq^{-1}$. Nach Lemma 2.8 ist auch $\sqsubseteq^{-1}$ eine Wohlquasiordnung.

(⇐) Angenommen sei, dass der Index von $\sim$ unendlich ist. Dann existiert eine unendliche Folge $m_1, m_2, \ldots$ von Elementen aus M, so dass $m_i \not\sim m_j$ für $i \neq j$. Dann muss nach Definition von $\sim$ aber $m_i \not\sqsubseteq m_j$ oder $m_i \not\sqsubseteq^{-1} m_j$ für $i \neq j$ gelten. Da sowohl $\sqsubseteq$ als auch $\sqsubseteq^{-1}$ Wohlquasiordnungen sind, kann es aber höchstens endlich viele paarweise nicht-vergleichbare Elemente in $\sqsubseteq$ und $\sqsubseteq^{-1}$ geben. Somit kann es im Widerspruch zu der Annahme keine solche Folge $m_1, m_2, \ldots$ geben.

□

Für die im nächsten bzw. übernächsten Kapitel erfolgende Charakterisierung der regulären Sprachen bzw. der erkennbaren Graphsprachen ist der Index einer Äquivalenzrelation auf den jeweiligen Sprachen entscheidend. Das nachfolgende Lemma liefert einen Zusammenhang zwischen dem Index einer Äquivalenzrelation und der Eigenschaft eine Wohlquasiordnung zu sein:

Lemma 2.12. *Eine Äquivalenzrelation* $\sim$ *auf einer Menge* M *hat einen endlichen Index genau dann, wenn* $\sim$ *eine Wohlquasiordnung auf der Menge* M *ist.*

Beweis.

(⇒) Sei $m_1, m_2, \ldots$ eine unendliche Folge von Elementen aus M. Dann existieren Indizes $0 < i < j$, so dass $m_i \sim m_j$ gilt. Dies folgt sofort aus der Tatsache, dass $\sim$ einen endlichen Index hat und daher nur endlich viele Äquivalenzklassen besitzt. Damit ist $\sim$ nach Satz 2.7 eine Wohlquasiordnung.

(⇐) Da $\sim$ eine Äquivalenzrelation ist, gilt $\sim = \sim^{-1}$ und daher auch $\sim = (\sim \cap \sim^{-1})$. Nach Lemma 2.11 hat $\sim$ daher einen endlichen Index.

□

Kapitel 3

Wortsprachen

In diesem Kapitel soll in die Theorie der (regulären) Wortsprachen eingeführt werden. Zunächst wird das Theorem hergeleitet, welches besagt, dass eine Wortprache genau dann regulär ist, wenn sie bezüglich einer monotonen Wohlquasiordnung nach oben abgeschlossen ist. Darauf aufbauend wird ein Algorithmus entwickelt, der automatisch überprüft, ob bestimmte (Zeichen-)Ersetzungsregeln invariant bezüglich einer regulären Sprache sind. Diese Ergebnisse werden im Anschluss an dieses Kapitel auf Pfeilsprachen verallgemeinert.

3.1 Theorie der regulären Sprachen

In diesem Abschnitt werden zunächst einige Grundbegriffe der Automatentheorie und der formalen Sprache erläutert, deren Kenntnis in Hinblick auf die weiteren Betrachtungen im Rahmen dieser Arbeit erforderlich ist. Anschließend werden in Anlehnung an [18, 8] verschiedene Charakterisierungen der Klasse der regulären Sprachen aufgezeigt.

Definition 3.1 (Alphabet, Wort, Formale Sprache)**.** *Sei* Σ *ein* Alphabet*, das heißt, eine Menge von Zeichen. Dann bezeichnet man mit* Σ^* *die Menge aller* Wörter*, das heißt, die Menge aller (endlichen) Zeichenketten mit Zeichen aus* Σ*. Das* leere Wort *(das Wort der Länge* 0*) wird mit* ε *bezeichnet. Die Menge aller nicht-leeren Wörter über* Σ *wird mit* Σ^+ *bezeichnet.*

Eine (formale) Sprache L *über* Σ *ist eine beliebige Teilmenge von* Σ^* *(*$L \subseteq \Sigma^*$*).*

Um die Klasse der regulären Sprachen definieren zu können, ist es zunächst nötig ein besonderes Automatenmodell einzuführen - den deterministischen endlichen Automaten.

Definition 3.2 (Deterministischer Endlicher Automat)**.** *Ein* (deterministischer) endlicher Automat (DEA) *ist ein 5-Tupel* $M = (S, \Sigma, \delta, s_0, F)$*, wobei*

- S *die Menge der* Zustände*,*
- Σ *das* Eingabealphabet *(mit* $S \cap \Sigma = \emptyset$*),*
- $\delta \colon S \times \Sigma \to S$ *die* Überführungsfunktion *(oder* Übergangsfunktion*),*
- $s_0 \in S$ *der* Startzustand *und*
- $F \subseteq S$ *die Menge der* Endzustände

ist. Dabei müssen S, Σ endliche Mengen sein.

Die von M akzeptiere Sprache *ist*

$$T(M) = \{x \in \Sigma^* \mid \hat{\delta}(s_0, x) \in F\},$$

dabei bezeichnet $\hat{\delta}$ die kanonische Erweiterung von δ von Einzelzeichen zu Wörtern, die wie folgt induktiv definiert ist:

$$\hat{\delta}(s, \varepsilon) := s$$
$$\hat{\delta}(s, aw) := \delta(\hat{\delta}(s, w), a)$$

Anschaulich gesprochen, ist ein deterministischer endlicher Automat so aufgebaut, dass er ein endliches Gedächtnis in Form der Zustände besitzt. Während der zeichenweisen Verarbeitung eines Wortes findet jeweils ein Zustandswechsel statt, der lediglich vom derzeitigen Zustand und dem aktuell gelesenem Zeichen abhängt. Sobald das gesamte Eingabewort verarbeitet wurde, wird geprüft, ob sich der Automat in einem der Endzustände befindet. Falls dies zutrifft, wird das zuvor eingelesene Wort akzeptiert, andernfalls wird das Wort abgelehnt. Die Sprache eines Automaten besteht somit aus allen Wörtern, die vom Automaten akzeptiert werden. Außerdem sind zwei deterministische endliche Automaten äquivalent zueinander, falls beide die gleiche Sprache akzeptieren.

Mit Hilfe dieses Automatenmodells lassen sich die regulären Sprachen als die von deterministischen, endlichen Automaten erkannten Sprachen einführen:

Definition 3.3 (Reguläre Sprache)**.** *Eine Sprache L über einem Alphabet Σ heißt* regulär*, wenn ein deterministischer endlicher Automat existiert, der die gleiche Sprache akzeptiert, das heißt es gilt: $T(M) = L$.*

Auf jeder formalen Sprachen lassen sich unter anderem die folgende Quasiordung und die folgende Äquivalenzrelation definieren:

Definition 3.4 (Myhill-Quasiordnung, Syntaktische Kongruenz)**.** *Sei die Sprache $L \subseteq \Sigma^*$ gegeben. Die* Myhill-Quasiordnung $\leq_L$ (relativ zu L) *ist folgendermaßen definiert: Seien $u, v, x, y \in \Sigma^*$ Wörter, dann gilt*

$$x \leq_L y \text{ gdw. } uxv \in L \implies uyv \in L.$$

Das Infimum der Myhill-Quasiordnung und der inversen Quasiordnung im Teilmengenverband, also $\leq_L \cap \leq_L^{-1}$, ergibt die syntaktische Kongruenz $\approx_L$ (auf der Sprache L)*. Diese ist definiert durch: Seien $u, v, x, y \in \Sigma^*$, dann gilt*

$$x \approx_L y \text{ gdw. } uxv \in L \iff uyv \in L.$$

Hieraus kann sofort verifiziert werden, dass die Myhill-Quasiordnung $\leq_L$ die Axiome einer Quasiordnung und die syntaktische Kongruenz $\approx_L$ die Axiome einer Kongruenzrelation erfüllen.

Wenn zwei Wörter $l, r \in \Sigma^*$ in Relation bezüglich der Myhill-Quasiordnung (relativ zu L) stehen, es ist also $l \leq_L r$, sagt man auch, dass L eine *Invariante (bezüglich der Regel $l \to r$)* ist. In diesem Kontext sind auch die folgenden Eigenschaften der Myhill-Quasiordnung von Bedeutung:

Satz 3.5. *Sei $L \subseteq \Sigma^*$ eine Sprache. Die Myhill-Quasiordnung $\leq_L$ (relativ zur Sprache L) ist monoton und L ist nach oben abgeschlossen bezüglich $\leq_L$.*

Beweis. Nachweis der Monotonie und der Abgeschlossenheit:

- Monotonie:
  Seien $x, y, z \in \Sigma^*$ beliebige Wörter mit $x \leq_L y$, dann gilt für alle $u, v \in \Sigma^*$:

  $$uxv \in L \implies uyv \in L.$$

  Dann folgt daraus aber auch, dass für alle $u, v \in \Sigma^*$ gilt: $uxzv \in L \implies uyzv \in L$. Dann ist $xz \leq_L yz$. Also ist $\leq_L$ rechts-monoton. Analog lässt sich die Links-Monotonie zeigen. Insgesamt erhält man so die Monotonie von $\leq_L$.

- Abgeschlossenheit:
  Seien $x \in L$ und $y \in \Sigma^*$ beliebige Wörter mit $x \leq_L y$, dann gilt nach der Definition von $\leq_L$ für alle $u, v \in \Sigma^*$:

  $$uxv \in L \implies uyv \in L.$$

  Dies gilt im Speziellen auch für $u = v = \varepsilon$, daraus folgt sofort:

  $$(x \in L \wedge x \leq_L y) \implies y \in L.$$

□

Den Zusammenhang zwischen der syntaktischen Kongruenz und der Myhill-Quasiordnung zeigt der folgende Satz.

Satz 3.6. *Die syntaktische Kongruenz $\approx_L$ auf einer Sprache $L \subseteq \Sigma^*$ besitzt einen endlichen Index genau dann, wenn die Myhill-Quasiordnung $\leq_L$ (relativ zu L) eine Wohlquasiordnung ist.*

Beweis.

(⇒) Nach Lemma 2.12 ist $\approx_L$ eine Wohlquasiordnung. Aus

$$\approx_L = \left(\leq_L \cap \leq_L^{-1}\right) \subseteq \leq_L$$

folgt damit nach Lemma 2.8, dass $\leq_L$ ebenfalls eine Wohlquasiordnung ist.

(⇐) Es sei angenommen, dass $\approx_L$ einen unendlichen Index besitzt. Dann existiert eine unendliche Folge von Wörtern $x_1, x_2, \ldots$ aus L, so dass $x_i \not\approx_L x_j$ für $i \neq j$ gilt. Da $\leq_L$ eine Wohlquasiordnung ist, existiert eine unendliche Teilfolge $y_1, y_2, \ldots$, so dass $y_i \leq_L y_j$ für $i < j$ gilt. Aus $y_i \not\approx_L y_j$ für $i < j$ folgt aber sogar $y_i <_L y_j$ [1] für $i < j$.

Es sei die Menge $M(x) = \{(u, v) \mid u, v \in \Sigma^* \text{ und } uxv \in L\}$. Nach Lemma 2.9 ist $\leq_L \times \leq_L$ eine Wohlquasiordnung auf $M(x)$. Aus $y_i \leq_L y_j$ für $i < j$ und der Monotonie von $\leq_L$ folgt dann für alle $u, v \in \Sigma^*$: $uy_iv \leq_L uy_jv$. Aus der Abgeschlossenheit von L folgt außerdem, dass $uy_iv \in L$ auch $uy_jv \in L$ impliziert. Zusammen mit $y_i \not\approx_L y_j$ für $i \neq j$ gilt daher, dass $M(y_i) \subsetneq M(y_j)$

[1] $x <_L y :\iff x \leq_L y \wedge x \not\leq_L^{-1} y$

für $i < j$ ist. Bei $M(y_i)$ handelt es sich also um eine unendlich streng aufsteigende Kette von Teilmengen von Σ^*.

Seien $u, u', v, v' \in \Sigma^*$ mit $u \leq_L u'$ und $v \leq_L v'$ beliebig gegeben, dann folgt aus der Monotonie von $\leq_L$ für alle y_i: $uy_i \leq_L u'y_i$ und $y_iv \leq_L y_iv'$. Nach Definition von $\leq_L \times \leq_L$ gilt somit $(u, v) \leq_L \times \leq_L (u', v')$. Da aus $(u, v) \in M(y_i)$ wegen $(u, v) \leq_L \times \leq_L (u', v')$ auch $(u', v') \in M(y_i)$ folgt, sind alle $M(y_i)$ abgeschlossen bezüglich $\leq_L \times \leq_L$. Damit erhält man eine unendlich streng aufsteigende Kette von Teilmengen von Σ^*, so dass die Teilmengen abgeschlossen bezüglich $\leq_L \times \leq_L$ sind. Daraus folgt aber, dass $\leq_L \times \leq_L$ keine Wohlquasiordnung ist. Dies ist ein Widerspruch, denn nach Voraussetzung ist $\leq_L$ eine Wohlquasiordnung und nach Lemma 2.9 damit auch $\leq_L \times \leq_L$.

□

Der folgende Satz, der auf die Arbeiten von Myhill [19] und Nerode [20] zurückgeht, charakterisiert eine reguläre Sprache L in Bezug auf Kongruenzen von endlichem Index:

Satz 3.7 (Satz von Myhill-Nerode)**.** *Die Sprache $L \subseteq \Sigma^*$ ist regulär genau dann, wenn L die Vereinigung von Äquivalenzklassen bezüglich einer Kongruenz $\sim$ auf Σ^* ist, die einen endlichen Index besitzt.*

Beweis.

(⇒) Sei $M = (S, \Sigma, \delta, s_0, F)$ der Automat, der die Sprache L akzeptiert. Sei außerdem die Relation $\equiv_M$ für Wörter $x, y \in \Sigma^*$ wie folgt definiert:

$$x \equiv_M y \text{ gdw. } \hat{\delta}(s, xv) \in F \iff \hat{\delta}(s, yv) \in F \quad \text{für alle } s \in S, v \in \Sigma^*.$$

Die Relation $\equiv_M$ ist aber eine Verfeinerung der Relation $\approx_L$. Dies kann wie folgt gezeigt werden: Seien $u, v, x, y \in \Sigma^*$ beliebige Wörter. Angenommen sei, dass $x \equiv_M y$ gelte, dann folgt

$$uxv \in L \iff \hat{\delta}(s_0, uxv) \in F \iff \hat{\delta}(s, xv) \in F \iff \hat{\delta}(s, yv) \in F \iff \hat{\delta}(s_0, uyv) \in F \iff uyv \in L,$$

wobei $s = \hat{\delta}(s_0, u)$ ist. Somit gilt $\equiv_M \subseteq \approx_L$. Daraus folgt

$$\begin{aligned} \mathit{Index}(\approx_L) &\leq \mathit{Index}(\equiv_M) \\ &= \text{Anzahl der Zustände, die von } s_0 \text{ aus erreichbar sind} \\ &\leq |S| \\ &< \infty. \end{aligned}$$

(⇐) Da $\approx_L$ einen endlichen Index besitzt, gilt für die Myhill-Nerode-Äquivalenz $\equiv_L$ wegen

$$\approx_L = \{(x, y) \mid \forall v, w \in \Sigma^* \colon vxw \in L \iff vyw \in L\} \subseteq \{(x, y) \mid \forall w \in \Sigma^* \colon xw \in L \iff yw \in L\} = \equiv_L,$$

dass der Index $\equiv_L$ kleiner als der Index von $\approx_L$ ist. Da $\approx_L$ aber einen endlichen Index besitzt, hat $\equiv_L$ ebenfalls einen endlichen Index. Somit ist L regulär.

□

Eine Verallgemeinerung des vorherigen Satzes liefert das nachfolgende Theorem, das auf Ehrenfeucht [8] zurückgeht.

Theorem 3.8 (Verallgemeinerter Satz von Myhill-Nerode)**.** *Sei $L \subseteq \Sigma^*$ eine Sprache. Dann sind die folgenden Aussagen äquivalent:*

(i) *L ist regulär,*

(ii) *L ist nach oben abgeschlossen bezüglich einer monotonen Wohlquasiordnung $\sqsubseteq$ auf Σ^*,*

(iii) *die Myhill-Quasiordnung $\leq_L$ (relativ zu L) ist eine Wohlquasiordnung.*

Beweis.

$(i) \Rightarrow (ii)$ Aus der Regularität von L folgt nach Satz 3.7, dass die syntaktische Kongruenz $\approx_L$ einen endlichen Index besitzt. Nach Lemma 2.12 ist $\approx_L$ eine Wohlquasiordnung. Aus der Definition von $\approx_L$ folgt außerdem sofort, dass Σ^* nach oben abgeschlossen bezüglich $\approx_L$ ist.

$(ii) \Rightarrow (iii)$ Aus der Monotonie von $\sqsubseteq$ folgt, dass für alle $u, v, x, y \in \Sigma^*$ gilt: $x \sqsubseteq y \implies uxv \sqsubseteq uyv$. Außerdem impliziert die Abgeschlossenheit von L, dass aus $uxv \in L$ auch $uyv \in L$ für alle $u, v, x, y \in \Sigma^*$ folgt. Nach Definition der Myhill-Quasiordnung gilt dann aber: $x \leq_L y$. Somit folgt aus $x \sqsubseteq y$ auch $x \leq_L y$ für alle $x, y \in \Sigma^*$. Nach Lemma 2.8 folgt aus der Tatsache, dass $\sqsubseteq$ eine Wohlquasiordnung ist, dass auch $\leq_L$ eine Wohlquasiordnung ist.

$(iii) \Rightarrow (i)$ Dies folgt sofort aus den Sätzen 3.6 und 3.7.

□

3.2 Überprüfung der Myhill-Quasiordnung

In diesem Abschnitt soll ein Algorithmus vorgestellt werden, der für zwei Wörter x und y entscheiden kann, ob diese in Relation bezüglich der Myhill-Quasiordnung $\leq_L$ (relativ zu einer regulären Sprache L) stehen. Die Arbeitsschritte des Algorithmus' sind dabei vergleichbar mit dem Ablauf des Algorithmus' für die Erzeugung des Minimalautomaten für eine reguläre Sprache.

Algorithmus 3.9.

- *Eingabe: $x, y \in \Sigma^*$ und ein DFA $M = (S, \Sigma, \delta, s_0, F)$ mit $T(M) = L$*
- *Ausgabe: „Ja", falls $x \leq_L y$ und „Nein", falls $x \not\leq_L y$*
- *Ablauf:*
  0. *Entferne unerreichbare Zustände des DFA (falls vorhanden). Bezeichne S' die Menge der erreichbaren Zustände.*
  1. *Stelle eine Tabelle aller Zustandspaare (s, s') mit $s, s' \in S$ auf.*
  2. *Markiere alle Paare (s, s') mit $s \in F$ und $s' \notin F$.*

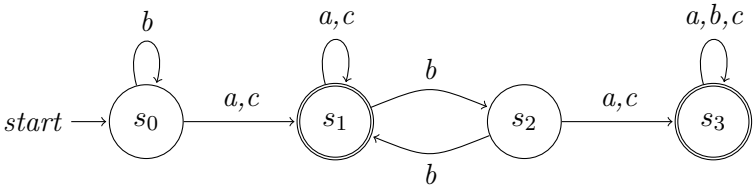
3. *Für jedes noch unmarkierte Paar (s, s') und jedes $a \in \Sigma$ teste, ob $(\delta(s, a), \delta(s', a))$ bereits markiert ist. Wenn ja: markiere auch (s, s').*
4. *Wiederhole den vorherigen Schritt so lange, bis sich keine Änderungen mehr ergeben.*
5. *Überprüfe nun, ob für jeden (erreichbaren) Zustand s_i mit $1 \leq i \leq |S'|$ das Paar $(\hat{\delta}(s_i, x), \hat{\delta}(s_i, y))$ unmarkiert ist. Falls alle Paare unmarkiert sind, gib „Ja“ aus, ansonsten gib „Nein“ aus.*

Zunächst werden im ersten dargestellten Schritt (Schritt null) alle unerreichbaren Zustände entfernt. Dies ist notwendig, da unerreichbare Zustände keine Auswirkungen auf die Myhill-Quasiordnung haben. Anschließend wird in den Schritten eins und zwei eine Tabelle aller Zustandspaare des Automaten erzeugt und initialisiert. Der Startwert für jedes Zustandspaar ist davon abhängig, ob der erste Zustand ein Endzustand und gleichzeitig der zweite Zustand kein Endzustand ist. Falls dies zutrifft, können zwei Wörter, die mit je einem der beiden Zuständen identifiziert werden, nicht in Relation bezüglich der Myhill-Quasiordnung zueinander stehen. Das erste Wort läge in diesem Fall in der Sprache, da der erreichbare Zustand nach Voraussetzung ein Endzustand war, während das zweite Wort jedoch nicht in der Sprache läge, da der erreichbare Zustand nach Vorraussetzung kein Endzustand war.

In Schritt drei werden die bisher noch unmarkierten Zustandspaar, die folglich noch in Relation bezüglich der Myhill-Quasiordnung stehen könnten, betrachtet. Diese werden darauf geprüft, ob beim Einlesen eines Alphabetzeichens ein markiertes Zustandspaar, das entsprechend nicht mehr in Relation steht, erreicht werden kann. Wenn ein solches markiertes Zustandspaar erreicht wird, kann auch das ursprüngliche Zustandspaar markiert werden, da die ursprünglichen Zustände durch das Einlesen dieses Alphabetzeichens getrennt werden können und somit nicht mehr in Relation bezüglich der Myhill-Quasiordnung stehen. Dieser Vorgang wird in Schritt vier für alle noch unmarkierten Zustandspaare wiederholt, bis entweder alle Zustandspaare markiert sind oder sich keine weiteren Änderungen mehr ergeben.

Schließlich wird in Schritt fünf für alle Zustände des Automaten überprüft, ob das jeweilige Zustandspaar, welches man durch Einlesen der beiden zu untersuchenden Wörter beginnend beim aktuellen Zustand erreicht, markiert ist oder nicht. Wenn die entsprechenden Zustandspaare aller Zustände unmarkiert sind, stehen die betrachteten Wörter in Relation bezüglich der Myhill-Quasiordnung, ansonsten nicht. Diesen Ablauf soll das folgende Beispiel veranschaulichen:

Beispiel 3.10. *Gegeben sei der folgende deterministische Automat:*



Zunächst wird eine Tabelle der Zustandspaare (s, s') angefertigt und gemäß Schritt zwei markiert:

s	s_0				
	s_1	*2*		*2*	
	s_2				
	s_3	*2*		*2*	
		s_0	s_1	s_2	s_3
		s'			

Wiederholte Anwendung von Schritt drei liefert:

s	s_0				
	s_1	*2*		*2*	
	s_2	*3.1*	*3.2*		
	s_3	*2*	*3.3*	*2*	
		s_0	s_1	s_2	s_3
		s'			

In Schritt fünf können nun unter anderem die folgenden Ergebnisse abgelesen werden:

$$a \not\leq_L b \qquad ab \not\leq_L b$$
$$a \not\leq_L ab \qquad ab \not\leq_L a$$
$$a \leq_L abc \qquad ab \leq_L abc$$

$$b \not\leq_L a \qquad abc \not\leq_L a$$
$$b \leq_L ab \qquad abc \not\leq_L b$$
$$b \leq_L abc \qquad abc \not\leq_L ab$$

Es bleibt noch zu zeigen, dass die Ergebnisse des Algorithmus' korrekt sind:

Satz 3.11 (Korrektheit des Algorithmus')**.** *Falls der Algorithmus „Ja“ ausgibt, so gilt $x \leq_L y$. Falls der Algorithmus „Nein“ ausgibt, gilt $x \not\leq_L y$.*

Beweis. Der Beweis der Korrektheit erfolgt in zwei Teilen.

1. Zuerst wird gezeigt, dass Folgendes für die Markierungen der Tabelle gilt: Das Paar (s, s') ist markiert genau dann, wenn ein $r \in \Sigma^*$ existiert, so dass gilt
$$\hat{\delta}(s, r) \in F \wedge \hat{\delta}(s', r) \notin F.$$
Der Beweis erfolgt durch Induktion über die Anzahl der Durchläufe der Tabelle (Schritte zwei bis vier)

   - Induktionsanfang: Falls (s, s') markiert ist, so wurde das Paar in Schritt zwei markiert und somit ist $s \in S'$ und $s' \notin F$. Wähle nun $r = \varepsilon$, dann gilt
$$\hat{\delta}(s, \varepsilon) = s \in F \wedge \hat{\delta}(s', \varepsilon) = s' \notin F.$$
   - Induktionsschritt: Falls (s, s') markiert ist, so wurde das Paar in Schritt drei markiert, weil Zustände s_1, s_2 und ein $a \in \Sigma$ existieren mit $s_1 = \delta(s, a)$ und $s_2 = \delta(s', a)$, so dass das Paar (s_1, s_2) nach Induktionsvoraussetzung bereits markiert ist. Also gilt für s_1, s_2, dass ein $r' \in \Sigma^*$ existiert mit
$$\hat{\delta}(s_1, r') \in F \wedge \hat{\delta}(s_2, r') \notin F.$$

Setze nun $r = ar'$, dann gilt

$$\hat{\delta}(s, ar') \in F \wedge \hat{\delta}(s', ar') \notin F.$$

2. Anschließend wird durch einen Widerspruchsbeweis gezeigt, dass die Ausgabe des Algorithmus' in beiden Fällen korrekt ist.

   a) Es sei angenommen, dass der Algorithmus „Ja" ausgibt, obwohl $x \not\leq_L y$ gilt. Dann existieren $\ell, r \in \Sigma^*$ mit

   $$(\ell xr \in L \wedge \ell yr \notin L) \quad \Longleftrightarrow \quad \Big(\hat{\delta}(s_0, \ell xr) \in F \wedge \hat{\delta}(s_0, \ell yr) \notin F\Big)$$

   bzw.

   $$\hat{\delta}(\underbrace{\hat{\delta}(s_0, \ell x)}_{=:s}, r) \in F \wedge \hat{\delta}(\underbrace{\hat{\delta}(s_0, \ell y)}_{=:s'}, r) \notin F.$$

   Dann wäre das Paar (s, s') aber durch den Algorithmus markiert worden (vgl. Teil 1) und der Algorithmus hätte somit im Widerspruch zur Annahme „Nein" ausgegeben.

   b) Angenommen, der Algorithmus gibt „Nein" aus, obwohl $x \leq_L y$ gilt. Dann gilt für alle $\ell, r \in \Sigma^*$: $\ell xr \in L \implies \ell yr \in L$. Da L regulär ist, kann Σ^* in endlich viele Äquivalenzklassen bzgl. $\equiv_L$ eingeteilt werden. Das bedeutet, es ist

   $$\Sigma^*/\equiv_L = \{[\ell_1], \ldots, [\ell_n]\}.$$

   Dann gilt für alle $\ell \in \Sigma^*$, dass ein $1 \leq i \leq n$ existiert, so dass $\ell \in [\ell_i]$ ist. Es genügt jeweils einen Repräsentanten aus jeder Äquivalenzklasse zu betrachten. Daher gilt

   $$\begin{aligned} &\forall \ell, r \in \Sigma^*\colon (\ell xr \in L \implies \ell yr \in L) \text{ gdw.} \\ &\quad \forall [\ell_i] \in \Sigma^*/\equiv_L, \forall r \in \Sigma^*\colon (\ell_i xr \in L \implies \ell_i yr \in L) \quad \text{falls } \ell \in [\ell_i]. \end{aligned}$$

   Da vor Beginn des eigentlichen Algorithmus' alle unerreichbaren Zustände aus der Zustandsmenge entfernt wurden, können die Äquivalenzklassen $[\ell_1], \ldots, [\ell_n]$ als die Zustände des Automaten M aufgefasst werden. Daher gilt für alle $1 \leq i \leq n$:

   $$\begin{aligned} &\forall [\ell_i] \in \Sigma^*/\equiv_L, \forall r \in \Sigma^*\colon (\ell_i xr \in L \implies \ell_i yr \in L) \text{ gdw.} \\ &\qquad \forall s_i \in S'\colon (\hat{\delta}(\hat{\delta}(s_i, x), r) \in F \implies \hat{\delta}(\hat{\delta}(s_i, y), r) \in F) \end{aligned}$$

   mit $s_i = \hat{\delta}(s_0, \ell_i)$. Dann sind die Paare $(\hat{\delta}(s_i, x), \hat{\delta}(s_i, y))$ aber vom Algorithmus nicht markiert worden (vgl. Teil 1). Damit gibt der Algorithmus aber im Widerspruch zur Annahme „Ja" aus.

□

Die Ergebnisse dieses Kapitels, im Speziellen der verallgemeinerte Satz von Myhill-Nerode und der Algorithmus zum Überprüfen von Invarianten, werden im nächsten Kapitel aufgegriffen und auf Pfeilsprachen erweitert.

Kapitel 4

Kategorientheorie und Pfeilsprachen

In diesem Kapitel soll der im vorherigen Kapitel entwickelte Algorithmus auf Pfeilsprachen erweitert werden. Dazu wird zunächst mit Hilfe der Kategorientheorie eine Charakterisierung der erkennbaren Pfeilsprachen entwickelt und anschließend ein Algorithmenschema vorgestellt. Dieses Schema dient im nächsten Kapitel als Grundlage zur Instanziierung eines Algorithmus' zum Überprüfen von Invarianten bei Graphsprachen.

4.1 Grundlagen der Kategorientheorie

Zunächst soll die Kategorientheorie kurz erläutert werden, soweit dies für das Verständnis der Konzepte der vorliegenden Arbeit notwendig ist. Innerhalb der Kategorientheorie werden die zu untersuchenden mathematischen Objekte dabei nicht durch ihre innere Struktur definiert, sondern vielmehr durch Relationen zwischen den einzelnen Objekten. Dies führt zu einer Abstraktion von konkreten Objekten, die in den verschiedenen Teildisziplinen der Mathematik untersucht werden, wodurch Ergebnisse einzelner Teilbereiche leicht auf andere Teilbereiche übertragen werden können. Daher kann die Kategorientheorie auch als eine Art „Metatheorie" betrachtet werden, welche die verschiedenen Bereiche der Mathematik miteinander verbindet. Für eine weitergehende Einführung sei auf [21] oder [17] verwiesen.

Die Definitionen und Sätze dieses Abschnittes sind aus [1, 9, 6] entnommen.

Definition 4.1 (Kategorie)**.** *Eine* Kategorie $\mathscr{C}$ *ist ein 4-Tupel*

$$\mathscr{C} = (O, \hom, \mathsf{id}, \circ)$$

bestehend aus

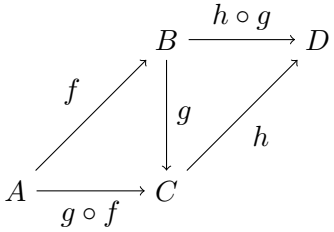
- *einer Klasse* O*, deren Elemente als* $\mathscr{C}$-Objekte A*,* B*,* C*, ... bezeichnet werden,*
- *je einer Klasse* $\hom(A, B)$ *für jedes Paar* (A, B) *von* $\mathscr{C}$*-Objekten, genannt* Hom-Klasse *mit* Definitionsbereich A *und* Zielbereich B*, deren Elemente* f*,* g*,* h*, ... als* $\mathscr{C}$-Pfeile (von A nach B) *bezeichnet werden,*
- *je einem Pfeil* id_A *von* A *nach* A *aus der Hom-Klasse* $\hom(A, A)$*, genannt* $\mathscr{C}$-Identität (auf A)*, für jedes* $\mathscr{C}$*-Objekt* A*,*
- *einer binären Operation* $\circ\colon \hom(A, B) \times \hom(B, C) \to \hom(A, C)$*, genannt* Komposition*, welche jedem Paar* (f, g) *von* $\mathscr{C}$*-Pfeilen von* A *nach* B *und von* B *nach* C *einen* $\mathscr{C}$*-Pfeil* $g \circ f$ *von* A *nach* C *zuordnet,*

so dass die folgenden Bedingungen erfüllt sind:

(C1) Assoziativität: Für alle $\mathscr{C}$-Pfeile $f \in \hom(A,B)$, $g \in \hom(B,C)$ und $h \in \hom(C,D)$ gilt

$$h \circ (g \circ f) = (h \circ g) \circ f,$$

das heißt

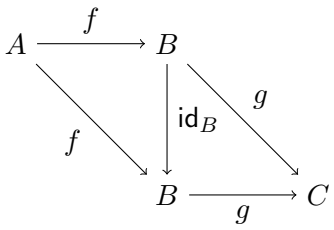


kommutiert stets.

(C2) Identität: Für alle $\mathscr{C}$-Identitäten id_B und für alle $\mathscr{C}$-Pfeile $f \in \hom(A,B)$ und $g \in \hom(B,C)$ gilt

$$\mathsf{id}_B \circ f = f \quad \wedge \quad g \circ \mathsf{id}_B = g,$$

das heißt,



kommutiert stets.

In der Kategorientheorie spielen *Diagramme* häufig eine wichtige Rolle und dienen als Mittel der Beweisführung und zur Darstellung von Zusammenhängen. Ein Diagramm (in einer Kategorie $\mathscr{C}$) ist dabei ein gerichteter Graph, dessen Knoten $\mathscr{C}$-Objekte und dessen Kanten $\mathscr{C}$-Pfeile sind. Zwei Beispiele für einfache Diagramme finden sich in obiger Definition.

Zur Vereinfachung der Notation wird außerdem die folgende Schreibweise eingeführt. Für eine beliebige Kategorie $\mathscr{C} = (O, \hom, \mathsf{id}, \circ)$ bezeichne

- $\mathcal{O}bj(\mathscr{C})$ alle $\mathscr{C}$-Objekte, also $\mathcal{O}bj(\mathscr{C}) := O$,
- $\mathcal{A}rr(\mathscr{C})$ die Klasse aller $\mathscr{C}$-Pfeile, das heißt die Vereinigung aller Klassen $\hom(A,B)$ in $\mathscr{C}$,
- $f\colon A \to B$ bzw. $A \xrightarrow{f} B$ den $\mathscr{C}$-Pfeil f von A nach B, also $f \in \hom(A,B)$,
- $f\,;g$ die Komposition von g hinter f, also $g \circ f$, für beliebige $\mathscr{C}$-Pfeile $f \in \hom(A,B)$ und $g \in \hom(B,C)$.

Falls klar ist, um welche Kategorie $\mathscr{C}$ es sich handelt, schreibt man auch kurz „Objekt“ statt „$\mathscr{C}$-Objekt“ bzw. „Pfeil“ statt „$\mathscr{C}$-Pfeil“. Außerdem wird $\mathscr{C}$ *lokal klein* genannt, falls für je zwei Objekte, die Klasse aller Pfeile zwischen diesen Objekten eine Menge ist. In diesem Fall spricht man auch von der *Hom-Menge* anstatt der Hom-Klasse.

Es ist zu beachten, dass die $\mathscr{C}$-Pfeile in obiger Definition nicht zwangsläufig strukturerhaltende Abbildungen sein müssen. Es ist nicht einmal erforderlich, dass es sich um Funktionen handelt, wie die folgenden beiden Beispiele für Kategorien zeigen:

Beispiel 4.2.

1. *Die Kategorie $\mathcal{Rel}$ ist die Kategorie, deren Objekte Mengen und deren Pfeile Relationen sind. Die Komposition von Pfeilen ist durch die Komposition von Relationen definiert.*

2. *Die Kategorie $\mathcal{Set}$ ist die Kategorie, deren Objekte Mengen und deren Pfeile Abbildungen sind. Die Komposition von Pfeilen ist durch die Komposition von Funktionen definiert.*

Definition 4.3 (Isomorphismus)**.** *Sei $\mathscr{C}$ eine Kategorie. Ein Pfeil $f \in \mathcal{A}rr(\mathscr{C})$ mit $f\colon A \to B$ wird* Isomorphismus *genannt, wenn ein Pfeil $g\colon B \to A$ existiert, so dass $f\,;g = \mathsf{id}_A$ und $g\,;f = \mathsf{id}_B$ ist. Die beiden Objekte A und B werden dann auch* isomorph *genannt.*

Wie bereits zu Beginn dieses Abschnittes erwähnt, dienen Kategorien zur Abstraktion von der inneren Struktur der mathematischen Objekte. Allerdings besitzt jede Kategorie wiederum eine eigene Struktur, die durch ihre Objekte und Pfeile festgelegt ist. Daher ist es naheliegend von dieser (kategoriellen) Struktur zu abstrahieren, indem jede Kategorie als Objekt einer größeren Kategorie betrachtet wird. Die Pfeile dieser „Kategorie der Kategorien“, die also Kategorien auf Kategorien abbilden, lassen sich wie folgt definieren:

Definition 4.4 (Funktor)**.** *Seien $\mathscr{C}$ und $\mathscr{D}$ Kategorien. Ein* Funktor $\mathcal{F}\colon \mathscr{C} \to \mathscr{D}$ von $\mathscr{C}$ nach $\mathscr{D}$ *ist ein Paar $(\mathcal{F}_{\mathcal{O}bj}, \mathcal{F}_{\mathcal{A}rr})$ von Abbildungen $\mathcal{F}_{\mathcal{O}bj}\colon \mathcal{O}bj(\mathscr{C}) \to \mathcal{O}bj(\mathscr{D})$, $\mathcal{F}_{\mathcal{A}rr}\colon \mathcal{A}rr(\mathscr{C}) \to \mathcal{A}rr(\mathscr{D})$, so dass gilt:*

1. $C \xrightarrow{f} C' \in \mathcal{A}rr(\mathscr{C}) \implies \mathcal{F}(C) \xrightarrow{\mathcal{F}(f)} \mathcal{F}(C') \in \mathcal{A}rr(\mathscr{D})$
2. *Identität.* $\mathcal{F}(\mathsf{id}_C) = \mathsf{id}_{\mathcal{F}(C)}$*, für alle* $C \in \mathcal{O}bj(\mathscr{C})$
3. *Komposition. Ist $g \circ f$ in $\mathscr{C}$ definiert, so gilt $\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f)$*

Ein Funktor bildet also Pfeile der einen Kategorie auf Pfeile der anderen Kategorie ab, so dass Identitäten und Kompositionen erhalten bleiben. Somit stellen Funktoren strukturerhaltende Abbildungen zwischen Kategorien dar.

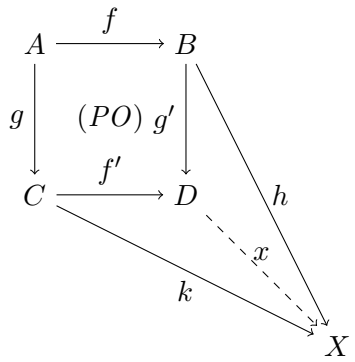
Da von der Struktur der Objekte einer Kategorie abstrahiert wird, können Objekte, die über spezielle Eigenschaften verfügen, nicht als solche gekennzeichnet werden. Um die Beispiele 4.2 aufzugreifen: Einem Objekt der Kategorie $\mathcal{Rel}$ oder $\mathcal{Set}$ ist also nicht ohne Weiteres anzusehen, ob es sich um die leere oder eine beliebige Menge handelt. Stattdessen müssen diese ausgezeichneten Objekte über

die Relation zu anderen Objekten – in Form von Pfeilen – definiert werden, so dass eine *universelle Eigenschaft* erfüllt ist, durch die diese ausgezeichneten Objekte eindeutig identifiziert werden können. Eine solche Möglichkeit besteht in der Konstruktion von so genannten Pushouts:

Definition 4.5 (Pushout)**.** *Seien eine Kategorie $\mathscr{C}$ und zwei Pfeile $f\colon A \to B$ und $g\colon A \to C$ gegeben. Ein* Pushout (D, f', g') (über f und g) *ist definiert durch*

- *ein* Pushout-Objekt D *und*
- *Pfeile $f'\colon C \to D$ und $g'\colon B \to D$ mit $f' \circ g = g' \circ f$,*

so dass die folgende universelle Eigenschaft erfüllt ist:
Für alle Objekte X und Pfeile $h\colon B \to X$ und $k\colon C \to X$ mit $k \circ g = h \circ f$ existiert ein eindeutiger Pfeil $x\colon D \to X$, so dass $x \circ g' = h$ und $x \circ f' = k$ gilt. Das heißt,



kommutiert stets.

Pushouts werden unter anderem für die Anwendung von so genannten *Graphtransformationen* mit kategorientheoretischen Mitteln benötigt. Dabei handelt es sich um das graphentheoretische Pendant zu Ableitungen im Falle von Wortsprachen, wobei während eines Transformationsschrittes ebenfalls eine Produktion in Form von *linker* und *rechter Seite* angewandt wird. Allerdings ist die Anwendung einer Graphproduktion ungleich schwieriger. Außer dem Suchen der linken und dem Ersetzen durch die rechte Seite ist darüber hinaus das Verbinden (auch *Verkleben* genannt) der (neuen) rechten Seite mit dem übrigen Graphen an bestimmten Stellen notwendig. Der Schritt der Verklebung kann kategorientheoretisch durch die Anwendung eines Pushouts ausgedrückt werden. Eine weitergehende Betrachtung dieses algebraischen Ansatzes zur Graphtransformation kann unter anderem in [9] gefunden werden.

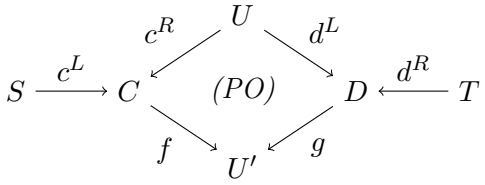
Als Nächstes soll der Begriff des Cospans und der damit verbundenen Cospan-Kategorie eingeführt werden. Diese stellen die Grundlage für Graphgrammatiken und den damit verbundenen Graphtransformationsregeln (s.o.) sowie für die Definition von (erkennbaren) Pfeilsprachen im nächsten Abschnitt dar.

Definition 4.6 (Cospan)**.** *Sei $\mathscr{C}$ eine Kategorie, in der Pushouts existieren. Ein* Cospan *c ist ein Paar von $\mathscr{C}$-Pfeilen*

$$c\colon S \xrightarrow{c^L} C \xleftarrow{c^R} T$$

mit demselben Zielbereich. Dabei werden S bzw. T als Quelle *(oder* inneres Interface*) bzw.* Ziel *(oder* äußeres Interface*) des Cospans c bezeichnet.*

Die Komposition von Cospans $c\colon S \to C \leftarrow U$ und $d\colon U \to D \leftarrow T$ ist durch das folgende, kommutierende Diagramm definiert:



wobei der mittlere Diamand ein Pushout und der komponierte Cospan durch:

$$(c\,;d)\colon S \xrightarrow{c^L\,;f} U' \xleftarrow{c^R\,;g} T.$$

definiert ist.

Definition 4.7 (Cospan-Kategorie)**.** *Sei $\mathscr{C}$ eine Kategorie, in der Pushouts existieren. Für zwei Cospans $c\colon S \to C \leftarrow T$ sowie $d\colon S \to D \leftarrow T$ soll $c \sim d$ genau dann gelten, wenn ein Isomorphismus $k\colon C \to D$ existiert, so dass $c^L\,;k = d^L$ und $c^R\,;k = d^R$ ist. Die Kategorie Cospan($\mathscr{C}$) wird definiert als die Kategorie, welche die gleichen Objekte wie die Kategorie $\mathscr{C}$ besitzt und deren Pfeile die $\sim$-Äquivalenzklassen von Cospans sind.*

4.2 Erkennbare Pfeilsprachen

In diesem Abschnitt wird zunächst der Begriff der Regularität von Wortsprachen auf so genannte Pfeilsprachen verallgemeinert – in diesem Zusammenhang spricht man auch von *erkennbaren Pfeilsprachen*. Die Definition erfolgt durch Automatenfunktoren, die eine Verallgemeinerung der endlichen Automaten darstellen, wie sie in [6] vorgestellt werden. Anschließend werden die Ergebnisse aus Kapitel 3 entsprechend für Pfeilsprachen erweitert, um die Grundlage zur Entwicklung eines Algorithmenschemas zum Überprüfen von Invarianten bei Pfeilsprachen zu schaffen.

Im Folgenden sei $\mathscr{C}$ eine feste Kategorie. Um Aussagen über Mengen und Sprachen machen zu können, sei außerdem vorausgesetzt, dass $\mathscr{C}$ lokal klein ist.

Definition 4.8 (Automatenfunktor, Erkennbarkeit)**.** *Sei $\mathscr{C}$ eine Kategorie. Dann bildet der Funktor $\mathcal{A}\colon \mathscr{C} \to \mathscr{Rel}$ jedes Objekt X aus $\mathscr{C}$ auf die endliche Menge $\mathcal{A}(X)$ und jeden Pfeil $f\colon X \to Y$ auf die Relation $\mathcal{A}(f) \subseteq \mathcal{A}(X) \times \mathcal{A}(Y)$ ab. Außerdem enthält jede Menge $\mathcal{A}(X)$ eine ausgezeichnete Menge $\mathrm{I}_X^{\mathcal{A}}$ von Startzuständen und eine ausgezeichnete Menge $\mathrm{F}_X^{\mathcal{A}}$ von Endzuständen als Teilmengen.*

Der Funktor $\mathcal{A}$ wird auch Automatenfunktor *genannt. Er ist* deterministisch*, wenn jede Relation $\mathcal{A}(f)$ eine Funktion ist und jede Startzustandsmenge $\mathrm{I}_X^{\mathcal{A}}$ genau ein Element enthält; dieses Element wird mit $\mathrm{i}_X^{\mathcal{A}}$ bezeichnet.*

Seien J, K zwei Objekte. Die (J,K)-Sprache $L_{J,K}(\mathcal{A})$ (von Pfeilen von J nach K) ist definiert als:

> *$f\colon J \to K$ ist in $L_{J,K}(\mathcal{A})$ enthalten genau dann, wenn $s \in \mathrm{I}_J^{\mathcal{A}}, t \in \mathrm{F}_K^{\mathcal{A}}$ existieren, welche in $\mathcal{A}(f)$ in Relation stehen.*

Eine Sprache $L_{J,K}$ von Pfeilen von J nach K ist erkennbar in $\mathscr{C}$*, wenn es die (J,K)-Sprache eines Automatenfunktors $\mathcal{A}\colon \mathscr{C} \to \mathscr{Rel}$ ist.*

Somit wird durch $\mathcal{A}(f)$ eine Transitionsrelation – ähnlich zu der von endlichen Automaten – definiert. Ein Unterschied besteht jedoch darin, dass der Automatenfunktor nicht nur eine einzelne Menge von Zuständen besitzt, sondern vielmehr für jedes Objekt eine eigene (endliche) Zustandsmenge existiert. Dies führt dazu, dass der Automatenfunktor im Allgemeinen aus unendlich vielen Zuständen besteht.

Die Funktoreigenschaft von $\mathcal{A}(f)$ garantiert, dass durch die Zerlegung von (komponierten) Pfeilen auf verschiedene Arten die Akzeptanz in keiner Weise beeinträchtigt wird. Im Falle von Wortsprachen tritt dieses Problem nicht auf, da Wörter im Wesentlichen auf lediglich eine Art und Weise in atomare Zeichen zerlegt werden können.

Beispiel 4.9. *Sei $M = (S, \Sigma, \delta, s_0, F)$ ein (deterministischer) endlicher Automat und sei Σ^* der freie Monoid über Σ, das heißt die Kategorie mit einem einzelnen Objekt X und den Wörtern über Σ als Pfeile von X nach X. Der Automatenfunktor $\mathcal{A}$, der die gleiche Sprache wie M erkennt, kann wie folgt konstruiert werden:*

- $\mathcal{A}(X) = S$ *mit* $\mathrm{i}_X^{\mathcal{A}} = s_0$ *und* $\mathrm{F}_X^{\mathcal{A}} = F$,
- $\mathcal{A}(w) = \{(s,t) \mid t \in \delta^*(s,w)\}$ *für* $w \in \Sigma^*$

Nachfolgend wird der Begriff der Quasi- bzw. Wohlquasiordnung sowie der Kongruenzrelation in geeigneter Weise auf Kategorien erweitert. Dazu wird eine Relation auf Kategorien als Familie von Relationen aufgefasst, wobei jede Relation der Familie auf genau einer Hom-Menge definiert ist. Dies bedeutet insbesondere, dass sich die betrachtete Relation zwischen zwei Pfeilen in der Regel ändert, sobald beide Pfeile von links bzw. rechts mit einem weiteren Pfeil verknüpft werden, da sich durch die Komposition unter Umständen der Definitions- bzw. Zielbereich der Pfeile ändert.

Definition 4.10 (Pfeilquasiordnung, Pfeilwohlquasiordnung)**.** *Sei $\mathscr{C}$ eine Kategorie. Sei eine Familie von Relationen*

$$\sqsubseteq_R = \{R_{J,K} \mid J, K \in \mathcal{O}bj(\mathscr{C})\}$$

gegeben, wobei die Elemente $R_{J,K}$ Quasiordnungen auf der Menge von $\mathscr{C}$-Pfeilen von J nach K sind, dann wird $\sqsubseteq_R$ Pfeilquasiordnung auf der Klasse von $\mathscr{C}$-Pfeilen *genannt. Falls alle $R_{J,K}$ Wohlquasiordnungen sind, wird die Familie $\sqsubseteq_R$ auch* Pfeilwohlquasiordnung *genannt. Eine Pfeilquasiordnung ist* links-monoton *bzw.* rechts-monoton*, falls für beliebige $\mathscr{C}$-Pfeile $a\colon H \to J$, $b, b'\colon J \to K$, $c\colon K \to M$ die Bedingung*

$$b\ R_{J,K}\ b' \implies (b\,;c)\ R_{J,M}\ (b'\,;c) \quad \textit{bzw.} \quad b\ R_{J,K}\ b' \implies (a\,;b)\ R_{H,K}\ (a\,;b')$$

erfüllt ist. Eine links- und rechts-monotone Pfeilquasiordnung heißt auch monoton.

Eine Teilklasse $S \subseteq \mathcal{A}rr(\mathscr{C})$ wird lokal abgeschlossen bezüglich $\sqsubseteq_R$ *genannt, falls für alle $\mathscr{C}$-Pfeile $a, b\colon J \to K$ folgende Bedingung erfüllt ist:*

$$a \in S \ \wedge\ a\ R_{J,K}\ b \implies b \in S.$$

Definition 4.11 (Pfeilkongruenz)**.** *Sei $\mathscr{C}$ eine Kategorie. Sei eine Familie von Relationen*

$$\sim_R = \{R_{J,K} \mid J, K \in \mathcal{O}bj(\mathscr{C})\}$$

gegeben, wobei die Elemente $R_{J,K}$ Äquivalenzrelationen auf der Menge von $\mathscr{C}$-Pfeilen von J nach K sind. Die Familie $\sim_R$ wird Pfeilkongruenz *genannt, falls für alle $\mathscr{C}$-Pfeile $a, a' \colon J \to K$, $b \colon K \to M$ gilt:*

$$\text{Aus } a \; R_{J,K} \; a', \text{ folgt } (a\,;b) \; R_{J,M} \; (a'\,;b).$$

Eine Kongruenz $\sim_R$ ist lokal endlich*, falls jede Relation $R_{J,K} \in \sim_R$ eine Äquivalenzrelation mit endlichem Index ist.*

Sei $a \colon J \to K$ ein $\mathscr{C}$-Pfeil. Im Folgenden sei mit $[\![a]\!]_{R_{J,K}}$ oder $[\![a]\!]_R$ – falls J und K aus dem Zusammenhang ersichtlich sind – die Kongruenzklasse von a bezeichnet. Statt $a \; R_{J,K} \; b$ kann auch kürzer $a \sim_R b$ geschrieben werden.

Im Weiteren stehen Pfeilsprachen im Vordergrund, die mit einem festen Objekt, zum Beispiel dem initialen Objekt, starten. Daher sei im restlichen Teil dieses Kapitels das Objekt J fixiert.

Analog zum Satz von Myhill-Nerode (Satz 3.7) im Fall von Wortsprachen kann auch die Klasse der erkennbaren Pfeilsprachen im Hinblick auf die Vereinigung von Kongruenzklassen einer endlichen (Pfeil-)Kongruenzrelation definiert werden.

Satz 4.12. *Sei $\mathscr{C}$ eine Kategorie. Eine Sprache $L_{J,K}$ ist erkennbar in $\mathscr{C}$ genau dann, wenn eine lokal endliche Pfeilkongruenz $\sim_R$ existiert, so dass $L_{J,K}$ die Vereinigung endlich vieler Äquivalenzklassen von $R_{J,K}$ ist.*

Beweis. Entnommen aus [6].

(⇒) Sei $\mathcal{A}$ ein Automatenfunktor, der die Sprache $L_{J,K}$ erkennt. Die Pfeilkongruenz $\sim_R = \{R_{M,N} \mid M, N \in \mathscr{C}\}$ lässt sich wie folgt konstruieren. Seien $a, b \colon M \to N$ beliebige $\mathscr{C}$-Pfeile. Es sei definiert:

$$a \; R_{J,K} \; b \text{ gdw. } \mathcal{A}(a) = \mathcal{A}(b).$$

Es ist klar, dass jede Äquivalenzrelation $R_{M,N}$ einen endlichen Index besitzt. Die Tatsache, das $\sim_R$ eine Pfeilkongruenz ist, folgt daher, dass $\mathcal{A}$ ein Funktor ist und somit die Komposition bewahrt. Daraus folgt dann umgehend

$$L_{J,K} = \{[\![a]\!]_R \mid a \colon J \to K \text{ und } \mathcal{A}(a)(\mathrm{I}_J^{\mathcal{A}}) \cap \mathrm{F}_K^{\mathcal{A}} \neq \emptyset\}.$$

(⇐) Diese Richtung des Beweises funktioniert nur für Mengen von Pfeilen, die bei dem festen Objekt J starten. Seien eine Pfeilkongruenz $\sim_R = \{R_{M,N} \mid M, N \in \mathcal{O}bj(\mathscr{C})\}$ und die Sprache $L_{J,K}$ gegeben. Der deterministische Automatenfunktor $\mathcal{A} \colon \mathscr{C} \to \mathcal{S}et$ ist wie folgt definiert:

- für ein Objekt H, sei Folgendes definiert: $\mathcal{A}(H) = \{[\![a]\!]_R \mid a \colon J \to H\}$,
- für einen Pfeil $b \colon H \to M$ und $[\![a]\!]_R \in \mathcal{A}(H)$ sei folgendes definiert: $\mathcal{A}(b)([\![a]\!]_R) = [\![a\,;b]\!]_R$.

Nun ist zu zeigen, dass $\mathcal{A}$ ein Funktor ist, was bedeutet, dass er Kompositionen und Identitäten bewahrt:

- *Komposition.* Seien $b\colon H \to M$ und $c\colon M \to N$ beliebige $\mathscr{C}$-Pfeile.

$$\begin{aligned}(\mathcal{A}(b)\,;\mathcal{A}(c))([\![a]\!]_{R_{J,H}}) &= \mathcal{A}(c)\left(\mathcal{A}(b)([\![a]\!]_{R_{J,H}})\right)\\ &= \mathcal{A}(c)([\![a\,;b]\!]_{R_{J,M}})\\ &= [\![a\,;b\,;c]\!]_{R_{J,N}}\\ &= \mathcal{A}(b\,;c)([\![a]\!]_{R_{J,H}})\end{aligned}$$

- *Identität.* Sei $a\colon J \to H$ ein beliebiger $\mathscr{C}$-Pfeil. Dann gilt:

$$\mathcal{A}(\mathsf{id}_H)([\![a]\!]_R) = [\![a\,;\mathsf{id}_H]\!]_R = [\![a]\!]_R = \mathsf{id}_{\mathcal{A}(H)}([\![b]\!]_R).$$

Zuletzt seien $\mathrm{S}_J^{\mathcal{A}}$ und $\mathrm{F}_K^{\mathcal{A}}$ wie folgt definiert:

$$\begin{aligned}\mathrm{S}_J^{\mathcal{A}} &= \{\mathsf{id}_J\},\\ \mathrm{F}_K^{\mathcal{A}} &= \{[\![a]\!]_R \mid a \in L_{J,K}\}.\end{aligned}$$

□

Wie schon in Kapitel 3 für Wortsprachen kann auch für Pfeilsprachen eine Art Myhill-Quasiordnung eingeführt werden. Diese bildet gleichfalls die Grundlage für das Algorithmenschema zum Überprüfen von Invarianten bei Pfeilsprachen, welches in Abschnitt 4.3 vorgestellt wird. Wie bereits oben erwähnt, ist dabei zu beachten, dass die Myhill-Pfeilquasiordnung eine Familie von Relationen darstellt und daher auf jeder Hom-Menge eine eigene Quasiordnung definiert ist.

Definition 4.13 (Myhill-Pfeilquasiordnung)**.** *Sei $\mathscr{C}$ eine Kategorie und $L_{J,K}$ eine Sprache von $\mathscr{C}$-Pfeilen. Dann wird die Familie von Quasiordnungen*

$$\leq_L = \{S_{M,N} \mid M, N \in \mathcal{O}bj(\mathscr{C})\}$$

Myhill-Pfeilquasiordnung (relativ zu $L_{J,K}$) *genannt, wenn für alle Quasiordnungen $S_{M,N}$ und beliebige $\mathscr{C}$-Pfeile $b, c\colon M \to N$ die folgende Bedingung erfüllt ist:*

$b\ S_{M,N}\ c$ gdw.

$$\forall a\colon J \to M, d\colon N \to K\colon\ ((a\,;b\,;d) \in L_{J,K} \implies (a\,;c\,;d) \in L_{J,K})\,.$$

Satz 4.14. *Sei $\mathscr{C}$ eine Kategorie und sei $L_{J,K}$ eine Sprache von $\mathscr{C}$-Pfeilen. Die Myhill-Pfeilquasiordnung (relativ zu $L_{J,K}$) ist monoton und die Sprache $L_{J,K}$ ist lokal nach oben abgeschlossen bezüglich $\leq_L$.*

Beweis. Nachweis der Monotonie und der Abgeschlossenheit:

- *Monotonie:* Seien $b, c\colon M \to N$ beliebige $\mathscr{C}$-Pfeile, für die $b \leq_L c$ gilt, dann folgt daraus für alle $\mathscr{C}$-Pfeile $a\colon J \to M$ und $d\colon N \to K$:

$$(a\,;b\,;d) \in L_{J,K} \implies (a\,;c\,;d) \in L_{J,K}.$$

Wähle nun $\mathscr{C}$-Pfeile $a''\colon J \to J'$, $a'\colon J' \to M$ bzw. $d'\colon N \to N'$, $d''\colon N' \to K$, so dass $a = (a''\,;a')$ bzw. $d = (d'\,;d'')$ gilt, insgesamt erhält man dadurch:

$$(a''\,;(a'\,;b\,;d')\,;d'') \in L_{J,K} \implies (a''\,;(a'\,;c\,;d')\,;d'') \in L_{J,K}.$$

Nach Definition von $\leq_L$ folgt daher: $(a'\,;b\,;d') \leq_L (a'\,;c\,;d')$.

- *Abgeschlossenheit:* Trivial, denn seien $a, b\colon J \to K$ beliebige $\mathscr{C}$-Pfeile, für die $a \leq_L b$ gilt, dann folgt aus $a \in L_{J,K}$ nach Definition von $\leq_L$ sofort auch $b \in L_{J,K}$.

□

Durch diese Einschränkung ist es nun möglich, die erkennbaren Pfeilsprachen in Bezug auf beliebige Pfeilquasiordnung – im Allgemeinen – und durch die Myhill-Quasiordnung – im Speziellen – zu charakterisieren, wie das folgende Theorem zeigt.

Theorem 4.15. *Sei $\mathscr{C}$ eine Kategorie und $L_{J,K}$ eine Sprache von $\mathscr{C}$-Pfeilen. Dann sind folgende Aussagen äquivalent:*

(i) *$L_{J,K}$ ist erkennbar in $\mathscr{C}$,*

(ii) *$L_{J,K}$ ist lokal nach oben abgeschlossen bezüglich einer monotonen Pfeilwohlquasiordnung $\sqsubseteq_R$,*

(iii) *die Myhill-Quasiordnung $\leq_L$ auf der Klasse von $\mathscr{C}$-Pfeile (relativ zu $L_{J,K}$) ist eine Pfeilwohlquasiordnung.*

Beweis. $(i) \Rightarrow (ii)$ Da $L_{J,K}$ erkennbar ist, existiert nach Satz 4.12 eine Pfeilkongruenz mit endlichem Index $\sim_R = \{R_{M,N} \mid M, N \in \mathcal{O}bj(\mathscr{C})\}$, so dass $L_{J,K}$ die Vereinigung endlich vieler Äquivalenzklassen von $R_{J,K}$ ist. Dass $\sim_R$ außerdem monoton ist, folgt unmittelbar aus der Definition von $\sim_R$. Es sei $\mathcal{A}$ der Automatenfunktor, der die Sprache $L_{J,K}$ erkennt, und für zwei Pfeile $c, d\colon M \to N$ sei angenommen, dass $c \sim_R d$ ist. Somit gilt nach Definition von $\sim_R$ auch $\mathcal{A}(c) = \mathcal{A}(d)$. Sei nun $a\colon L \to M$ ein beliebiger Pfeil, dann gilt

$$\mathcal{A}(a\,;c) = (\mathcal{A}(a)\,;\mathcal{A}(c)) = (\mathcal{A}(a)\,;\mathcal{A}(d)) = \mathcal{A}(a\,;d).$$

Daraus folgt aber unmittelbar $(a\,;c) \sim_R (a\,;d)$. Analog zeigt man die Rechts-Monotonie von $\sim_R$.

Da $\sim_R$ lokal endlich ist, hat jede Äquivalenzklasse $R_{M,N} \in \sim_R$ einen endlichen Index. Aus Lemma 2.12 folgt damit gleichzeitig, dass $R_{M,N}$ eine Wohlquasiordnung ist. Weil dies für alle $R_{M,N}$ mit $M, N \in \mathcal{O}bj(\mathscr{C})$ gilt, ist $\sim_R$ eine monotone Pfeilwohlquasiordnung.

$(ii) \Rightarrow (iii)$ Seien $b, c\colon M \to N$ beliebige $\mathscr{C}$-Pfeile. Aus der Monotonie von $\sqsubseteq_R$ folgt sofort:

$$b \sqsubseteq_{R_{M,N}} c \implies (a\,;b\,;d) \sqsubseteq_{R_{J,K}} (a\,;c\,;d) \quad \text{für alle } \mathscr{C}\text{-Pfeile } a\colon J \to M, d\colon N \to K.$$

Außerdem impliziert die Abgeschlossenheit von $L_{J,K}$, dass aus $a\,;b\,;d \in L_{J,K}$ auch $(a\,;c\,;d) \in L_{J,K}$ folgt. Nach Definition der Myhill-Quasiordnung auf der Klasse von $\mathscr{C}$-Pfeile gilt somit $b \leq_L c$. Nach Lemma 2.8 folgt aus der Tatsache, dass $\sqsubseteq_R$ eine Wohlquasiordnung ist, dass auch $\leq_L$ eine Wohlquasiordnung ist.

$(iii) \Rightarrow (i)$ Es sei angenommen, dass die Sprache $L_{J,K}$ nicht erkennbar ist. Dann kann die Pfeilkongruenz $\sim_L = (\leq_L \cap \leq_L^{-1}) = \{T_{J,K} \mid J, K \in \mathcal{O}bj(\mathscr{C})\}$ nach Satz 4.12 nicht lokal endlich sein. Das heißt, für ein $M \in \mathcal{O}bj(\mathscr{C})$ existiert eine Folge $a_1, a_2, \ldots$ von $\mathscr{C}$-Pfeilen von J nach M, so dass $\neg(a_i \; T_{J,M} \; a_j)$ für $i < j$ gilt. Da $\leq_L$ eine Pfeilwohlquasiordnung ist, existiert eine Wohlquasiordnung $S_{J,M} \in \leq_L$ mit $T_{J,M} \subseteq S_{J,M}$. Daher muss nach Satz 2.7 eine Teilfolge $b_1, b_2, \ldots$ von $a_1, a_2, \ldots$ existieren, so dass $b_i \; S_{J,M} \; b_j$ für $i < j$ gilt. Aus $\neg(b_i \; T_{J,M} \; b_j)$ folgt aber, dass $\neg(b_j \; S_{J,M} \; b_i)$ für $i < j$ gilt.

Für beliebige $\mathscr{C}$-Pfeile $a\colon J \to M$ sei $\mathcal{M}(a) = \{d\colon M \to K \mid (a\,;d) \in L_{J,K}\}$. Aus $b_i \; S_{J,M} \; b_j$ für $i < j$ und der Monotonie von $S_{J,M}$ folgt dann für alle $\mathscr{C}$-Pfeile $c\colon M \to K$: $(b_i\,;c) \; S_{J,K} \; (b_j\,;c)$. Aus der Abgeschlossenheit von $L_{J,K}$ folgt außerdem, dass $(b_i\,;c) \in L_{J,K}$ auch $(b_j\,;c) \in L_{J,K}$ impliziert. Zusammen mit $\neg(a_i \; T_{J,M} \; a_j)$ für $i < j$ gilt daher, dass $\mathcal{M}(b_i) \subsetneq \mathcal{M}(b_j)$ für $i < j$ ist. Bei $\mathcal{M}(b_i)$ handelt es sich also um eine unendliche, streng aufsteigende Kette von Teilklassen von $\mathcal{A}rr(\mathscr{C})$.

Seien beliebige $\mathscr{C}$-Pfeile $c, d\colon M \to K$ mit $c \; S_{M,K} \; d$ gegeben, dann folgt aus der Monotonie von $\leq_L$ für alle b_i: $(b_i\,;c) \; S_{J,K} \; (b_i\,;d)$. Daher sind alle $\mathcal{M}(b_i)$ abgeschlossen bezüglich $S_{M,K}$. Damit erhält man eine unendliche, streng aufsteigende, $S_{M,K}$-abgeschlossene Kette von Teilklassen von $\mathcal{A}rr(\mathscr{C})$. Daraus folgt aber nach Satz 2.7, dass $S_{M,K}$ keine Wohlquasiordnung ist. Daher ist $\leq_L$ keine Pfeilwohlquasiordnung, was ein ein Widerspruch zur Annahme ist.

□

4.3 Überprüfung der Myhill-Pfeilquasiordnung

In diesem Abschnitt soll in Anlehnung an den in Abschnitt 3.2 vorgestellten Algorithmus ein erweitertes Algorithmenschema vorgestellt werden. Dieses Schema stellt die Grundlage für einen Algorithmus dar, mit dem für zwei Pfeile b und c entschieden werden kann, ob diese in Relation bezüglich der Myhill-Pfeilquasiordnung $\leq_L$ (relativ zu einer Pfeilsprache $L_{J,K}$) stehen. Ein Algorithmus kann nicht direkt angegeben werden, da die Menge der $\mathscr{C}$-Pfeile selbst für lokal kleine Kategorien im Allgemeinen unendlich ist.

Algorithmus 4.16 (Schema)**.** *Seien ein deterministischer Automatenfunktor $\mathcal{A}$, der die Sprache $L_{J,K}$ erkennt, und zwei $\mathscr{C}$-Pfeile $b, c\colon M \to N$ gegeben.*

1. *Stelle für die Zustandsmenge $\mathcal{A}(N)$ eine Menge aller Zustandspaare (s, s') mit $s, s' \in \mathcal{A}(N)$ auf.*

2. *Markiere alle Paare (s, s') mit $s, s' \in \mathcal{A}(N)$ für die ein $\mathscr{C}$-Pfeil $d\colon N \to K$ existiert, so dass $\mathcal{A}(d)(s) \in \mathrm{F}_K^{\mathcal{A}}$ und $\mathcal{A}(d)(s') \notin \mathrm{F}_K^{\mathcal{A}}$ ist.*

3. *Überprüfe nun für jeden $\mathscr{C}$-Pfeil $a\colon J \to M$, ob im vorherigen Schritt das Zustandspaar $\big(\mathcal{A}(a\,;b)(\mathrm{i}_J^{\mathcal{A}}), \mathcal{A}(a\,;c)(\mathrm{i}_J^{\mathcal{A}})\big)$ markiert worden ist. Falls mindestens ein Paar markiert ist, stehen b und c nicht in Relation bezüglich der Myhill-Pfeilquasiordnung $\leq_L$, ansonsten stehen b und c in Relation.*

Zunächst wird im ersten dargestellten Schritt eine Tabelle aller Zustandspaare der Zustandsmenge $\mathcal{A}(N)$ erzeugt und initialisiert.

In Schritt zwei wird jedes Zustandspaar (s, s') aus $\mathcal{A}(N)$ markiert, dessen erster Zustand s von der Funktion $\mathcal{A}(d)$ auf einem Endzustand abgebildet wird und dessen zweiter Zustand s' von der Funktion $\mathcal{A}(d)$ nicht auf einen Endzustand abgebildet wird. Das heißt, es ist $\mathcal{A}(d)(s) = s_d \in \mathrm{F}_K^{\mathcal{A}}$ und $\mathcal{A}(d)(s') = s'_d \notin \mathrm{F}_K^{\mathcal{A}}$. Denn sei beliebiger Pfeil $f\colon J \to M$ gegeben, so dass

$$\mathcal{A}(f\,;b)(\mathrm{i}_J^{\mathcal{A}}) = s \text{ und } \mathcal{A}(f\,;c)(\mathrm{i}_J^{\mathcal{A}}) = s'$$

ist, dann gilt auch

$$\mathcal{A}(f\,;b\,;d)(\mathrm{i}_J^{\mathcal{A}}) = s_d \in \mathrm{F}_K^{\mathcal{A}} \text{ und } \mathcal{A}(f\,;c\,;d)(\mathrm{i}_J^{\mathcal{A}}) = s'_d \notin \mathrm{F}_K^{\mathcal{A}}.$$

Somit ist $(f\,;b\,;d) \in L_{J,K}$ und $(f\,;c\,;d) \notin L_{J,K}$, womit b nicht in Myhill-Pfeilquasiordnung zu c steht. Dieser Schritt wird für alle Zustandspaare und für alle Pfeile von N nach K wiederholt.

Schließlich wird in Schritt drei für alle Pfeile $a\colon J \to M$ überprüft, ob das Zustandspaar $\big(\mathcal{A}(a\,;b)(\mathrm{i}_J^{\mathcal{A}}), \mathcal{A}(a\,;c)(\mathrm{i}_J^{\mathcal{A}})\big)$ markiert ist. Falls das Paar für mindestens einen Pfeil markiert ist, muss nach Schritt zwei ein Pfeil $d\colon N \to K$ existieren, so dass $(a\,;b\,;d) \in L_{J,K}$ und $(a\,;c\,;d) \notin L_{J,K}$ ist. Daher kann in diesem Fall b nicht in Myhill-Pfeilquasiordnung zu c stehen. Nur wenn das Paar für alle Pfeile $a\colon J \to M$ unmarkiert ist, steht b in Myhill-Pfeilquasiordnung zu c.

Um das Algorithmenschema im nächsten Kapitel für Graphsprachen instanziieren zu können, werden die zulässigen Pfeile weiter eingeschränkt. Durch diese Änderung müssen nur noch endlich viele Pfeile in der entsprechenden Kategorie betrachtet werden, wodurch sichergestellt ist, dass der Algorithmus nach endlicher Zeit hält.

Kapitel 5

Graphsprachen

In diesem Kapitel wird die zuvor eingeführte Theorie auf Graphsprachen angewandt. Zunächst wird gezeigt, wie Graphen durch so genannte Graphoperationen konstruiert werden können. Anschließend wird das zuvor vorgestellte Algorithmenschema zur Überprüfung von Invarianten anhand eines konkreten Algorithmus' für Graphsprachen implementiert. Den Abschluss dieses Kapitels bildet die Definition eines Automatenfunktors, der dazu genutzt werden kann, Systeme, die durch Graphen modelliert werden können, auf bestimmte Eigenschaften zu überprüfen.

5.1 Graphsprachen und Graphoperationen

In diesem Abschnitt werden zunächst Hypergraphen sowie Hypergraphmorphismen als Verallgemeinerung von Graphen und Graphmorphismen eingeführt, welche die Grundlage für erkennbare (Hyper-)graphsprachen darstellen.

Definition 5.1 (Hypergraph, Hypergraphmorphismus)**.** *Sei eine Menge* Σ *von* Marken *gegeben. Ein* Hypergraph G *(nachfolgend auch* Graph *genannt) ist ein* 4*-Tupel* $\langle V_G, E_G, \mathsf{att}_G, \mathsf{lab}_G \rangle$ *mit* $V_G \cap E_G = \emptyset$ *bestehend aus*

- V_G *der Menge von* Knoten *von* G*,*
- E_G *der Menge von* Hyperkanten *von* G*,*
- $\mathsf{att}_G \colon E_G \to V_G^*$ *der* Verknüpfungsfunktion*,*
- $\mathsf{lab}_G \colon E_G \to \Sigma$ *der* Markierungsfunktion*,*

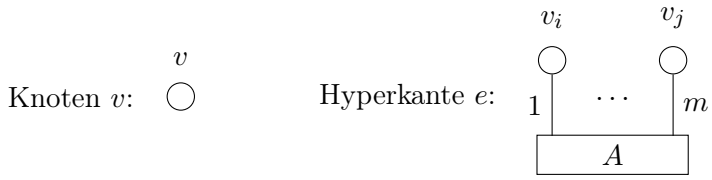
dabei bezeichnet V_G^* *die Menge der endlichen Folgen von Knoten aus* V_G *und* $\mathsf{att}_i(e)$ *das* i*-te Element dieser Folge. Außerdem wird mit* $\emptyset$ *der* leere Hypergraph *und mit* D_n *der* diskrete Graph *mit Knotenmenge* $V_n = \{v_0, \ldots, v_{n-1}\}$ *und leerer Hyperkantenmenge bezeichnet.*

Ein Hypergraphmorphismus f *zwischen zwei Hypergraphen* G *und* H *ist ein Paar* $\langle f_V, f_E \rangle$ *von Funktionen* $f_V \colon V_G \to V_H$ *und* $f_E \colon E_G \to E_H$*, welches die folgende Bedingung erfüllt:*

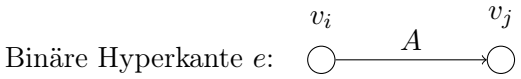
$$f_E \,;\mathsf{lab}_H = \mathsf{lab}_G \quad \textit{und} \quad f_E \,;\mathsf{att}_H = \mathsf{att}_G \,; f_V^*,$$

wobei f_V^* *die kanonische Erweiterung von* f_V *auf Folgen von Knoten bezeichnet.*

Zur graphischen Beschreibung von Hypergraphen wird die folgende Notation benutzt:



falls $|\mathsf{att}_G(e)| = m$, $\mathsf{att}_1(e) = v_i, \ldots, \mathsf{att}_m(e) = v_j$ und $\mathsf{lab}(e) = A$ ist. Im Falle von binären Hyperkanten wird auch die folgende Notation benutzt:



falls $|\mathsf{att}_G(e)| = 2$, $\mathsf{att}_1(e) = v_i, \mathsf{att}_2(e) = v_j$ und $\mathsf{lab}(e) = A$ ist.

Mit Hilfe der obigen Definitionen können im Folgenden die Kategorie *Graph* der Graphen und ihre entsprechende Cospan-Kategorie *Cospan*(*Graph*) definiert werden:

Definition 5.2 (Kategorie *Graph*)**.** *Die Kategorie Graph von Graphen ist definiert als die Kategorie, die endliche (Hyper-)Graphen als Objekte und Graphmorphismen als Pfeile besitzt.*

Die Kategorie *Cospan*(*Graph*) ist entsprechend definiert als die Kategorie, deren Objekte endliche (Hyper-)Graphen und deren Pfeile Cospans von Graphen mit diskreten Interfaces sind. Nach [6] ist dadurch die Erkennbarkeit von Graphsprachen nicht eingeschränkt.

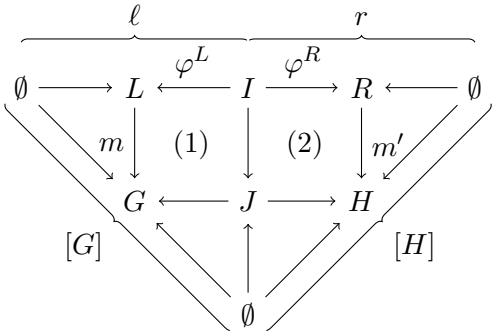
Einen Pushout über die Pfeile $f\colon U \to G$ und $g\colon U \to H$ in der Kategorie von Graphen erhält man, indem man die disjunkte Vereinigung von G und H bildet und durch die kleinste Äquivalenzrelation $\equiv$ (auf Knoten und Hyperkanten) faktorisiert, die $f(x) \equiv g(x)$ für alle (Knoten und Hyperkanten) x aus U erfüllt.

Zwischen der Cospan-Kategorie *Cospan*(*Graph*) und Graphgrammatiken besteht ein enger Zusammenhang. Eine Graphtransformationsregel (nach dem „Double Pushout"-Ansatz (DPO)) p ist ein Span[1] von Graphen der Form

$$p\colon L \xleftarrow{\varphi^L} I \xrightarrow{\varphi^R} R.$$

Mit I wird der *Kontext* bezeichnet, in dem die Ersetzung der *linken Seite* L durch die *rechte Seite* R stattfindet. Seien des Weiteren zwei Graphen G und H gegeben, dann werden mit $[G]\colon \emptyset \to G \leftarrow \emptyset$ und $[H]\colon \emptyset \to H \leftarrow \emptyset$ die Cospans bezeichnet, die aus den Graphen G bzw. H mit leerer Quelle und leerem Ziel bestehen. Eine *(direkte) Graphtransformation* von einem Graphen G zu einem Graphen H durch die Regel p, in Zeichen $[G] \Rightarrow_p [H]$, ist dann durch das folgende „Double Pushout Diagramm" gegeben, wobei (1) und (2) Pushouts sind:

[1] Spans sind das kategorientheoretische, duale Konzept zu Cospans. Siehe auch Definition 4.6.



Das bedeutet vereinfacht gesagt, dass durch die Anwendung der Graphtransformationsregel p auf den Graphen G

- die Knoten aus der Menge $m\left(V_L \setminus \varphi^L(V_I)\right)$ sowie die Hyperkanten aus der Menge $m\left(E_L \setminus \varphi^L(E_I)\right)$ aus G entfernt werden und
- die Knoten aus der Menge $m'\left(V_R \setminus \varphi^R(V_I)\right)$ sowie die Hyperkanten aus der Menge $m'\left(E_R \setminus \varphi^R(E_I)\right)$ zu G hinzugefügt werden.

Außerdem können die linke und die rechte Seite dieser Graphtransformationsregel als Cospans

$$\ell\colon \emptyset \to L \xleftarrow{\varphi^L} I \quad \text{bzw.} \quad r\colon \emptyset \to R \xleftarrow{\varphi^R} I$$

aufgefasst werden. Somit kann die Graphtransformationsregel $[G] \Rightarrow_p [H]$ genau dann angewendet werden, wenn ein Cospan $c\colon I \to J \leftarrow \emptyset$ existiert, so dass $[G] = \ell\,;c$ und $[H] = r\,;c$ ist. Eine weitergehende Einführung in die Theorie der Graphtransformation findet sich unter anderem in [9, 23].

Für den Rest dieses Kapitels sei die betrachtete Kategorie die Cospan-Kategorie *Cospan*(*Graph*). Mittels dieser Cospan-Kategorie lassen sich die erkennbaren Graphsprachen definieren, indem Graphen (ohne entsprechende Interfaces) als Cospans mit leeren Interfaces aufgefasst werden.

Definition 5.3 (Erkennbare Graphsprache)**.** *Eine Menge L von Graphen ist eine* erkennbare Graphsprache*, wenn*

$$L_{\emptyset,\emptyset} = \{[G]\colon \emptyset \to G \leftarrow \emptyset \mid G \in L\}$$

eine erkennbare Sprache in Cospan(*Graph*) *ist.*

Im Folgenden werden sechs atomare Graphoperationen definiert, mit deren Hilfe (Hyper)graphen in einem algebraischen Sinn verknüpft werden können. Diese Graphoperationen basieren ursprünglich auf Coucelles Graphalgebra [4] und wurden auf Cospans übertragen [6]. Die Graphoperationen dienen anschließend dazu, um Cospans der Form $c\colon \emptyset \to G \xleftarrow{inj.} D_n$ für einen beliebigen Graphen G und $n \in \mathbb{N}$ zu konstruieren.

Im Weiteren wird mit $G_1 \oplus G_2$ die *disjunkte Vereinigung zweier Graphen G_1 und G_2* bezeichnet. Außerdem ist $f \oplus g\colon G_1 \oplus G_2 \to H_1 \oplus H_2$ die *disjunkte Vereinigung zweier Graphmorphismen $f\colon G_1 \to H_1$ und $g\colon G_2 \to H_2$* definiert als

$$(f \oplus g)(v) = \begin{cases} f(v) & \text{falls } v \in G_1 \\ g(v) & \text{sonst} \end{cases}, \quad \text{für alle } v \in G_1 \oplus G_2$$

und als

$$(f \oplus g)(e) = \begin{cases} f(e) & \text{falls } e \in G_1 \\ g(e) & \text{sonst} \end{cases}, \quad \text{für alle } e \in G_1 \oplus G_2.$$

Mit $\mathbb{N}_n$ wird die Menge $\{0, \ldots, n-1\}$ bezeichnet.

Definition 5.4 (Atomare Graphoperationen)**.** *Sei ein Cospan* $c\colon \emptyset \to G \leftarrow D_n$ *gegeben.*

- **Restriktion des äußeren Interfaces***: Sei ein Pfeil*

  $$\rho\colon D_{n-1} \to D_n, \quad v_i \mapsto v_i$$

  zwischen zwei diskreten Graphen gegeben. Dann wird der Cospan res_n *definiert durch*

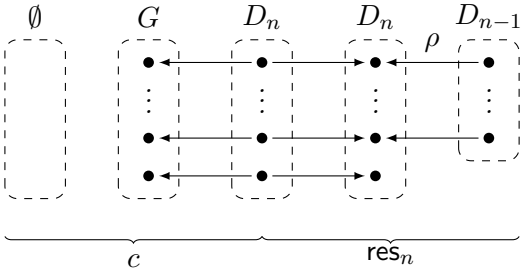
  $$\mathsf{res}_n\colon D_n \xrightarrow{\mathsf{id}_{D_n}} D_n \xleftarrow{\rho} D_{n-1}.$$

  Die Restriktion des äußeren Interfaces von c *ist dann die Komposition von* c *mit* res_n*, also*

  $$(c\,;\mathsf{res})\colon \emptyset \xrightarrow{c^L} G \xleftarrow{\rho\,;\,c^R} D_{n-1}.$$

  Der Cospan res_n *verschattet somit den letzten Knoten (entsprechend der Aufzählung der Knotenmenge) des äußeren Interfaces.*

  $(c\,;\mathsf{res}_n) =$

- **Permutation des äußeren Interfaces***: Seien eine Permutation*

  $$\pi\colon \mathbb{N}_n \to \mathbb{N}_n, \quad i \mapsto \begin{cases} i+1, & \textit{falls } i < n \\ 0, & \textit{falls } i = n \end{cases}$$

  und ein Pfeil

  $$\sigma\colon D_n \to D_n, \quad v_i \mapsto v_{\pi(i)}$$

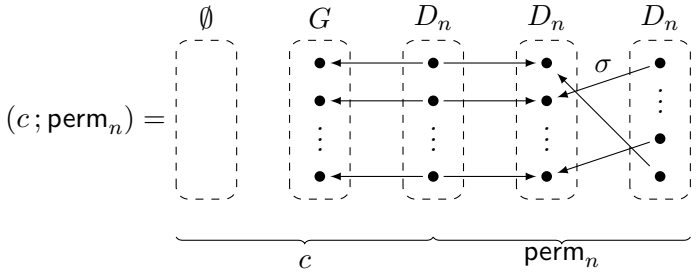
  zwischen zwei diskreten Graphen gegeben. Dann wird der Cospan perm_n *definiert durch*

  $$\mathsf{perm}_n\colon D_n \xrightarrow{\mathsf{id}_{D_n}} D_n \xleftarrow{\sigma} D_n.$$

  Die Permutation des äußeren Interfaces von c *ist dann die Komposition von* c *mit* perm_n*, also*

  $$(c\,;\mathsf{perm}_n)\colon \emptyset \xrightarrow{c^L} G \xleftarrow{\sigma\,;\,c^R} D_n.$$

  Der Cospan perm_n *permutiert die Knoten des äußeren Interfaces so, dass jeder Knoten auf seinen Nachfolgeknoten (bezüglich der Aufzählung der Knotenmenge) abgebildet wird.*



- **Transposition des äußeren Interfaces**: *Seien eine Transposition*

$$\tau\colon \mathbb{N}_n \to \mathbb{N}_n, \quad i \mapsto \begin{cases} 1, & \textit{falls } i = 0 \\ 0, & \textit{falls } i = 1 \\ i, & \textit{sonst} \end{cases}$$

und ein Pfeil

$$\sigma\colon V_n \to V_n, \quad v_i \mapsto v_{\tau(i)}$$

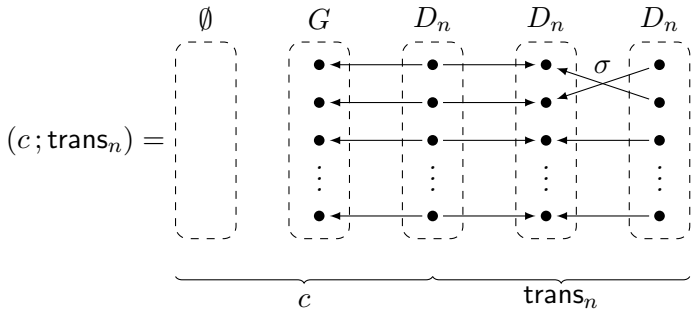
zwischen zwei diskreten Graphen gegeben. Dann wird der Cospan trans_n *definiert durch*

$$\mathsf{trans}_n\colon D_n \xrightarrow{\mathsf{id}_{D_n}} D_n \xleftarrow{\sigma} D_n.$$

Die Transposition des äußeren Interfaces von c *ist dann die Komposition von* c *mit* trans_n*, also*

$$(c\,;\mathsf{trans}_n)\colon \emptyset \xrightarrow{c^L} G \xleftarrow{\sigma\,;c^R} D_n.$$

Der Cospan trans_n *bildet die Knoten des äußeren Interfaces so ab, dass lediglich die ersten beiden Knoten (bezüglich der Aufzählung der Knotenmenge) untereinander transponiert werden.*



- **Fusion zweier Knoten des äußeren Interfaces**: *Es sei* $n > 1$ *und außerdem seien eine Äquivalenzrelation* $\theta = \mathsf{id}_{V_n} \cup \{(v_0, v_1), (v_1, v_0)\}$*, ein Pfeil* θ_{map}*, der jeden Knoten aus* D_n *auf seine* θ*-Äquivalenzklasse abbildet, sowie ein Pfeil*

$$\varphi\colon D_{n-1} \to D, \quad v_i \mapsto [\![v_{i+1}]\!]_\theta,$$

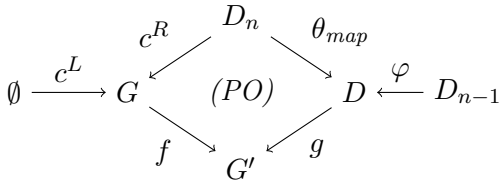
wobei D *der diskrete Graph mit der Knotenmenge* $\{[\![v]\!]_\theta \mid v \in V_n\}$ *ist, gegeben. Dann wird der Cospan* fuse_n *definiert durch*

$$\mathsf{fuse}_n\colon D_n \xrightarrow{\theta_{map}} D \xleftarrow{\varphi} D_{n-1}.$$

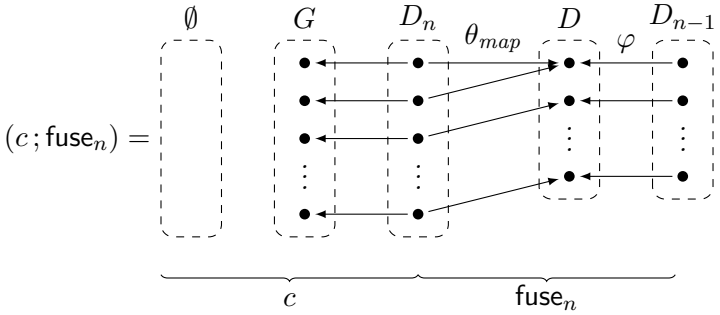
Die Fusion zweier Knoten des äußeren Interfaces von c *ist dann die Komposition von* c *mit* fuse_n*, also*

$$(c\,;\mathsf{fuse}_n)\colon \emptyset \xrightarrow{c^L\,;\,f} G' \xleftarrow{\varphi\,;\,g} D_{n-1}$$

wobei die Cospan-Komposition wie folgt aussieht:



Der Cospan fuse_n *verklebt die ersten beiden Knoten (bezüglich der Aufzählung der Knotenmenge) des äußeren Interfaces und entfernt anschließend den zweiten Knoten aus dem äußeren Interface.*



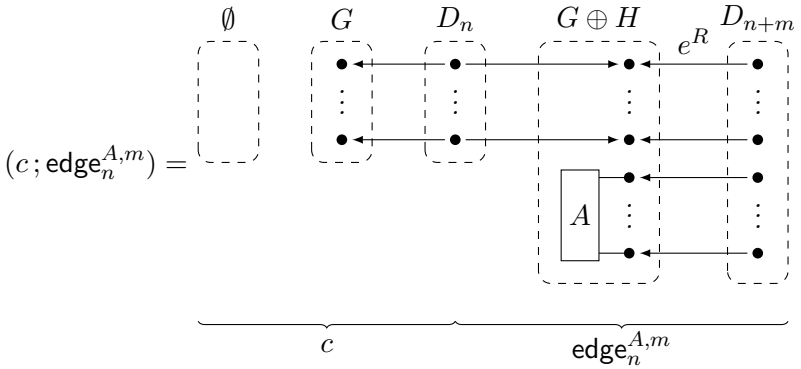
- **Disjunkte Vereinigung mit einer Hyperkante***: Seien eine Kantenbeschriftung* $A \in \Sigma$*,* $m \in \mathbb{N}$ *sowie ein Hypergraph* H*, der aus* n *isolierten Knoten und einer einzelnen* m*-stelligen Hyperkante* h *besteht, die mit* A *beschriftet ist, gegeben. Dann wird der Cospan* $\mathsf{edge}_n^{A,m}$ *definiert durch*

$$\mathsf{edge}_n^{A,m}\colon D_n \to D_n \oplus H \xleftarrow{e^R} D_{n+m}$$

mit $e^R = \mathsf{id}_{D_n} \oplus \eta$ *und* $\eta(v_i) = \mathsf{att}_{i-n+1}(h)$ *für* $n \leq i < m$*. Die* disjunkte Vereinigung von c mit einer Hyperkante h *ist dann die Komposition von* c *mit* $\mathsf{edge}_n^{A,m}$*, also*

$$(c\,;\mathsf{edge}_n^{A,m})\colon \emptyset \to G \oplus H \xleftarrow{c^R \oplus \eta} D_{n+m}.$$

Der Cospan $\mathsf{edge}_n^{A,m}$ *fügt dem Graphen* G *eine isolierte* m*-stellige mit* A *beschriftete Hyperkante hinzu und erweitert das äußere Interface so, dass die* m *letzten Knoten (bezüglich der Aufzählung der Knotenmenge) auf die* m *Knoten der* A*-Hyperkante abgebildet werden.*



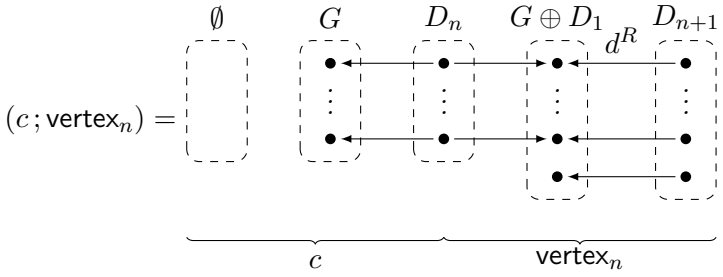
- **Disjunkte Vereinigung mit einem Knoten**: *Der Cospan* vertex_n *wird definiert durch*

$$\mathsf{vertex}_n \colon D_n \to D_{n+1} \xleftarrow{d^R} D_{n+1}$$

mit $d^R = \mathsf{id}_{D_n} \oplus i$ *und* $i(v_n) = v_n$. *Die* disjunkte Vereinigung von c mit einem isolierten Knoten v *ist dann die Komposition von* c *mit* vertex_n, *also*

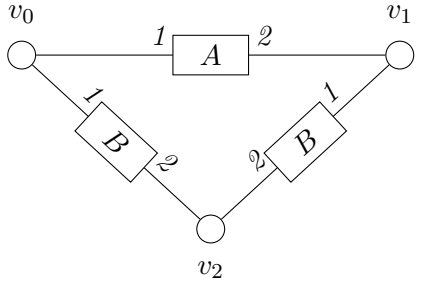
$$(c\,;\mathsf{vertex}_n) \colon \emptyset \to G \oplus D_1 \xleftarrow{c^R \oplus i} D_{n+1}.$$

Der Cospan vertex_n *fügt dem Graphen* G *einen isolierten Knoten hinzu und erweitert das äußere Interface so, dass der letzte Knoten des äußeren Interfaces (bezüglich der Aufzählung der Knotenmenge) auf den neuen Knoten abgebildet wird.*



Wie mittels dieser Graphoperationen Graphen und Cospans von Graphen konstruiert werden können, zeigt das nachfolgende Beispiel.

Beispiel 5.5. *Sei der Graph* G *wie folgt gegeben:*



Dann kann der Cospan $[G]\colon \emptyset \to G \leftarrow \emptyset$ *durch die Komposition der folgenden Graphoperationen konstruiert werden:*

$$\mathsf{edge}_0^{B,2}\,;\mathsf{edge}_2^{B,2}\,;\mathsf{trans}_4\,;\mathsf{perm}_4\,;\mathsf{fuse}_4\,;\mathsf{edge}_3^{A,2}\,;\mathsf{perm}_5\,;\mathsf{perm}_5\,;\mathsf{perm}_5\,;\\ \mathsf{fuse}_5\,;\mathsf{perm}_4\mathsf{res}_4\,;\mathsf{perm}_3\,;\mathsf{fuse}_3\,;\mathsf{res}_2\,;\mathsf{res}_1.$$

Im Weiteren soll gezeigt werden, dass durch die gegeben Graphoperationen alle Cospans der Form $c\colon \emptyset \to G \xleftarrow{inj.} D_n$ für einen beliebigen Graphen G und beliebiges $n \in \mathbb{N}$ konstruiert werden können. Der Beweis erfolgt in mehreren Schritten. Dazu werden die beiden folgenden Lemmata benötigt:

Lemma 5.6. *Sei* π *eine Permutation einer* n*-elementigen Menge, dann gilt:*

1. π *kann als Komposition von Zykeln dargestellt werden.*
2. π *kann als Komposition von Transpositionen dargestellt werden.*

Beweis.

1. Siehe [2, Kapitel 6.6].
2. Siehe [24, Kapitel 6.4].

□

Nach Lemma 5.6 kann jede Permutation π auf die Form

$$\pi = (x_1x_2)\,;\ldots;(x_{n-1}x_n)$$

gebracht werden. Daher genügt es nachfolgend zu zeigen, dass mit den in Definition 5.4 vorgestellten Operationen jede Transposition konstruiert werden kann.

Lemma 5.7. *Jede Transposition* $\tau = (v_iv_j)$ *zweier Knoten* v_i, v_j *für* $1 \leq i < j \leq n$ *kann durch Komposition der Graphoperationen* perm_n *und* trans_n *konstruiert werden.*

Beweis. Es ist

$$\begin{aligned}
\tau &= (v_iv_j)\\
&= \begin{pmatrix} v_1 & \ldots & v_{i-1} & v_i & v_{i+1} & \ldots & v_{j-1} & v_j & v_{j+1} & \ldots & v_n \\ v_1 & \ldots & v_{i-1} & v_j & v_{i+1} & \ldots & v_{j-1} & v_i & v_{j+1} & \ldots & v_n \end{pmatrix}\\
&= \begin{pmatrix} v_1 & \ldots & v_j & v_{j+1} & \ldots & v_n \\ v_j & \ldots & v_n & v_1 & \ldots & v_{j-1} \end{pmatrix};\\
&\quad \begin{pmatrix} v_1 & v_2 & \ldots & v_{j-i} & v_{n-i+1} & \ldots & v_n \\ v_1 & v_{n-j+i+2} & \ldots & v_n & v_{n-j+2} & \ldots & v_{n-j+i+1} \end{pmatrix};\\
&\quad \begin{pmatrix} v_1 & v_2 & \ldots & v_n \\ v_n & v_1 & \ldots & v_{n-1} \end{pmatrix};\\
&\quad \begin{pmatrix} v_1 & v_2 & \ldots & v_{i+2} & v_{i+3} & \ldots & v_n \\ v_1 & v_{n-i} & \ldots & v_n & v_2 & \ldots & v_{n-i-1} \end{pmatrix};\\
&\quad \begin{pmatrix} v_1 & \ldots & v_{j-1} & v_j & \ldots & v_n \\ v_{n-j+2} & \ldots & v_n & v_1 & \ldots & v_{n-j+1} \end{pmatrix}
\end{aligned}$$

$$= \underbrace{\begin{pmatrix} v_1 & v_2 & \dots & v_n \\ v_n & v_1 & \dots & v_{n-1} \end{pmatrix}; \dots ; \begin{pmatrix} v_1 & v_2 & \dots & v_n \\ v_n & v_1 & \dots & v_{n-1} \end{pmatrix}}_{(n-j+1)\text{-mal}};$$

$$\begin{pmatrix} v_1 & v_2 & \dots & v_n \\ v_n & v_1 & \dots & v_{n-1} \end{pmatrix}; \begin{pmatrix} v_1 & v_2 & v_3 & \dots & v_n \\ v_2 & v_1 & v_3 & \dots & v_n \end{pmatrix};$$

$$\dots; \qquad (j-i-1)\text{-mal}$$

$$\begin{pmatrix} v_1 & v_2 & \dots & v_n \\ v_n & v_1 & \dots & v_{n-1} \end{pmatrix}; \begin{pmatrix} v_1 & v_2 & v_3 & \dots & v_n \\ v_2 & v_1 & v_3 & \dots & v_n \end{pmatrix};$$

$$\begin{pmatrix} v_1 & v_2 & v_3 & \dots & v_n \\ v_2 & v_1 & v_3 & \dots & v_n \end{pmatrix};$$

$$\begin{pmatrix} v_1 & v_2 & \dots & v_n \\ v_n & v_1 & \dots & v_{n-1} \end{pmatrix}; \begin{pmatrix} v_1 & v_2 & v_3 & \dots & v_n \\ v_2 & v_1 & v_3 & \dots & v_n \end{pmatrix};$$

$$\dots; \qquad (n+i-(j+1))\text{-mal}$$

$$\begin{pmatrix} v_1 & v_2 & \dots & v_n \\ v_n & v_1 & \dots & v_{n-1} \end{pmatrix}; \begin{pmatrix} v_1 & v_2 & v_3 & \dots & v_n \\ v_2 & v_1 & v_3 & \dots & v_n \end{pmatrix};$$

$$\underbrace{\begin{pmatrix} v_1 & v_2 & \dots & v_n \\ v_n & v_1 & \dots & v_{n-1} \end{pmatrix}; \dots ; \begin{pmatrix} v_1 & v_2 & \dots & v_n \\ v_n & v_1 & \dots & v_{n-1} \end{pmatrix}}_{(j-1)\text{-mal}}$$

$$= \mathsf{perm}_n^{n-j+1} ; (\mathsf{perm}_n ; \mathsf{trans}_n)^{j-i-1} ; \mathsf{perm}_n ; (\mathsf{perm}_n ; \mathsf{trans}_n)^{n+i-(j+1)} ; \mathsf{perm}_n^{j-1}$$

□

Eine entscheidende Grundlage zur Instanziierung des Algorithmenschemas 4.3 im nächsten Abschnitt liefert der folgende Satz:

Satz 5.8. *Seien ein beliebiger Graph G und $n \in \mathbb{N}$ gegeben. Falls der Cospan $c\colon \emptyset \to G \xleftarrow{inj.} D_n$ durch Komposition von Cospans $c_1\colon \emptyset \to G_1 \xleftarrow{inj.} D_m$ und $c_2\colon D_m \to G_2 \xleftarrow{inj.} D_n$ entsteht, also $c = c_1 ; c_2$, dann existieren zwei Folgen von Graphoperationen $\mathrm{op}_1, \dots, \mathrm{op}_i$ und $\mathrm{op}_{i+1}, \dots, \mathrm{op}_j$, so dass c_1 als Komposition von Graphoperation dargestellt und c_1 zu c fortgesetzt werden kann. Das heißt, es ist*

$$c_1 = \mathrm{op}_1 ; \dots ; \mathrm{op}_i \quad \textit{und} \quad c = c_1 ; \mathrm{op}_{i+1} ; \dots ; \mathrm{op}_j.$$

Beweisskizze. Um den Cospan $c_1\colon \emptyset \to G_1 \xleftarrow{inj.} D_m$ durch Komposition der Graphoperationen zu konstruieren, sind die nachfolgenden Schritte anzuwenden. Seien im Folgenden $V_{G_1} = \{v_0, \dots, v_r, v_{r+1}, \dots, v_s\}$, wobei $V' = \{v_{r+1}, \dots, v_s\}$ ohne Beschränkung der Allgemeinheit die Menge der isolierten Knoten von G_1 ist, und $E_{G_1} = \{e_1, \dots, e_t\}$.

1. Für jede k-stellige, mit A beschriftete Hyperkante $e \in E_{G_1}$ komponiere mit $\mathsf{edge}_z^{A,k}$ für ein geeignetes $z \in \mathbb{N}$.

2. Seien $e_\mu, e_\nu \in E_{G_1}$ zwei beliebige Hyperkanten, für die $|\mathsf{att}_{G_1}(e_\mu)| = k$ und $|\mathsf{att}_{G_1}(e_\nu)| = \ell$ gilt, und seien $E' = E_{G_1} \setminus \{e_{\mu+1}, \dots, e_t\}$ und $E'' = E_{G_1} \setminus \{e_{\nu+1}, \dots, e_t\}$. Falls Indizes p und q mit $1 \le p \le k$ und $1 \le q \le \ell$ existieren, so dass $\mathsf{att}_p(e_\mu) = \mathsf{att}_q(e_\nu)$ ist, dann komponiere mit

$$(v_0 v_x) ; (v_1 v_y) ; \mathsf{fuse}_z ; (v_0 v_{x-1}),$$

für ein geeignetes $z \in \mathbb{N}$ und für

$$x = \sum_{e \in E'} |\mathsf{att}_{G_1}(e)| + p \quad \text{und} \quad y = \sum_{e \in E''} |\mathsf{att}_{G_1}(e)| + q,$$

wobei sich die Transpositionen $(v_0 v_x)$, $(v_1 v_y)$ und $(v_0 v_{x-1})$ nach Satz 5.7 durch perm_φ und trans_ψ für geeignete $\varphi, \psi \in \mathbb{N}$ darstellen lassen.

3. Wiederhole Schritt zwei, so lange noch Indizes p und q für Hyperkanten $e_\mu, e_\nu \in E_{G_1}$ existieren. Anschließend ist die Abbildung vom äußeren Interface in den so konstruierten Graphen injektiv.

4. Für jeden isolierten Knoten $v \in V'$ komponiere mit vertex_z für ein geeignetes $z \in \mathbb{N}$. Wenn alle isolierten Knoten hinzugefügt wurden, ist der Graph G_1 vollständig konstruiert.

5. Falls ein Knoten $v_\mu \in V_{G_1}$ mit $v_\mu \notin \mathrm{Im}\left(c_1^R\right)$ existiert, komponiere mit $(v_\mu v)\,;\mathsf{res}_z$ für ein geeignetes $z \in \mathbb{N}$, wobei v der letzte Knoten (bezüglich der Aufzählung der Knotenmenge) ist, der noch im äußerem Interface des konstruierten Cospans liegt.

6. Wiederhole Schritt fünf, so lange noch Knoten $v_\mu \in V_{G_1}$ mit $v_\mu \notin \mathrm{Im}\left(c_1^R\right)$ existieren. Im Anschluss befinden sich insgesamt m Knoten im äußeren Interface.

7. Permutiere die m Knoten des äußeren Interface wie folgt

$$\begin{pmatrix} v_1 & \dots & v_m \\ c_1^R(v_1) & \dots & c_1^R(v_m) \end{pmatrix}.$$

Durch die Komposition der oben aufgelisteten Graphoperationen wird der Cospan c_1 erzeugt. Die Erweiterung von c_1 zum Cospan c durch die Graphoperationen $\mathrm{op}_{i+1}, \dots, \mathrm{op}_j$ kann in ähnlicher Weise durch die Schritte eins bis sieben konstruiert werden. Allerdings müssen gegebenenfalls zunächst weitere Knoten aus dem Interface D_m durch die fuse-Operation verklebt (sowie durch entsprechende vorhergehende und nachfolgende Transpositionen permutiert) werden. Außerdem ist nach Schritt zwei darauf zu achten, dass unter Umständen die Knoten einer neuen Hyperkante mit Knoten aus dem Interface D_m verklebt werden müssen. Sei daher $E_{G_2} = E_{G_1} \cup \{e_{t+1}, \dots, e_u\}$. Dann muss Schritt zwei wie folgt erweitert werden:

2a. Sei $e_\alpha \in E_{G_2} \setminus E_{G_1}$ eine beliebige Hyperkante, für die $|\mathsf{att}_{G_2}(e_\alpha)| = \lambda$ gilt, und sei $E''' = E_{G_2} \setminus \{e_{\alpha+1}, \dots, e_u\}$. Falls ein Index $1 \leq w \leq \lambda$ existiert, so dass $\mathsf{att}_w(e_\alpha) = v$ für ein $v \in D_m$ ist, dann komponiere mit

$$(v_0 v_x)\,;(v_1 v)\,;\mathsf{fuse}_z\,;(v_0 v_{x-1}),$$

für ein geeignetes $z \in \mathbb{N}$ und für

$$x = \sum_{e \in E'''} |\mathsf{att}_{G_2}(e)| + w,$$

wobei sich die Transpositionen $(v_0 v_x)$, $(v_1 v)$ und $(v_0 v_{x-1})$ nach Satz 5.7 durch perm_φ und trans_ψ für geeignete $\varphi, \psi \in \mathbb{N}$ darstellen lassen.

□

5.2 Einschränkung der zulässigen Cospans

In diesem Abschnitt sollen die betrachteten Cospans und auch die Myhill-Pfeilquasiordnung für Cospans von Graphen, im Weiteren Myhill-Graphquasiordnung genannt, weiter eingeschränkt werden. Wie bereits erläutert, ist dies erforderlich, um das in Abschnitt 4.3 vorgestellte Algorithmenschema implementieren zu können. Ein erster Schritt dazu war die Einführung der atomaren Graphoperationen in Definition 5.4. Nach Satz 5.8 können damit alle Cospans mit leerem inneren Interface konstruiert werden. Allerdings ist dies alleine nicht ausreichend, da vom Algorithmus nach wie vor unendlich viele Cospans betrachtet werden müssten.

An dieser Stelle sollen daher beschränkte Cospans eingeführt werden, bei denen die Größe des Graphen und des äußeren Interfaces beschränkt ist.

Definition 5.9 (Beschränkter Cospan)**.** *Ein Cospan $c\colon S \to G \leftarrow T$ heißt* (durch k) beschränkt*, wenn eine endliche Folge von Graphoperationen $\mathrm{op}_1, \ldots, \mathrm{op}_j$ existiert, so dass*

$$c = \mathrm{op}_1 ; \ldots ; \mathrm{op}_j$$

ist und für alle Graphoperationen $op_i\colon D_{n_i} \to H \leftarrow D_{m_i}$ mit $1 \leq i \leq j$ gilt, dass $n_i, m_i \leq k$ ist. In diesem Fall wird k eine Schranke (von c) *genannt.*

Außerdem wird die Myhill-Graphquasiordnung für Cospans von Graphen ebenfalls eingeschränkt.

Definition 5.10 (Einseitige Myhill-Graphquasiordung)**.** *Sei $L_{\emptyset,\emptyset}$ eine Graphsprache über Cospan($\mathcal{Graph}$). Dann wird die Familie von Quasiordnungen*

$$\preccurlyeq_L = \{S_{D_n} \mid n \in \mathbb{N}\}$$

einseitige Myhill-Graphquasiordnung (relativ zu $L_{\emptyset,\emptyset}$) *genannt, wenn für alle Quasiordnungen S_{D_m} und beliebige Cospans $b, c\colon \emptyset \to D_m$ die folgende Bedingung erfüllt ist:*

$$b\ S_{D_m}\ c\ \mathsf{gdw.}\ \forall(d\colon D_m \to \emptyset)\colon \big((b\,;d) \in L_{\emptyset,\emptyset} \implies (c\,;d) \in L_{\emptyset,\emptyset}\big)\,.$$

Im Falle von Cospans von Graphen ist es ohne Beschränkung der Allgemeinheit ausreichend, die einseitige Myhill-Graphquasiordnung zu betrachten, wie der nachfolgende Satz zeigt:

Satz 5.11. *Seien $b, c\colon \emptyset \to D_n$ zwei Cospans. Dann gilt $b \leq_L c$ genau dann, wenn $b \preccurlyeq_L c$ ist.*

Beweis.

(⇒) Trivial. Setze $a = \mathsf{id}_\emptyset$.

(⇐) Sei $b \preccurlyeq_L c$. Dann ist zu zeigen, dass

$$\forall(a\colon \emptyset \to \emptyset), (d\colon M \to \emptyset)\colon \big((a\,;b\,;d) \in L_{\emptyset,\emptyset} \implies (a\,;c\,;d) \in L_{\emptyset,\emptyset}\big)\,.$$

Seien zwei Cospans $a\colon \emptyset \to \emptyset$ und $d\colon M \to \emptyset$ beliebig gegeben, so dass $(a\,;b\,;d) \in L_{\emptyset,\emptyset}$ ist. Weil die Komposition über dem leeren Interface eine disjunkte Vereinigung ist und die disjunkte Vereinigung zweier Graphen kommutativ ist, folgt daraus:

$$(a\,;b\,;d) = (b\,;d\,;a) \in L_{\emptyset,\emptyset} \overset{b \preccurlyeq_L c}{\Longrightarrow} (c\,;d\,;a) = (a\,;c\,;d) \in L_{\emptyset,\emptyset}.$$

Da die Cospans a und d beliebig gegeben waren, folgt damit $b \leq_L c$.

□

Durch die Beschränkung der zulässigen Cospans erhält man allerdings eine schwächere Myhill-Graphquasiordnung:

Definition 5.12 (Beschränkte, einseitige Myhill-Graphquasiordung)**.** *Sei $L_{\emptyset,\emptyset}$ eine Graphsprache über Cospan(Graph) und $k \in \mathbb{N}$. Dann wird die Familie von Quasiordnungen*

$$\preccurlyeq_L^k = \{S_{D_n}^k \mid n \in \mathbb{N}, n \leq k\}$$

beschränkte, einseitige Myhill-Graphquasiordnung (relativ zu $L_{\emptyset,\emptyset}$) genannt, wenn für alle Quasiordnungen $S_{D_m}^k$ mit $m \leq k$ und beliebige, durch k beschränkte Cospans $b, c\colon \emptyset \to D_m$ die folgende Bedingung erfüllt ist:

$b\ S_{D_m}^k\ c$ gdw. *für alle durch k beschränkten Cospans $d\colon D_m \to \emptyset$ gilt:*

$$\big((b\,;d) \in L_{\emptyset,\emptyset} \implies (c\,;d) \in L_{\emptyset,\emptyset}\big)\,.$$

Satz 5.13. *Seien $b, c\colon \emptyset \to D_n$ zwei Cospans. Dann folgt aus $b \preccurlyeq_L c$ auch $b \preccurlyeq_L^k c$.*

Beweis. Trivial. Folgt direkt aus den Definitionen von $\preccurlyeq_L$ und $\preccurlyeq_L^k$. □

Die Gegenrichtung gilt wahrscheinlich nicht, allerdings müsste dies durch ein entsprechendes Gegenbeispiel widerlegt werden.

In der Praxis stellt sich meist ein weiteres Problem. Das in Abschnitt 4.3 vorgestellte Algorithmenschema setzt einen deterministischen Automatenfunktor voraus. Dieser kann im schlimmsten Fall exponentiell größer sein als der zugehörige (nicht-deterministische) Automatenfunktor, der die gleiche Sprache erkennt. Daher sollen im Folgenden auch nicht-deterministische Automatenfunktoren zugelassen werden.

Allerdings ist dabei zu beachten, dass bei der Berechnung der Myhill-Pfeilquasiordnung der Eintrag der Relationstabelle, die während des Ablaufs des Algorithmus' aufgestellt wird, für ein Zustandspaar (s_0, s_1) von den jeweiligen Nachfolgezuständen s_0' bzw. s_1' abhängt. Falls die jeweiligen Nachfolgezustände nicht eindeutig sind, weil es sich um einen nicht-deterministischen Automatenfunktor handelt, kann es dazu kommen, dass unklar ist, ob die Zustände s_0 und s_1 in Relation stehen. Die Abbildung 5.1 zeigt die Problematik:

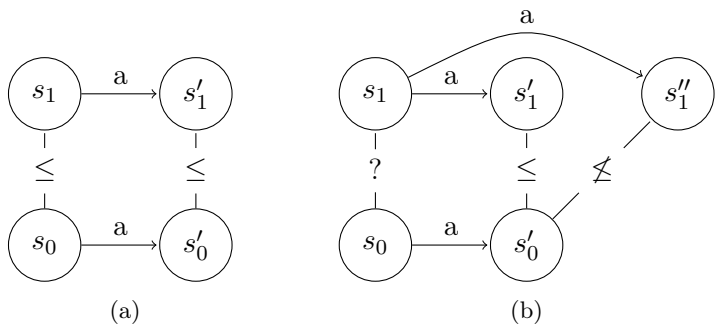


Abbildung 5.1: Ausschnitte aus zwei Automaten: (a) Deterministischer Automat; (b) Nicht-deterministischer Automat

Im Folgenden soll auf den in Abbildung 5.1(a) gezeigten Ausschnitt eines deterministischen Automaten näher eingegangen werden. Da die Zustände s_0 und s_1 jeweils eindeutige, zueinander in Relation stehende Nachfolgezustände Nachfolgezustände, s'_0 und s'_1, haben, kann daraus gefolgert werden, dass auch die Zustände s_0 und s_1 in Relation stehen. Im Gegensatz dazu zeigt die Abbildung 5.1(b) einen nicht-deterministischen Automaten, der sich dadurch unterscheidet, dass der Zustand s_1 keinen eindeutigen Nachfolgezustand besitzt, sondern entweder in den Zustand s'_1 oder s''_1 übergeht. In dem gegebenen Beispiel stehen die Zustände s'_0 und s'_1 in Relation, die Zustände s'_0 und s''_1 jedoch nicht. Daher ist es unklar, ob die Zustände s_0 und s_1 ebenfalls in Relation stehen oder nicht. Dies hängt davon ab, in welchen Nachfolgezustand der Zustand s_0 übergeht.

Um dieses Problem zu vermeiden, bieten sich zwei Ansätze an:

1. Erzeugung des deterministischen, minimalen Automatenfunktors, der die gleiche Sprache erkennt wie der nicht-deterministische Automatenfunktor.

2. Unterapproximation der Myhill-Quasiordnung durch eine feinere Relation.

Die Erzeugung des minimalen Automatenfunktors durch die Potenzmengenkonstruktion führt, wie schon erläutert, zu einer exponentiellen Vergrößerung der Zustandsmengen. Da im Rahmen dieser Arbeit Invarianten überprüft werden sollen, die durch das Nicht-Vorhandensein bestimmter Subgraphen ausgedrückt werden, und die dafür benötigten (nicht-deterministischen) Automatenfunktoren sehr große Zustandsmengen besitzen, ist der erste Ansatz nicht praktikabel.

Durch den zweiten Ansatz, die Unterapproximation, wird nicht mehr die eigentliche Myhill-Graphquasiordnung berechnet, da es sich um eine Verfeinerung handelt. Stattdessen wird nur noch die Bedingung gestellt, dass zwei Zustände, die bzgl. der Unterapproximation in Relation stehen, auch bzgl. der Myhill-Graphquasiordnung in Relation stehen müssen. Das bedeutet, es ist ein so genannter *einseitiger Fehler* zugelassen, die Abbildung 5.2 verdeutlicht den Zusammenhang.

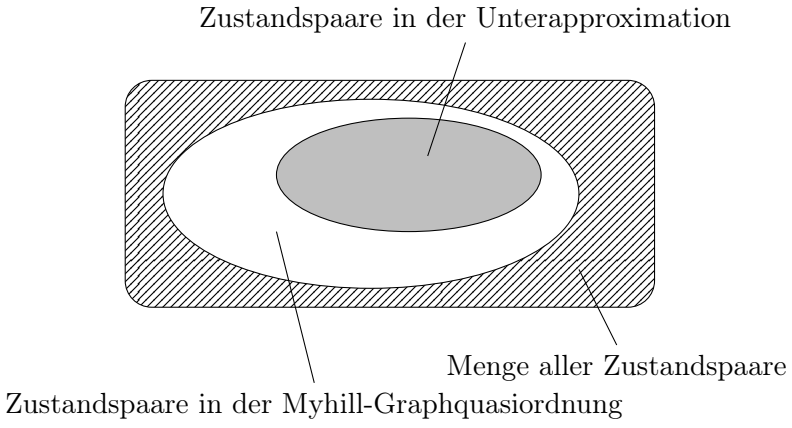


Abbildung 5.2: Schematische Darstellung der Unterapproximation

Als eine mögliche Unterapproximation bietet sich die Simulationsrelation an. Diese setzt zwei Cospans b und c in Relation, wenn sich alle Übergänge von Zuständen, welche nach Einlesen des Cospans b erreicht werden können, durch

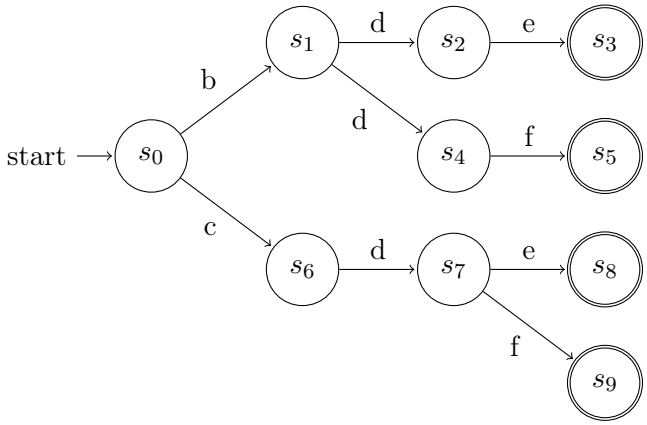


Abbildung 5.3: Beispiel für die Simulationsrelation

entsprechende Übergänge von Zuständen simulieren lassen, die durch Einlesen des Cospans c erreicht werden können. Dabei wird ein Zustand s dann und nur dann durch einen Zustand s' simuliert, wenn für jeden Übergang, bei dem s in einen Endzustand überführt wird, auch s' durch einen entsprechenden Übergang in einen Endzustand überführt wird.

In Abbildung 5.3 ist ein Beispiel dargestellt, auf das an dieser Stelle näher eingegangen werden soll. Der Cospan b steht in Simulationsrelation zu c (in Zeichen: $b \leq_S c$), da alle Übergänge, mit denen vom Zustand s_1 aus ein Endzustand erreicht werden kann, durch entsprechende Übergänge vom Zustand s_6 aus *simuliert* werden können. Das heißt, alle (Mehrfach-)Übergange, die von s_1 aus möglich sind, sind auch von s_6 aus möglich. Außerdem führen alle Übergänge, bei denen man von s_1 aus in einen Endzustand gelangt, auch von s_6 aus in einen Endzustand. Dies muss nicht notwendigerweise derselbe Endzustand sein. Umgekehrt stehen die beiden Cospans nicht in Relation, das heißt s_6 wird nicht durch s_1 simuliert. Dies kann wie folgt gezeigt werden:

Vom Zustand s_1 aus kann nicht jeder Übergang, mit dem vom Zustand s_6 aus ein Endzustand erreichbar ist, simuliert werden. Zunächst ist die einzige Möglichkeit ein d-Übergang, bei dem der Automat von s_6 nach s_7 wechselt. Dieser Schritt muss nun von s_1 aus dadurch simuliert werden, dass der Automat entweder nach s_2 oder nach s_4 wechselt. Falls der Automat nach s_2 übergegangen und der nächste Schritt ein f-Übergang wäre, könnte dieser Übergang nicht mehr simuliert werden. Analog kann ein e-Übergang vom Zustand s_4 nicht simuliert werden. Damit sind vom Zustand s_1 aus Zustände erreichbar, die nicht in Simulationsrelation zu den Zuständen stehen, die vom Zustand s_6 aus erreichbar sind. Daher kann auch der Zustand s_1 den Zustand s_6 nicht simulieren.

Formal kann die Simulationsrelation wie folgt definiert werden:

Definition 5.14 (Beschränkte Simulationsrelation)**.** *Sei $L_{\emptyset,\emptyset}$ eine Graphsprache und $\mathcal{A}$ der Automatenfunktor, der die Sprache $L_{\emptyset,\emptyset}$ erkennt. Dann wird die Familie von Quasiordnungen*

$$\leq_S^k = \{S_{D_n} \mid n \leq k, n \in \mathbb{N}\}$$

eingeschränkte Simulationrelation (relativ zu $L_{\emptyset,\emptyset}$) *genannt, wenn für alle Qua-*

siordnungen S_{D_n} und beliebige, durch k beschränkte Cospans $b, c\colon \emptyset \to D_n$ die folgende Bedingung erfüllt ist:

$$b\ S_{D_n}\ c \text{ gdw. } \forall s_1 \in \mathcal{A}(b)(\mathrm{I}_\emptyset^{\mathcal{A}})\colon \exists s_2 \in \mathcal{A}(c)(\mathrm{I}_\emptyset^{\mathcal{A}})\colon \forall d\colon D_n \to \emptyset\colon \\ \mathcal{A}(d)(s_1) \cap F_\emptyset^{\mathcal{A}} \neq \emptyset \implies \mathcal{A}(d)(s_2) \cap F_\emptyset^{\mathcal{A}} \neq \emptyset$$

Es ist entscheidend, dass man fordert, dass es für jeden, mittels b erreichbaren Zustand s mindestens einen, mittels c erreichbaren Zustand s' geben muss, der s simuliert. Dies nur für einen, mittels b erreichbaren Zustand zu fordern, wäre nicht ausreichend, da es sich dann nicht mehr um eine Unterapproximation handeln würde. Diesen Zusammenhang zeigt auch das Beispiel in Abbildung 5.4 dargestellten Beispiels zeigen (mit „Pseudo“-Simulationsrelation sei dabei die abgeschwächte Form der Simulationsrelation bezeichnet):

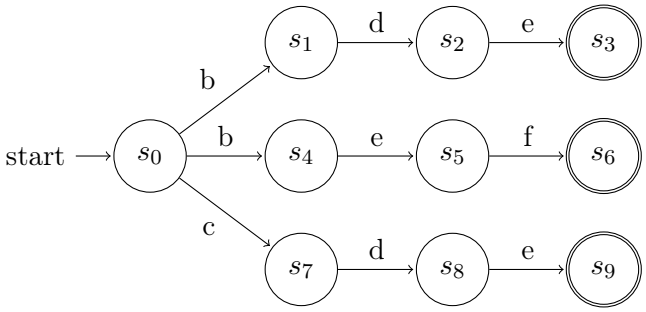


Abbildung 5.4: Beispiel für die „Pseudo“-Simulationsrelation

Vom Zustand s_0 aus gelangt man durch einen b-Übergang entweder in den Zustand s_1 oder in den Zustand s_4 und entsprechend durch einen c-Übergang in den Zustand s_7. Der Zustand s_1 wird nun durch den Zustand s_7 simuliert. Aber obwohl demnach die Cospans b und c in „Pseudo“-Simulationsrelation stehen, handelt es sich nicht um eine Unterapproximation. Dies liegt daran, dass von s_0 aus auch ein b-Übergang nach s_4 möglich ist. Der Zustand s_4 wird allerdings nicht durch den Zustand s_7 simuliert, da von s_7 aus kein e-Übergang existiert und daher der Zustand des Automaten undefiniert ist. Der Wert für die Unterapproximation ist somit unklar, denn je nachdem welcher von beiden b-Übergängen ausgewählt wird, befindet man sich innerhalb oder außerhalb der Myhill-Graphquasiordnung, die es zu approximieren gilt. Daher ist die „Pseudo“-Simulationsrelation keine alternative Unterapproximation für die Myhill-Graphquasiordnung.

Nachfolgend ist ein auf dem Algorithmenschema 4.16 basierender Algorithmus angegeben. Dieser ist dahingehend eingeschränkt, dass statt beliebigen Cospans nur noch die oben definierten beschränkten Cospans zugelassen sind und statt der beschränkten Myhill-Graphquasiordnung die Simulationsrelation berechnet wird. Durch die Beschränkung der Cospans ist sichergestellt, dass nur eine endliche Anzahl an Cospans untersucht werden muss, wodurch dieser Algorithmus ohne weitere Änderungen implementierbar ist.

Algorithmus 5.15.

- *Eingabe: Beschränkte Cospans $b, c\colon \emptyset \to D_n$ mit $n \leq k$ und ein Automatenfunktor $\mathcal{A}$, der die Graphsprache $L_{\emptyset,\emptyset}$ erkennt.*
- *Ausgabe: „Ja“, falls $b \leq_S^k c$ und „Nein“, falls $b \not\leq_S^k c$.*
- *Ablauf:*
  0. *Mit $\mathcal{A}'$ sei der Automatenfunktor bezeichnet, den man aus $\mathcal{A}$ erhält, wenn man alle unerreichbaren Zustände entfernt.*
  1. *Stelle für jede Menge $\mathcal{A}'(D_i)$ mit $0 \leq i \leq k$ jeweils eine Menge aller Zustandspaare (s_0, s_1) mit $s_0, s_1 \in \mathcal{A}'(D_i)$.*
  2. *Markiere alle Paare (s_0, s_1) mit $s_0, s_1 \in \mathcal{A}'(D_0)$ für die $s_0 \in \mathrm{F}_{D_0}^{\mathcal{A}'}$ und $s_1 \notin \mathrm{F}_{D_0}^{\mathcal{A}'}$.*
  3. *Für jedes noch unmarkierte Paar (s_0, s_1) mit $s_0, s_1 \in \mathcal{A}'(D_i)$ für $0 \leq i \leq k$ und jede Graphoperation* op *teste, ob für alle $s_0' \in \mathcal{A}'(\mathrm{op})(s_0)$ ein $s_1' \in \mathcal{A}'(\mathrm{op})(s_1)$ existiert, so dass das Paar (s_0', s_1') unmarkiert ist. Wenn nicht: markiere auch (s_0, s_1).*
  4. *Wiederhole den vorherigen Schritt solange, bis sich keine Änderungen für die Mengen von Zustandspaaren mehr ergeben.*
  5. *Überprüfe für jeden Zustand $i \in \mathrm{I}_{D_0}^{\mathcal{A}'}$, ob für jeden Zustand $s_0 \in \mathcal{A}'(b)(i)$ ein Zustand $s_1 \in \mathcal{A}'(c)(i)$ existiert, so dass (s_0, s_1) unmarkiert ist. Falls für jedes s_0 ein solches s_1 existiert, gib „Ja“ aus, ansonsten gib „Nein“ aus.*

Konkret bedeutet dies, dass einerseits (nicht-deterministische) Automatenfunktoren zugelassen sind. Dies ist notwendig, um den in Abschnitt 5.3 vorgestellten Automatenfunktor (praktisch) implementieren zu können (siehe auch Kapitel 6). Andererseits führt die dafür notwendige Unterapproximation dazu, dass der in diesem Abschnitt beschriebene, einseitiger Fehler zugelassen wird. Dies hat zur Folge, dass der Algorithmus an dieser Stelle „Nein“ ausgibt, obwohl die eingegebenen Cospans b und c in Relation bezüglich der Myhill-Graphquasiordnung zueinander stehen. Ein konkretes Beispiel, bei dem dieses Problem auftritt, ist in Kapitel 7 angegeben.

5.3 Ein Automatenfunktor zum Prüfen von Invarianten

In diesem Abschnitt soll ein Automatenfunktor vorgestellt werden, der die Sprache aller Graphen erkennt, die einen bestimmten Subgraphen U enthalten. Da sich Systeme und erwünschte wie unerwünschte Systemeigenschaften oftmals in Form von Graphen modellieren lassen, kann die erwünschte bzw. unerwünschte Systemeigenschaft als der vom Automatenfunktor gesuchte Subgraph U dienen. Die Graphen, die die Zustände des Systems modellieren, werden dann daraufhin untersucht, ob sie den gesuchten Subgraph, der die erwünschte bzw. unerwünschte Systemeigenschaft modelliert, enthalten. Damit kann der Nachweis, dass das System über die gewünschte Eigenschaft verfügt, durch eine Vorwärtsanalyse geführt werden, indem gezeigt wird, dass die Graphsprache eine Invariante bezüglich der

erwünschten Eigenschaft ist. Somit bleibt die Eigenschaft unter dem dynamischen Verhalten des Systems erhalten. Ebenso kann durch eine Rückwärtsanalyse gezeigt werden, dass die Graphsprache, die der Automatenfunktor erkennt, eine Invariante bezüglich der unerwünschten Systemeigenschaft ist. Das bedeutet, wenn das System über die unerwünschte Eigenschaft verfügt, dann gehörte sie bereits vor dem initialen (System-)Zustand zu dem System.

Im Folgenden soll die Definition und Konstruktion des Automatenfunktors vorgestellt werden. Die grundlegende Idee beim Erkennen des gesuchten Subgraphen U in einem Graphen G kann wie folgt beschrieben werden: Nach Satz 5.8 kann der Cospan $[G]\colon \emptyset \to G \leftarrow \emptyset$ in einzelne, atomare Graphoperationen zerlegt werden, deren Komposition diesen Cospan erzeugt. Daher wird in jedem Arbeitsschritt des Automatenfunktors eine solche Graphoperation verarbeitet. Dabei speichert bzw. aktualisiert der Automatenfunktor zwei Informationen: zum einen welchen Teil von U er bereits erkannt hat bzw. durch die Graphoperation erkennt und zum anderen welche Knoten von U sich noch im (äußeren) Interface des (letzten komponierten) Cospans befinden. In Abbildung 5.5 ist eine solche Situation dargestellt. Der bisher erkannte Teil von U, bevor die nächste Graphoperation komponiert wird, ist grau unterlegt und schraffiert. Die Knoten (hier nicht dargestellt), die in der Schnittmenge von D_n und dem grau unterlegten, schraffierten Bereich liegen, werden außerdem auf ihre entsprechenden Knoten in U abgebildet. Nachdem die Graphoperation komponiert wurde, wird der gesamte, grau unterlegte Teil von U erkannt und die Knoten (ebenfalls nicht dargestellt), die in der Schnittmenge von D_m und dem grau unterlegten Bereich liegen, wiederum auf die entsprechenden Knoten des Subgraphen U abgebildet.

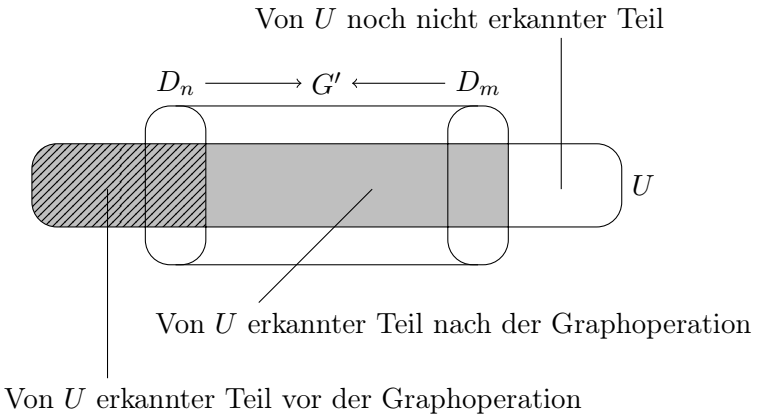


Abbildung 5.5: Beispiel für einen Automatenfunktor

Formal lässt sich dies wie folgt definieren:

- Für jeden Graphen G ist die Zustandsmenge definiert als

$$\mathcal{A}(G) = \{(U_G, f_G) \mid U_G \leq U, f_G\colon V_G \rightharpoonup V_{U_G}\}.$$

  Jeder Graph wird auf ein Paar bestehend aus einem Graphen U_G und einer partiellen Funktion f_G abgebildet. Dabei ist der Graph U_G ein Subgraph des gesuchten Graphen U und die Funktion f_G bildet die Knoten aus G entweder

auf die entsprechenden Knoten aus U_G ab, falls diese existieren, oder ist für den jeweiligen Knoten undefiniert. Das heißt der jeweilige Knoten wird auf $\perp$ abgebildet, falls in dem Subgraphen U_G kein entsprechender Knoten vorkommt.

- Für jeden Graphen G ist die Startzustandsmenge definiert als

$$\mathrm{I}_G^{\mathcal{A}} = \begin{cases} \{(\emptyset, \emptyset)\}, & \text{falls } G = \emptyset \\ \emptyset, & \text{sonst} \end{cases}.$$

- Für jeden Graphen G ist die Endzustandsmenge definiert als

$$\mathrm{F}_G^{\mathcal{A}} = \begin{cases} \{(U, \emptyset)\}, & \text{falls } G = \emptyset \\ \emptyset, & \text{sonst} \end{cases}.$$

- Als Pfeile werden nur die in Definition 5.4 vorgestellten Graphoperationen zugelassen. Dies ist ausreichend, da nach Satz 5.8 alle Cospans der Form $c\colon \emptyset \to G \leftarrow D_n$ durch diese Graphoperationen erzeugt werden können:
  - *Restriktion des äußeren Interfaces*: Der Cospan res_n wird auf die Relation $\mathcal{A}(\mathsf{res}_n) \subseteq \mathcal{A}(D_n) \times \mathcal{A}(D_{n-1})$ abgebildet, die zwei Zustände (U_D, f_D) und (U'_D, f'_D) genau dann in Relation setzt, wenn $U_D = U'_D$ ist, für alle Knoten $v \in V_{D_{n-1}}$ gilt $f_D(v) = f'_D(v)$ und es ist $f_D(V_{n-1}) \notin \mathrm{Im}\,(f'_D)$.

    Durch den Cospan res_n wird der erkannte Graph nicht verändert. Stattdessen wird der letzte Knoten (entsprechend der Aufzählung der Bildmenge), der im Bild der Funktion f_D liegt, verschattet.
  - *Permutation des äußeren Interfaces*: Der Cospan perm_n wird auf die Relation $\mathcal{A}(\mathsf{perm}_n) \subseteq \mathcal{A}(D_n) \times \mathcal{A}(D_n)$ abgebildet, die zwei Zustände (U_D, f_D) und (U'_D, f'_D) genau dann in Relation setzt, wenn $U_D = U'_D$ ist und für alle Knoten $v_i \in V_{D_n} = \{v_0, \ldots, v_{n-1}\}$ gilt

$$f'_D(v_i) = \begin{cases} f_D(v_{i+1}), & \text{falls } i < n \\ f_D(v_0), & \text{sonst} \end{cases}.$$

    Durch den Cospan perm_n wird der erkannte Graph nicht verändert. Stattdessen werden die Knoten, die im Bild der Funktion f_D liegen, auf ihren Nachfolger (bezüglich der Aufzählung der Bildmenge) abgebildet.
  - *Transposition des äußeren Interfaces*: Der Cospan trans_n wird auf die Relation $\mathcal{A}(\mathsf{trans}_n) \subseteq \mathcal{A}(D_n) \times \mathcal{A}(D_n)$ abgebildet, die zwei Zustände (U_D, f_D) und (U'_D, f'_D) genau dann in Relation setzt, wenn $U_D = U'_D$ ist und für alle Knoten $v_i \in V_{D_n} = \{v_0, \ldots v_{n-1}\}$ gilt

$$f'_D(v_i) = \begin{cases} f_D(v_1), & \text{falls } i = 0 \\ f_D(v_0), & \text{falls } i = 1 \\ f_D(v_i), & \text{sonst} \end{cases}.$$

    Durch den Cospan trans_n wird der erkannte Graph nicht verändert. Stattdessen werden die ersten beiden Knoten (bezüglich der Aufzählung der Bildmenge), die im Bild der Funktion f_D liegen, vertauscht und die restlichen Knoten unverändert abgebildet.

– *Fusion zweier Knoten des äußeren Interfaces*: Der Cospan fuse_n wird auf die Relation $\mathcal{A}(\mathsf{fuse}_n) \subseteq \mathcal{A}(D_n) \times \mathcal{A}(D_{n-1})$ abgebildet, die zwei Zustände (U_D, f_D) und (U'_D, f'_D) genau dann in Relation setzt, wenn $U_D = U'_D$ sowie $f_D(v_0) = f_D(v_1)$ ist und für alle Knoten $v \in V_{D_{n-1}}$ gilt $f_D(v) = f'_D(v)$.

  Durch den Cospan fuse_n wird der erkannte Graph nicht geändert. Stattdessen wird der zweite Knoten aus dem Interface entfernt. Allerdings ist der Cospan nur dann anwendbar, wenn die ersten beiden Knoten (bezüglich der Aufzählung der Knotenmenge) auf denselben Knoten des Subgraphen U_D abgebildet werden.

– *Disjunkte Vereinigung mit einer Hyperkante*: Der Cospan $\mathsf{edge}_n^{A,m}$ wird auf die Relation $\mathcal{A}(\mathsf{edge}_n^{A,m}) \subseteq \mathcal{A}(D_n) \times \mathcal{A}(D_{n+m})$ abgebildet. Es sind zwei Fälle zu unterscheiden, für die zwei Zustände (U_D, f_D) und (U'_D, f'_D) in Relation gesetzt werden. Dabei ist zu beachten, dass der Automatenfunktor für jede passende Hyperkante entscheidet, ob es sich um eine der gesuchten Hyperkanten handelt oder nicht. Eine Hyperkante heißt in diesem Zusammenhang passend, wenn die Hyperkante über die korrekte Stelligkeit und die korrekte Beschriftung einer Hyperkante aus dem gesuchten Subgraph verfügt:

  Mit e sei die durch $\mathsf{edge}_n^{A,m}$ neu hinzugefügte Hyperkante bezeichnet

  1. Fall: $e \in E_U$
  Die beiden Zustände werden genau dann in Relation gesetzt, wenn eine Knotenmenge $V = \{\mathsf{att}_1(e), \ldots, \mathsf{att}_m(e)\}$ existiert, so dass $V \subseteq \mathrm{Im}\,(f_D) \cup (V_U \setminus V_{D_n})$, $U'_D = U_D \dot\cup \{e\} \cup V$, $f'_D(v_i) = f_D(v_i)$ für $0 \leq i < n$ und $f'_D(v_i) = \mathsf{att}_{i-n+1}(v_i)$ für $n \leq i < m$ ist.

  2. Fall: $e \notin E_U$
  Die beiden Zustände werden genau dann in Relation gesetzt, wenn eine Knotenmenge $V \subseteq \mathrm{Im}\,(f'_D \setminus f_D)$ existiert, so dass $U'_D = U_D \cup V$, $f'_D(v_n) = v$ für $0 \leq i < n$ und $f'_D(v_i) \in (\mathrm{Im}\,(f_D) \cup V \cup \{\perp\})$ für $n \leq i < m$ ist.

  Durch den Cospan $\mathsf{edge}_n^{A,m}$ wird der Automatenfunktor (nicht-deterministisch) in einen von mehreren möglichen Zuständen versetzt. Der jeweilige Nachfolgezustand hängt davon ab, ob

  * die neu hinzugefügte Hyperkante e in dem gesuchten Graphen U vorkommt (und bisher nicht hinzugefügt wurde). In diesem Fall wird der bisher erkannte Graph U_D um die Hyperkante e erweitert und die mit e verbundenen Knoten zu U_D hinzugefügt (falls sie noch nicht in U_D vorhanden sind). Die Funktion f_D wird um m Knoten erweitert, so dass jeder neue Knoten auf den entsprechenden, mit e verbundenen Knoten in dem Graphen U'_D abgebildet wird.

  * die neu hinzugefügte Hyperkante e nicht in dem gesuchten Graphen U vorkommt, aber Knoten mit e verbunden sind, die in U liegen, aber bisher nicht dem erkannten Graphen U_D liegen. In diesem Fall wird der erkannte Graph U_D um die bisher nicht enthaltenen Knoten erweitert. Die Funktion f_D wird um m Knoten erweitert, so

dass jeder neue Knoten auf den entsprechenden, mit e verbundenen Knoten in dem Graphen U'_D abgebildet wird.

- die neu hinzugefügte Hyperkante e nicht in dem gesuchten Graphen U vorkommt und auch die mit e verbundenen Knoten nicht in U liegen. In diesem Fall wird der erkannte Graph U_D nicht verändert. Die Funktion f_D wird um m Knoten erweitert, so dass jeder neue Knoten auf $\perp$ abgebildet wird.

– *Disjunkte Vereinigung mit einem Knoten*: Der Cospan vertex_n wird auf die Relation $\mathcal{A}(\mathsf{vertex}_n) \subseteq \mathcal{A}(D_n) \times \mathcal{A}(D_{n+1})$ abgebildet. Es sind drei Fälle zu unterscheiden, wann zwei Zustände (U_D, f_D) und (U'_D, f'_D) in Relation gesetzt werden. Dabei kann der Automatenfunktor entscheiden, ob der neu hinzugefügte Knoten Teil des gesuchten Subgraphen ist oder nicht:

Mit v sei der durch vertex_n neu hinzugefügte Knoten bezeichnet

1. Fall: $v \in \mathrm{Im}\,(f_D)$
Die beiden Zustände werden genau dann in Relation gesetzt, wenn $U'_D = U_D$, $f'_D(v_n) = v$ und $f'_D(v_i) = f_D(v_i)$ für $0 \leq i < n$ ist.

2. Fall: $v \notin \mathrm{Im}\,(f_D)$ und $v \in V_U$
Die beiden Zustände werden genau dann in Relation gesetzt, wenn $v \notin V_{D_n}$, $U'_D = U_D \cup \{v\}$, $f'_D(v_n) = v$ und $f'_D(v_i) = f_D(v_i)$ für $0 \leq i < n$ ist.

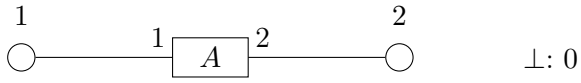
3. Fall: $v \notin V_U$
Die beiden Zustände werden genau dann in Relation gesetzt, wenn $U'_D = U_D$, $f'_D(v_n) = \perp$ und $f'_D(v_i) = f_D(v_i)$ für $0 \leq i < n$ ist.

Durch den Cospan vertex_n wird der Automatenfunktor (nicht-deterministisch) in einen von mehreren möglichen Zuständen versetzt. Der jeweilige Nachfolgezustand hängt davon ab, ob

- der neu hinzugefügte Knoten v bereits erkannt wurde. In diesem Fall wird der erkannte Graph U_D nicht verändert. Die Funktion f_D wird um einen Knoten erweitert, der auf den Knoten v abgebildet wird.
- der neu hinzugefügte Knoten v bisher nicht erkannt wurde, aber in dem gesuchten Graphen U vorkommt. In diesem Fall wird der erkannte Graph U_D um den Knoten v erweitert. Die Funktion f_D wird ebenfalls um einen Knoten erweitert, der auf den neuen Knoten v abgebildet wird.
- der neu hinzugefügte Knoten v nicht in dem gesuchten Graphen U vorkommt. In diesem Fall wird der erkannte Graph U_D nicht verändert. Die Funktion f_D wird um einen Knoten erweitert, der auf $\perp$ abgebildet wird.

Zur Verdeutlichung der Funktionsweise des Automatenfunktors sollen exemplarisch einige Zustandsübergänge aufgezeigt werden. Der gesuchte Subgraph U soll der Graph aus Beispiel 5.5 sein. Um die Darstellung zu vereinfachen, wird die Interfacefunktion f_G für einen Zustand (U_G, f_G) dadurch angegeben, dass die

Knoten aus U_G, auf welche die Interfacefunktion abbildet, nummeriert werden. Die Nummerierung entspricht dabei den Indizes der jeweiligen Knoten aus G, die auf den nummerierten Knoten (aus U_G) abgebildet werden. Falls die Interfacefunktion für einen Knoten aus G undefiniert ist, wird dies dadurch gekennzeichnet, dass neben dem Index des Knotens das Symbol $\perp$ platziert wird. Sei beispielsweise der folgende Zustand (U_{G_1}, f_{G_1}) gegeben:

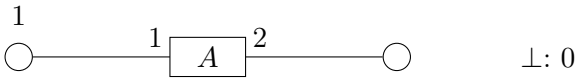


Der bisher erkannte Teil des Subgraphen U besteht in diesem Fall aus der mit A beschrifteten Hyperkante und den Knoten v_0 und v_1. Die Knoten $v_0', v_1', v_2' \in V_{G_1}$ werden wie folgt von der Interfacefunktion f_{G_1} abgebildet:

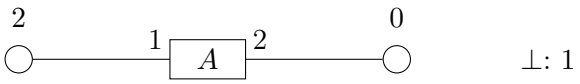
$$f_{G_1}(v_0') = \perp, \quad f_{G_1}(v_1') = v_1, \quad f_{G_1}(v_2') = v_2.$$

Von diesem Zustand existieren unter anderem die folgenden Nachfolgezustände, wobei an dieser Stelle nur einige der möglichen Nachfolgezustände betrachtet werden sollen:

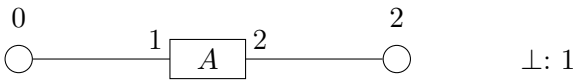
- res_3: Der Knoten v_2' wird aus dem Definitionsbereich der Interfacefunktion entfernt.



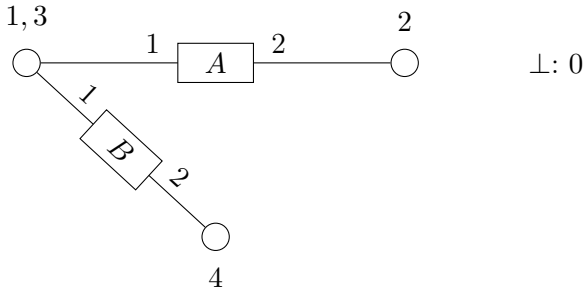
- perm_3: Alle Interfaceknoten werden rotiert.

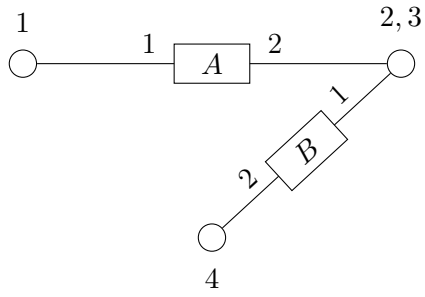


- trans_3: Die ersten beiden Interfaceknoten werden vertauscht.



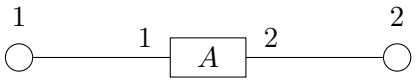
- $\mathsf{edge}_3^{B,2}$: Es sind unter anderem die folgenden Nachfolgezustände erreichbar:
  - Es wird eine der beiden gesuchten B-Hyperkanten hinzugefügt.



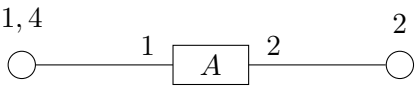


⊥: 0

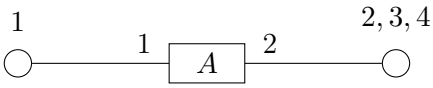
– Es werden nur Knoten, aber keine Hyperkanten erkannt.



⊥: 0, 4

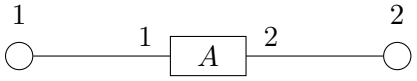


⊥: 0



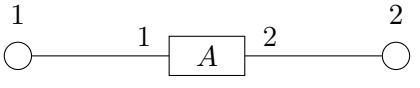
⊥: 0

– Es werden weder Knoten noch Hyperkanten erkannt.



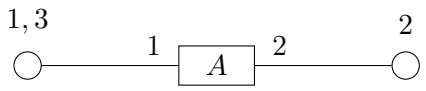
⊥: 0, 3, 4

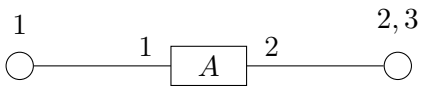
- vertex_3: Es sind die folgenden Nachfolgezustände erreichbar:
  – Es wird ein neuer Knoten erkannt.



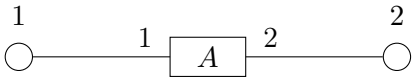
⊥: 0

– Es wird ein Knoten erkannt, der bereits gesehen wurde und noch im Interface enthalten ist.

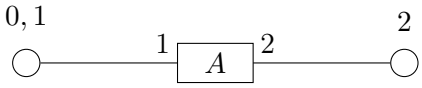




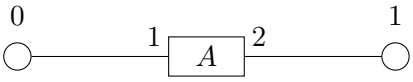
– Es wird weder ein neuer Knoten noch ein Knoten aus dem Interface erkannt.



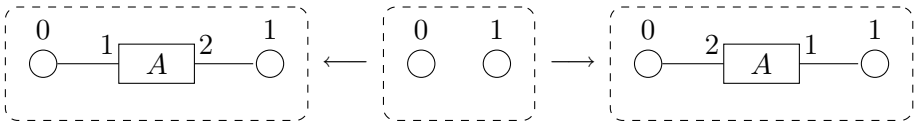
- fuse_3: Diese Graphoperation ist für diesen Zustand nicht anwendbar. Betrachte daher den folgenden Zustand:



Durch Anwendung der Graphoperation fuse_3 ergibt sich der folgende Nachfolgezustand:



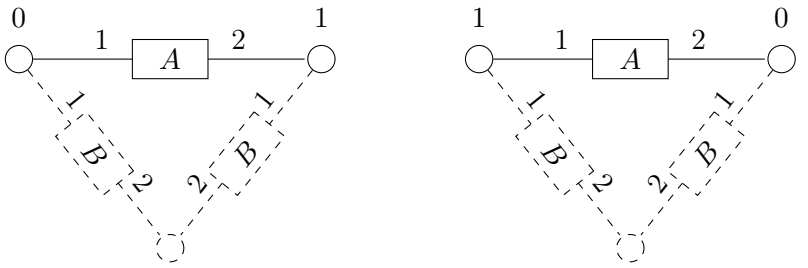
Zum Abschluss soll ein kurzes Beispiel gegeben werden. Betrachtet werden soll die Sprache aller Graphen, die den Graphen aus Beispiel 5.5 enthalten. Die Frgae ist, ob diese Sprache eine Invariante für die folgende Granphtransformationsregel ist:



Durch die Anwendung der Regel wird also genau eine zweistellige, mit A beschriftete Hyperkante umgekehrt. Dass die Sprache invariant unter Anwendung dieser Granphtransformationsregel ist, kann man sich durch die folgenden Überlegungen verdeutlichen:

Durch das Einlesen der linken bzw. der rechten Seite der oben vorgestellten Regel sind insgesamt 17 verschiedene Nachfolgezustände (für jede der beiden Seiten) erreichbar. Allerdings unterscheiden sich die beiden Nachfolgezustandsmengen nur in zwei Zuständen. Diese beiden unterschiedlichen Zustände sollen im Folgenden näher betrachtet werden (siehe Abbildung 5.6, wobei die gestrichelten Linien andeuten sollen, dass diese Teile noch erkannt werden müssen). Die beiden betrachteten Zustände unterscheiden sich wie folgt: Im Falle des Zustands für die linke Seite wird der erste Knoten der Interfacefunktion auf den ersten und der

zweite Knoten der Interfacefunktion auf den zweiten Knoten der A-Hyperkante abgebildet (siehe Abbildung 5.6(a)). Im Falle des Zustands für die rechte Seite ist dies anders. Dort wird der erste Knoten der Interfacefunktion auf den zweiten und der zweite Knoten der Interfacefunktion auf den ersten Knoten der A-Hyperkante abgebildet (siehe Abbildung 5.6(b)). Die restlichen Zustände, die in beiden Nachfolgezustandsmengen enthalten sind, sind nicht weiter von Interesse, da auf Grund der Reflexivität der Myhill-Graphquasiordnung diese Zustände ohnehin in Relation stehen.



(a) Zustand für die linke Seite (b) Zustand für die rechte Seite

Abbildung 5.6: Erreichbare Zustände beim Einlesen der linken bzw. rechten Seite

Trivialerweise verhalten sich beide Zustände beim Verarbeiten der Graphoperationen fuse, perm, res, trans und vertex gleichwertig. Falls der linke Zustand einen Übergang für eine dieser Operationen macht und einen Nachfolgezustand erreicht, kann der rechte Zustand darauf reagieren, indem er in einen entsprechenden Nachfolgezustand übergeht. Dieser unterscheidet sich nur dadurch vom Nachfolgezustand der linken Seite, dass die Nummerierung für die beiden, mit der A-Hyperkante verbundenen, Knoten vertauscht sind (vergleiche die Abbildungen 5.6(a) und 5.6(b)).

Das Verhalten beim Verarbeiten der edge-Operation soll detaillierter betrachtet werden. Falls die neu hinzugefügte Hyperkante entweder nicht in dem gesuchten Graphen vorkommt, oder eine weitere, zweistellige, mit A beschriftete Hyperkante ist, reagieren die beiden Zustände analog zu dem Fall der vertex-Operation. Der einzige Unterschied besteht darin, dass die vertex-Operation nicht einmal, sondern zweimal – entsprechend der Stelligkeit der neuen Hyperkante – verarbeitet wird.

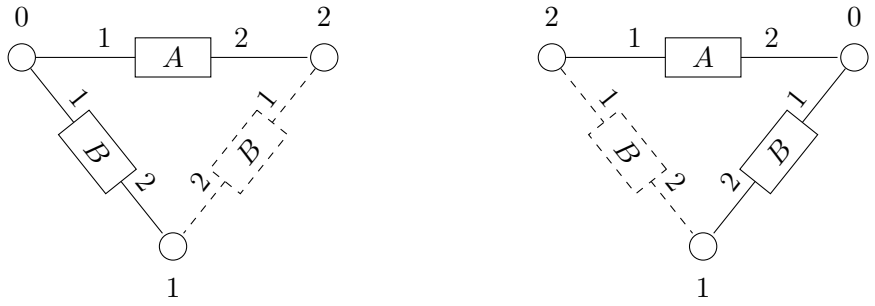
Falls es sich bei der neu hinzugefügten Hyperkante allerdings um eine zweistellige, mit B beschriftete Hyperkante handelt, ist das Verhalten für die linke und die rechte Seite unterschiedlich. Um diesen Unterschied besser zu verdeutlichen soll nicht nur die Operation $\mathsf{edge}_2^{B,2}$, sondern der folgende Ablauf betrachtet werden:

$$a = \mathsf{edge}_2^{B,2} ; \mathsf{perm}_4 ; \mathsf{trans}_4 ; \mathsf{perm}_4 ; \mathsf{fuse}_4.$$

Von den jeweils vier Nachfolgezuständen, die durch das Verarbeiten des angegeben Ablaufs erreicht werden können, sind drei Zustände nicht von Interesse, da diese Zustände ebenfalls durch zweimaliges Verarbeiten vertex-Operation erreichbar sind. Daher kann die rechte Seite entsprechend auf das Verhalten der linken Seite reagieren.

Daher werden nun die beiden Nachfolgezustände weiter betrachtet, in denen die neue B-Hyperkante als zum gesuchten Subgraphen gehörig erkannt wurde.

Wie in Abbildung 5.7 zu sehen ist, wird durch das Einlesen des Ablaufs a durch den Nachfolgezustand für die linke Seite, genau die B-Hyperkante erkannt, die den gleichen, ausgehenden Knoten wie die bereits erkannte A-Hyperkante hat (Abbildung 5.7(a)); durch den Nachfolgezustand für die rechte Seite wird allerdings die andere B-Hyperkante erkannt, deren ausgehender Knoten identisch mit den eingehenden Knoten der erkannten A-Hyperkante ist (Abbildung 5.7(b)).



(a) Nachfolgezustand für die linke Seite (b) Nachfolgezustand für die rechte Seite

Abbildung 5.7: Nachfolgezustände beim Einlesen des Ablaufs a

Analog kann der Zustand für die rechte Seite darauf reagieren, wenn der Zustand für die linke Seite die andere B-Hyperkante erkennt. Durch dieses Verhalten, bei dem linke und rechte Seite anschaulich gesprochen, „symmetrisch“ reagieren, wird verdeutlicht, warum die Sprache aller Graphen, die diesen „Dreiecksgraphen“ enthalten, eine Invariante bezüglich der betrachteten Graphtransformationsregel ist.

In Kapitel 6 wird erläutert, wie der hier vorgestellte Automatenfunktor implementiert werden kann. In Kapitel 7 schließlich wird ein Anwendungsbeispiel gegeben, an dem die Möglichkeiten des Automatenfunktors und des zuvor vorgestellten Algorithmus' als Mittel der Verifikation aufgezeigt wird.

Kapitel 6

Implementierung

Um die praktische Realisierbarkeit der theoretischen Erkenntnisse untersuchen zu können, wurde im Rahmen dieser Arbeit ein Programm entwickelt, mit dessen Hilfe automatisch der in Abschnitt 5.3 beschriebene Automatenfunktor für einen Subgraphen U erzeugt werden kann. Des Weiteren wurde mit diesem Programm der in Abschnitt 5.15 vorgestelle Algorithmus zum Überprüfen von Invarianten bei Graphsprachen implementiert. Dadurch soll zum vor allem gezeigt werden, wie die vorgestellte Theorie in der Praxis eingesetzt werden kann und welche Schwierigkeiten dabei auftreten können.

6.1 Aufbau des Programms

Die Implementierung des Programms erfolgte in der Programmiersprache *Java*, wobei zum Schreiben zum Schreiben von XML-Dateien außerdem die freie XML-API *JDOM* [15] in der Version 1.1 eingesetzt wurde. Die Bedienung des Programms ist wahlweise über die Kommandozeile (siehe Abbildung 6.1) oder über eine graphische Benutzeroberfläche (siehe Abbildung 6.2) möglich. Das Programm gliedert sich in insgesamt fünf verschiedene Programmpakete, denen die folgenden Teilbereiche zugeordnet sind:

- Globale Programmverwaltung
- Generierung und Verwaltung des Automatenfunktors
- Repräsentation und Manipulation von Hypergraphen

```
Berechnet den Automatenfunktor f³r die Sprache aller Graphen, die (mindestens) e
inen Subgraphen aus einer bestimmten Menge von Subgraphen enthalten.
> add Subgraph_Dreieck.xml Subgraph_Doppelkante.xml
Der Subgraph 'Subgraph_Dreieck' wurde hinzugef³gt.
Der Subgraph 'Subgraph_Doppelkante' wurde hinzugef³gt.
Die folgenden Subgraphen werden gesucht:
* Subgraph_Dreieck
* Subgraph_Doppelkante
HINWEIS: Sobald mindestens ein Subgraph erkannt wurde, akzeptiert der Automat de
n Eingabegraphen
> create 4 verbose
Die ausf³hrliche Ausgabe wurde aktiviert.
Berechne die Subgraphen des gesuchten Graphen und die partiellen Funktionen
Konfiguriere den Automaten
Erzeuge Subgraphtransitionen
Erzeuge Funktionstransitionen
Erzeuge Zustandsmengen des Automaten
Der Automat besteht aus insgesamt 1732 Zustőnden.
F³r die Interfacegr÷■e 0 gibt es 8 Zustőnde.
F³r die Interfacegr÷■e 1 gibt es 26 Zustőnde.
F³r die Interfacegr÷■e 2 gibt es 92 Zustőnde.
F³r die Interfacegr÷■e 3 gibt es 338 Zustőnde.
F³r die Interfacegr÷■e 4 gibt es 1268 Zustőnde.
Berechne die Subgraphen des gesuchten Graphen und die partiellen Funktionen
Konfiguriere den Automaten
```

Abbildung 6.1: Die Kommandozeile

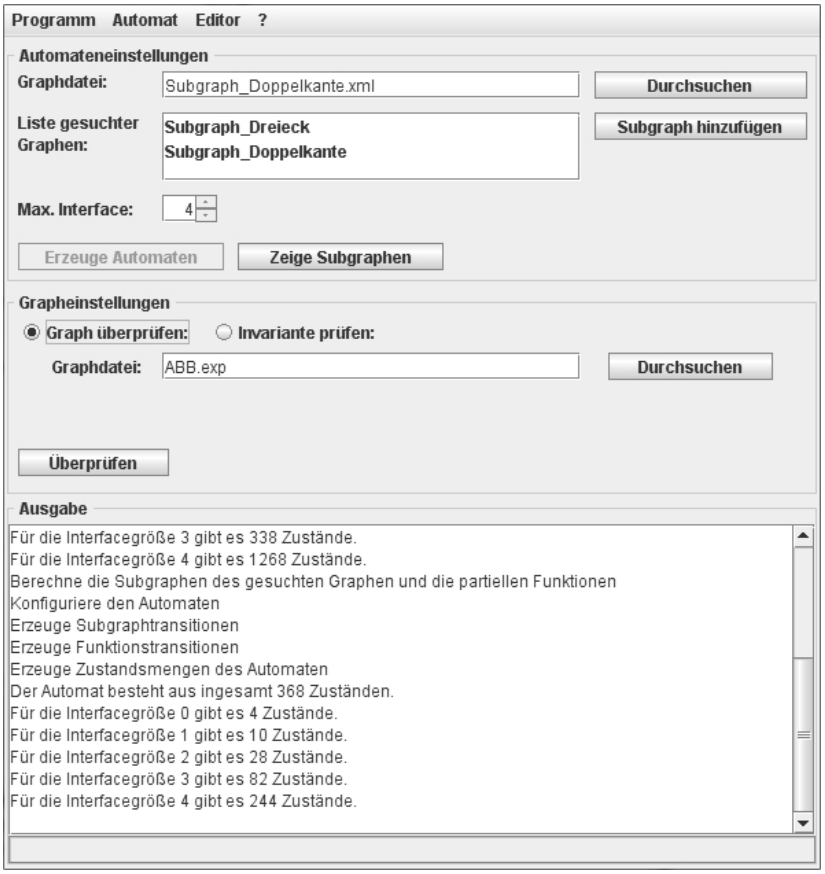


Abbildung 6.2: Das Hauptfenster

- Graphische Benutzeroberfläche
- Ereignisse und Ereignisverarbeitung

Das Paket für die globale Programmverwaltung stellt in erster Linie Klassen zum Starten (`Main`) und zur Konfiguration (`ConfigParser`) des Programms sowie zur textbasierten Steuerung (`MainShell`) und Protokollierung des Programmablaufs (`LogSystem`) bereit. Des Weiteren werden zwei Schnittstellen zur Ein- (`GraphReaderInterface`) bzw. zur Ausgabe (`GraphWriterInterface`) der benötigten Subraphen bereitgestellt. Implementiert werden diese Schnittstellen zum einen von der Klasse `XMLGraphReader`, die Subgraphen aus einer XML-Datei einliest, und zum anderen durch die Klassen `ShellGraphWriter` und `GraphVizWriter`, die Graphen in textueller Form auf dem Terminal bzw. in graphischer Form[1] in einer Bilddatei ausgeben. Außerdem werden durch die Klasse `XMLGraphReader` Methoden zur Ausnahmebehandlung für das Einlesen der Subgraph-XML-Dateien bereit gestellt.

Das Paket zur Generierung und Verwaltung des Automatenfunktors, welches das Kernstück des Programms darstellt, enthält verschiedene Klassen, die den

[1] Um die entsprechenden Grafiken erzeugen zu können, muss das Programm *GraphViz* (www.graphviz.org) installiert sein und der Installationspfad in der Konfigurationsdatei angegeben werden.

Automatenfunktor (`Automaton`), die Zustände (`State`) sowie die partiellen Funktionen (`Function`), die Teil der Zustände sind, modellieren. Außerdem existiert aus konzeptionellen Gründen ein Modul zum Zugriff auf den Automatenfunktor (`AutomatonSimulator`), um den direkten Zugriff von außen auf den Automatenfunktor bzw. die entsprechende Klasse zu unterbinden. Weiterhin enthält das Paket eine Schnittstelle (`AutomatonRelation`), welche die Ordnungsrelation, die durch den Algorithmus 4.16 berechnet wird, repräsentiert sowie eine Klasse (`SimulationRelation`), welche die Schnittstelle implementiert und den Algorithmus 5.15 berechnet. Die Schritte, die zur Erzeugung des Automatenfunktors notwendig sind, werden in Abschnitt 6.2 näher vorgestellt.

Das Paket für die Repräsentation und Manipulation von Hypergraphen bietet Klassen an, die zur programminternen Darstellung von Knoten (`Vertex`), Hyperkanten (`Edge`) und Hypergraphen (`Hypergraph`) sowie zur Verarbeitung von Graphoperationen (`GraphOperations`) dienen.

Die gesamte Funktionalität, welche für die graphische Bedienung benötigt wird, wird in dem Paket für die graphische Benutzeroberfläche zur Verfügung gestellt. Das Paket besteht im Einzelnen aus den Klassen zur Anzeige des Hauptfensters (`MainWindow`) und zur Anzeige von Dialogfenstern (`DialogOptionWindow`) sowie einer Klasse `GraphEditor`, die einen graphischen Editor darstellt, mit dem die zu Graphoperationen gehörigen Graphen visualiert[2] werden können (siehe auch Abschnitt 6.3).

Das Paket für Ereignisse und Ereignisverarbeitung enthält Ereignisse, die über Änderungen des Automatenfunktors (`ChangeEvent`) und über Protokollereignisse (`LogEvent`) informieren, sowie entsprechende Listener[3], die auf die Ereignisse reagieren (`ChangeListener` bzw. `LogListener`).

Eine detaillierte Beschreibung der einzelnen Pakete, Klassen und Schnittstellen sowie der bereitgestellten Methoden kann der *Javadoc*-Dokumentation, die auf der beiliegenden CD enthalten ist, entnommen werden.

6.2 Generierung des Automatenfunktors

Für die automatische Erzeugung des gesuchten Automatenfunktors sind folgenden Schritte notwendig, die im Anschluss näher vorgestellt werden sollen:

1. Konfiguration des Automatenfunktors
2. Erzeugung aller Subgraphen (des gesuchten Graphen)
3. Erzeugung aller partiellen Funktionen (welche die Knoten des Interfaces in die Knotenmenge des jeweiligen Subgraphen abbilden)
4. Erzeugung der Zustandsmengen des Automaten
5. Berechnung der Übergangsfunktion für jeden Subgraphen
6. Berechnung der Übergangsfunktion für jede partielle Funktion
7. Berechnung der Übergangsfunktion für jeden Zustand

[2] Für den Grapheditor ist ebenfalls das Programm *GraphViz* erforderlich. Siehe auch [2]
[3] vgl. Entwurfsmuster „Observer“

In Schritt eins erfolgt einerseits die Auswahl des gesuchten Graphen sowie andererseits die Einstellung der maximalen Interfacegröße (siehe Abschnitt 5.2). Um die gleichzeitige Suche nach mehreren Subgraphen zu ermöglichen, wodurch wiederum die gleichzeitige Überprüfung mehrerer Invarianten möglich wird, können auch mehrere Subgraphen ausgewählt werden. Allerdings müssen alle Subgraphen in einem selbstdefiniertem XML-Format vorliegen, da die XML-Dateien vor der weiteren Verarbeitung eingelesen und geparst werden, um diese anschließend in `Hypergraph`-Objekte umzuwandeln. Mit Hilfe des `Hypergraph`-Objekts bzw. der `Hypergraph`-Objekte für den Fall, dass mehrere Subgraphen ausgewählt wurden, sowie der Interfacegröße wird eine Instanz des `AutomatonSimulator` erzeugt, die zur Verwaltung und Steuerung des Automatenfunktors dient.

Die nachfolgenden Schritte zur Erzeugung eines Automatenfunktors werden zunächst für den Fall erläutert, dass in Schritt eins lediglich ein einzelner Subgraph ausgewählt wurde. Falls mehrere Subgraphen gesucht werden sollen, müssen diese Schritte entsprechend für jedes `Hypergraph`-Objekt wiederholt werden. Anschließend werden die einzelnen Automatenfunktoren zu einem gemeinsamen Automatenfunktor vereinigt.

Wurde in Schritt eins nur ein Subgraph ausgewählt, wird in Schritt zwei zunächst für den gesuchten Graphen die Menge aller Subgraphen bestimmt. Diese Menge enthält alle Graphen, die der Automatenfunktor während der Verarbeitung in den verschiedenen Zuständen erkennen kann. Für die Berechnung wird die Potenzmenge der Knotenmenge des gesuchten Graphen gebildet und anschließend über Teilmengen dieser Potenzmenge iteriert. In jedem Iterationsschritt werden die folgende Schritte ausgeführt (siehe auch: Programm 6.1):

1. Für die aktuelle Knotenmenge wird der zugehörige diskrete Graph erzeugt (Zeile 5 bis Zeile 10).

2. Für die aktuelle Knotenmenge wird die Potenzmenge der Menge der Hyperkanten bestimmt, die nur Knoten verbinden, die in der aktuellen Knotenmenge enthalten sind (Zeile 13).

3. Für jede Teilmenge aus der Hyperkantenpotenzmenge (die im vorherigen Schritt berechnet wurde) wird ein neuer Hypergraph erzeugt, der aus allen Knoten und Hyperkanten besteht, die in den aktuellen Knoten- und Hyperkantenteilmengen enthalten sind (Zeile 15 bis Zeile 30).

Nachdem über alle Knotenteilmengen iteriert wurde, sind alle Subgraphen des gesuchten Graphen erzeugt worden.

In Schritt drei wird die Menge aller partiellen Funktionen bestimmt, deren Definitionsbereich die Knotenmenge eines diskreten Graphen D_n ist, der durch die maximale Interfacegröße beschränkt wird, und deren Zielmenge die Knotenmenge eines Subgraphen ist, der in Schritt zwei erzeugt wurde. Um die Berechnung zu vereinfachen, werden die n Knoten des Definitionsbereichs durch die Zahlen von 0 bis $n - 1$ modelliert (siehe Zeile 1 bis Zeile 4, Programm 6.2). Außerdem wird jede partielle Funktion als totale Funktion betrachtet, deren Zielmenge um ein zusätzliches „Infimum“-/„Bottom“-Element erweitert wurde. Indem wie in Schritt zwei über die Knotenmengen der Subgraphen des gesuchten (Sub-)graphen iteriert wird, berechnet sich die Menge aller partiellen Funktionen demnach wie folgt (siehe auch: Programm 6.2):

```
1    //Iteriere über alle Teilmengen aus der Knotenpotenzmenge
2    for(vertexSetIt = vertexPowerset.iterator();vertexSetIt.hasNext();) {
3      vertexSubset = vertexSetIt.next();
4      //Erzeuge zuerst den diskreten Graphen
5      discreteHypergraph = new Hypergraph("Graph");
6      //Iteriere über die Knoten aus der Knotenteilmenge
7      for(vertexIt = vertexSubset.iterator();vertexIt.hasNext();) {
8        //füge alle Knoten aus der Teilmenge in den diskreten Graphen ein
9        discreteHypergraph.createVertex(vertexIt.next());
10     }
11     //Berechne die Kantenpotenzmenge, die nur Kanten enthält, deren Knoten in der
12     //aktuellen Knotenmenge enthalten sind
13     edgePowerset = getPowerset(wantedGraph.getEdgeSet(vertexSubset));
14     //Iteriere über alle Teilmengen aus der Kantenpotenzmenge
15     for(edgeSetIt = edgePowerset.iterator();edgeSetIt.hasNext();) {
16       //Entnehme nächste Teilmenge
17       edgeSubset = edgeSetIt.next();
18       //Erzeuge einen neuen Graphen aus dem aktuellen diskreten Graphen
19       tempHypergraph = discreteHypergraph.clone();
20       tempHypergraph.setGraphName(tempHypergraph.getGraphName()+"-"+
             graphCounter);
21       //Iteriere über die Kanten aus der Kantenteilmenge
22       for(edgeIt = edgeSubset.iterator();edgeIt.hasNext();) {
23         //füge alle Kanten aus der aktuellen Kantenmenge in den neuen Graphen ein
24         tempHypergraph.createEdge(edgeIt.next());
25       }
26       //Neuen Subgraphen hinzufügen
27       subgraphSet.add(tempHypergraph);
28       //Zähler hochzählen
29       graphCounter++;
30     }
31   }
```

Programm 6.1: Methode zur Erzeugung aller Subgraphen (des gesuchten Subgraphen)

1. Die aktuelle Knotenteilmenge wird um das „Bottom“-Element erweitert (Zeile 11 und Zeile 12).

2. Für jede Knotenmenge eines diskreten Graphen wird die Menge aller Funktionen bestimmt, die von dieser Knotenmenge in die aktuelle Knotenmenge des aktuellen Subgraphen des gesuchten Graphen abbilden (Zeile 17).

3. Die so berechnete Funktionenmenge wird der gesuchten Menge aller partiellen Funktionen hinzugefügt (Zeile 20).

Für die Erzeugung der Zustandsmengen des Automaten in Schritt vier wird die Menge aller Paare bestehend aus den Subgraphen, die in Schritt zwei berechnet wurden, und den in Schritt drei berechneten, partiellen Funktionen, deren Zielmenge die Knotenmenge des jeweiligen Subgraphen ist, gebildet. Die Menge der Startzustände enthält als Einzigen den Zustand, der aus dem leeren Graphen und der leeren Funktion besteht. Der einzige Endzustand ist der Zustand, dessen Graph der gesuchte Subgraph ist und dessen Funktion ebenfalls die leere Funktion ist (siehe auch die Definition der Start- und Endzustände in Abschnitt 5.3).

```
1   for(int i=0;i<=maxInterfaceSize;i++) {
2     finNumSets.add(new HashSet<String>(finNumSet));
3     finNumSet.add(String.valueOf(i));
4   }
5   //Iteriere über alle Teilmengen aus der Knotenpotenzmenge
6   for(vertexSetIt = vertexPowerset.iterator();vertexSetIt.hasNext();) {
7     vertexSubset = vertexSetIt.next();
8     //Funktionsmenge für die aktuelle Knotenteilmenge vorbereiten
9     functionSets.put(vertexSubset.toString(),
10          new HashSet<Function<String, Vertex>>());
11    tempVertexSet = new HashSet<Vertex>(vertexSubset);
12    tempVertexSet.add(Vertex.undefinedVertex);
13    //Durchlaufe alle endlichen Teilmengen (bzw. Interfaces)
14    for(finSetIt = finNumSets.iterator();finSetIt.hasNext();) {
15      finNumSet = finSetIt.next();
16      //Berechne alle Funktionen zw. dem aktuellen Interface und der Knotenmenge
17      set = getAllFunctions(finNumSet, tempVertexSet,
18              Vertex.undefinedVertex);
19      //und füge die berechnete Menge in die Funktionenmenge ein
20      functionSets.get(vertexSubset.toString()).addAll(set);
21    }
22  }
```

Programm 6.2: Methode zur Erzeugung aller partiellen Funktionen (von der Interfaceknotenmenge in die Subgraphknotenmenge)

Die Berechnung der Übergangsfunktion für jeden Subgraphen in Schritt fünf dient zur Vorbereitung der Zustandsübergangsfunktion. Für jeden Subgraphen, der in Schritt zwei erzeugt wurde, und für jede atomare Graphoperation (siehe Definition 5.4) wird die Menge der Subgraphen bestimmt, die nach Anwendung der jeweiligen Graphoperation auf den aktuellen Subgraphen erkannt werden können. Die einzelnen Graphoperationen wirken sich wie folgt auf die einzelnen Subgraphen aus:

- $\mathsf{edge}_n^{A,m}$: Die Menge der „Nachfolgegraphen“ besteht aus dem aktuellen Subgraphen und allen Graphen, die man erhält, wenn in dem aktuellen Subgraphen eine m-stellige, mit A-beschriftete Hyperkante einfügt wird. Das heißt, falls die Knoten der neuen Hyperkante noch nicht erkannt wurden, wird zunächst die Knotenmenge um die Knoten erweitert, die mit der neuen Hyperkante verbunden sind. Ansonsten werden der Knotenmenge nur Knoten hinzugefügt, die noch nicht erkannt wurden. Anschließend wird die Hyperkantenmenge um die neue Hyperkante erweitert.

- vertex_n: Die Menge der „Nachfolge“-Graphen besteht aus dem aktuellen Subgraphen und allen Graphen, die man erhält, wenn man in dem aktuellen Subgraphen einen einzelnen, isolierten Knoten hinzufügt.

- fuse_n, perm_n, res_n, trans_n: Diese Operationen haben keine Auswirkungen auf den erkannten Subgraphen, sondern wirken sich nur auf die partielle Funktion aus.

In Schritt sechs wird die Übergangsfunktion für jede partielle Funktion, die in Schritt drei erzeugt wurde, berechnet. Durch die Übergangsfunktion wird die Menge

der partiellen Funktionen bestimmt, die durch Anwendung der Graphoperationen auf die aktuelle partielle Funktion entstehen. Dies dient ebenfalls zur Vorbereitung der Berechnung der Zustandsübergangsfunktion. Je nach Graphoperation lässt sich die Übergangsfunktion wie folgt bestimmen:

- $\mathsf{edge}_n^{A,m}$: Die Menge der „Nachfolgefunktionen“ besteht aus allen partiellen Funktionen, deren Definitionsbereich um m Elemente größer ist als der der ursprünglichen Funktion, und die für den restlichen Definitionsbereich übereinstimmen.[4]

- vertex_n: Die Menge der „Nachfolgefunktionen“ besteht aus allen partiellen Funktionen, deren Definitionsbereich um ein Element größer ist als der der ursprünglichen Funktion, und die für den restlichen Definitionsbereich übereinstimmen.[4]

- fuse_n: Die Menge der „Nachfolgefunktionen“ besteht entweder aus der partiellen Funktion, aus deren Definitionbereich das letzte Element (bezüglich der Knotennummerierung) entfernt wurde und die ansonsten mit der ursprünglichen Funktion übereinstimmt, falls die Bilder der ersten beiden Knoten (bezüglich der Knotennummerierung) gleich sind, oder ist ansonsten die leere Menge.

- perm_n: Die Menge der „Nachfolgefunktionen“ besteht aus der partiellen Funktion, die entsteht, wenn jedes Element des Definitionsbereichs auf das Bild des Nachfolgeelementes (bezüglich der Knotennummerierung der ursprünglichen Funktion) abgebildet wird.

- res_n: Die Menge der „Nachfolgefunktionen“ besteht entweder aus der partiellen Funktion, aus dessen Definitionsbereich das letzte Element (bezüglich der Knotennummerierung) entfernt wurde und die für die übrigen Elemente mit der ursprünglichen partiellen Funktion übereinstimmt, falls die ursprüngliche Funktion injektiv ist, oder ist ansonsten die leere Menge.

- trans_n: Die Menge der „Nachfolgefunktionen“ besteht aus der partiellen Funktion, deren Bilder für die ersten beiden Elemente (bezüglich der Knotennummerierung) des Definitionsbereichs vertauscht sind und für die restlichen Elemente des Definitionsbereichs übereinstimmen.

Nachdem die Subgraph- und Funktionsübergangsfunktion berechnet wurde, kann in Schritt sieben die eigentliche Zustandsübergangsfunktion aufgestellt werden. Es ist zu beachten, dass in den Schritten fünf und sechs zwar eine Vorauswahl stattgefunden hat, allerdings muss an dieser Stelle nochmals eine Prüfung der Subgraphen und Funktionen in Abhängigkeit der Graphoperation durchgeführt werden. Ansonsten enthalten die Nachfolgezustandsmengen gegebenenfalls unzulässige Nachfolgezustände:

- $\mathsf{edge}_n^{A,m}$: Die Menge der Nachfolgezustände besteht aus allen Zuständen, die einen durch die Subgraphübergangsfunktion bestimmten Subgraphen

[4] Eine genauere Einschränkung der zulässigen Funktionen ist an dieser Stelle nicht möglich, da keine Informationen darüber vorliegen, welche Hyperkante bzw. welcher Knoten hinzugefügt wurde.

erkennen und die Knoten des diskreten Graphen entsprechend einer durch die Funktionsübergangsfunktion bestimmten partiellen Funktion abbilden. Außerdem muss für jede partielle Funktion gelten, dass die letzten m Knoten des Definitionsbereichs auf die m Knoten der neuen, mit A beschrifteten Hyperkante (in der Reihenfolge, in der die Knoten an die Hyperkante angebunden sind) abgebildet werden.

- vertex_n: Die Menge der Nachfolgezustände besteht aus allen Zuständen, deren Definitionsbereich um ein Element größer ist als der, der ursprünglichen Funktion, und die für den restlichen Definitionsbereich übereinstimmen.

- fuse_n, perm_n, res_n, trans_n: Die Menge der Nachfolgezustände besteht aus dem Zustand, der den durch die Subgraphübergangsfunktion bestimmten Subgraphen erkennt und die Knoten des diskreten Graphen entsprechend der durch die Funktionsübergangsfunktion bestimmten partiellen Funktion abbildet. Falls die partielle Funktion nicht existiert, so existiert auch kein Nachfolgezustand für den jeweiligen Zustand und die entsprechende Graphoperation.

Anschließend werden die Subgraph- und die Funktionsübergangsfunktion gelöscht, da diese im Weiteren nicht mehr benötigt werden.

Falls in Schritt eins ein einzelner Graph ausgewählt wurde, ist die Erzeugung des Automatenfunktors an dieser Stelle beendet. Falls jedoch mehrere Subgraphen gesucht werden, wird der oben beschriebene Vorgang ab dem zweiten Schritt wiederholt. Sobald der zweite Automatenfunktor erzeugt wurde, werden beide Automatenfunktoren vereinigt, indem sowohl die Zustandsmengen als auch die Übergangsfunktionen der beiden Automatenfunktoren vereinigt werden. Während dieser Vereinigung werden die Zustände der beiden Automatenfunktoren umbenannt, so dass eine disjunkte Vereinigung stattfindet. Die Erzeugung und Vereinigung weiterer Automatenfunktoren wird für jeden gesuchten Graphen wiederholt, der in Schritt eins ausgewählt wurde.

Sobald die Vereinigung aller (Teil-)Automatenfunktoren abgeschlossen ist, ist die Erzeugung des gesuchten Automatenfunktors abgeschlossen.

6.3 Der Grapheditor

Da die korrekte Komposition der Graphoperationen zur Konstruktion eines vorgegeben Cospans mitunter sehr komplex sein kann, ist während der Entwicklung dieses Programms auch ein Werkzeug zur Speicherung und Visualisierung von Graphoperationen entstanden. Mit Hilfe des so genannten Grapheditors kann eine Liste von Graphoperationen erzeugt und gleichzeitig mittels *GraphViz* angezeigt werden. Auf diese Weise kann überprüft werden, ob durch die Komposition der einzelnen Graphoperationen der korrekte Cospan beschrieben wird.

In Abbildung 6.3 ist die Benutzeroberfläche des Grapheditors dargestellt. Das Fenster teilt sich in drei Bereiche auf. Den Hauptbereich bildet die Anzeigefläche zur Visualisierung der Cospans (❶). Dabei sind die Knoten des zum Cospan gehörigen Graphen als Ellipsen und die Hyperkanten als Rechtecke dargestellt. Die Verknüpfungsfunktion der einzelnen Hyperkanten ergibt sich durch die Nummerierung der Verbindungslinien zwischen Hyperkante und Knoten. Die Zahlen

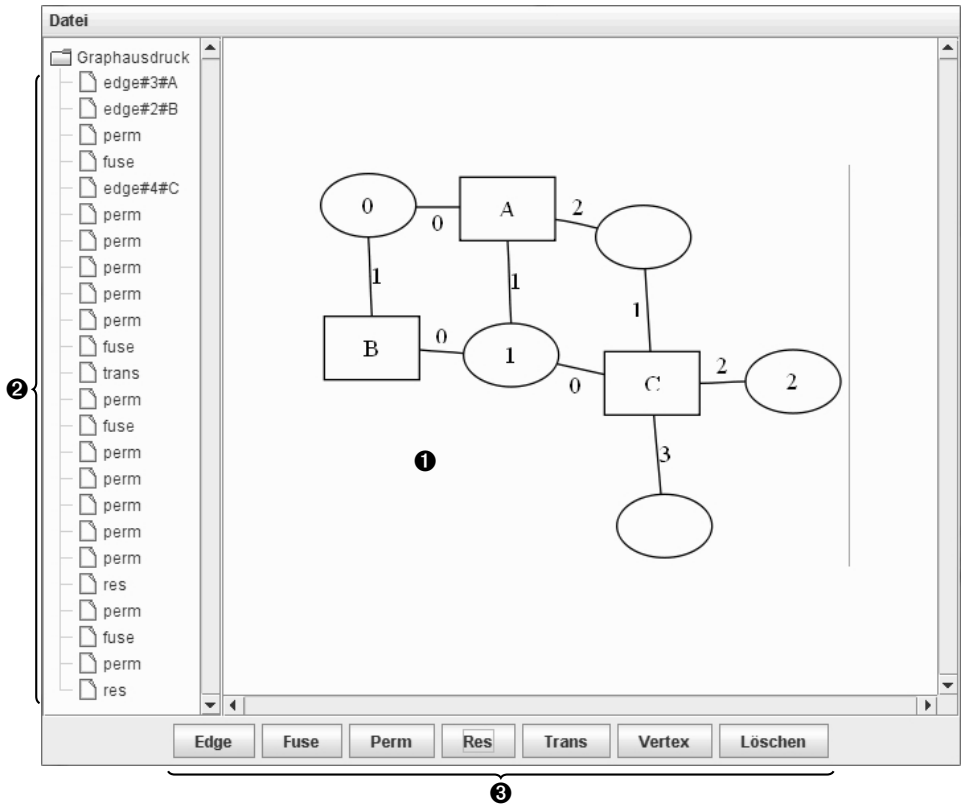


Abbildung 6.3: Benutzeroberfläche des Grapheditors

innerhalb der Ellipsen markieren die Knoten, die in dem äußeren Interface des Cospans liegen, entsprechend ihrer Reihenfolge. Auf der linken Seite befindet sich der Bereich zur Darstellung der Liste der Graphoperationen (❷). An der Unterseite des Fenster befinden sich die Schaltflächen, mit denen der Liste weitere Graphoperationen hinzugefügt werden können (❸).

Durch Betätigen der einzelnen Schaltflächen wird zum einen der Graph und das äußere Interface des Cospans entsprechend der ausgewählten Graphoperation geändert, zum anderen wird die ausgewählte Graphoperation ans Ende der Liste angefügt. Sobald der gesuchte Cospan konstruiert wurde und die Liste der Graphoperation vollständig ist, kann diese über das Datei-Menü gespeichert werden. Es ist ebenfalls möglich, gespeicherte Listen von Graphoperationen zu laden und nachträglich zu erweitern.

Im nachfolgenden Kapitel können nun die in dieser Arbeit vorgestellten Ansätze mit Hilfe dieses Programms auf ein Beispiel aus dem Bereich der verteilten und dynamischen Systeme angewandt werden.

Kapitel 7

Verifikation von Zugriffsregeln für ein Mehrbenutzersystem

In diesem Kapitel sollen die zuvor vorgestellte Theorie und das entwickelte Programm auf ein Beispiel aus dem Bereich der verteilten und dynamischen Systeme angewandt werden, um die Einsatzmöglichkeiten sowie Vor- und Nachteile dieses Ansatzes aufzuzeigen. Dazu soll mit Hilfe des Algorithmus' zur Überprüfung von Invarianten bei Graphsprachen die Sicherheit von Zugriffsregeln für ein Mehrbenutzersystem verifiziert werden.

7.1 Modellierung des Mehrbenutzersystems

Um die Möglichkeiten des Algorithmus' zur Überprüfung von Invarianten aufzuzeigen, werden in diesem Abschnitt die Schritte zur Modellierung eines (verteilten) Mehrbenutzersystems vorgestellt. Dieses Mehrbenutzersystem soll über die folgenden Zugriffsregeln zum Gewähren, Entziehen sowie Ändern von Zugriffsrechten verfügen:

- Benutzer mit Leserecht auf Objekt erzeugen
- Benutzer mit Schreibrecht auf Objekt erzeugen
- Neues Objekt mit Leserecht erzeugen
- Neues Objekt mit Schreibrecht erzeugen
- Benutzer mit Leserecht wechseln
- Benutzer mit Schreibrecht wechseln
- Objekt mit Leserecht wechseln
- Objekt mit Schreibrecht wechseln
- Objekte mit Leserecht tauschen
- Objekte mit Schreibrecht tauschen
- Leserecht entziehen
- Schreibrecht entziehen
- Leserecht in Schreibrecht umwandeln
- Schreibrecht in Leserecht umwandeln

- Benutzer löschen
- Objekt löschen

Das Mehrbenutzersystem kann als ein Hypergraph modelliert werden, dessen Knoten die Benutzer und Objekte des Systems sind und dessen Hyperkanten die Zugriffsrechte zwischen Benutzern und Objekten darstellen. Es ist allerdings nicht notwendig das gesamte Modell des Mehrbenutzersystem zu betrachten, stattdessen müssen für die Verifikation sicherheitskritische Strukturen definiert werden.

Im Falle des Mehrbenutzersystems ist vor allem der schreibende Zugriff auf Objekte als kritisch zu betrachten. Es sind zwei Fälle zu berücksichtigen:

- Ein Benutzer besitzt zweifachen, schreibenden Zugriff auf dasselbe Objekt (Fall: *Doppelter Zugriff*).
- Zwei Benutzer besitzen schreibenden Zugriff auf dasselbe Objekt (Fall: *Zwei Benutzer*).

Das Mehrbenutzersystem ist genau dann sicher, wenn es nicht möglich ist, einen Systemzustand zu erreichen, der gegen diese Sicherheitsbedingungen verstößt. Um dies zu überprüfen, können die beiden sicherheitskritischen Fälle, wie in Abbildung 7.1 dargestellt, modelliert werden. Dabei sei, wie bereits oben beschrieben, angenommen, dass sowohl Benutzer als auch Objekte als Knoten dargestellt werden. Zur besseren Unterscheidung von Benutzer- bzw. Objektknoten werden in den folgenden Darstellungen erstere mit einem u (für „user“) und letzere mit einem o (für „object“) beschriftet. Diese „Typisierung“ der Knotenmenge ist für die eigentliche Modellierung des Systems nicht vorgesehen. Allerdings könnte dies durch Anfügen von einstelligen, mit u bzw. o beschrifteten Hyperkanten, an die einzelnen Knoten nachgebildet werden. Durch diese Typisierung kann dann sichergestellt werden, dass sinnlose Zugriffsrechte, wie zum Beispiel der lesende Zugriff von einem Objekt auf ein anderes Objekt ausgeschlossen sind. Um die Komplexität des Systemmodells nicht unnötig zu erhöhen, soll im Folgenden angenommen werden, dass solche sinnlosen Zugriffsrechte nicht auftreten.

Wenn ein Benutzer schreibenden Zugriff auf ein Objekt hat, wird dies dadurch modelliert, dass von dem Knoten, der den Benutzer darstellt, eine (Hyper-)Kante, die mit W (für „writeable access“) beschriftet ist, zu dem Knoten führt, der das Objekt darstellt. Entsprechend wird der lesende Zugriff auf ein Objekt durch eine mit R (für „readable access“) beschriftete (Hyper-)Kante dargestellt.

(a) Subgraph: Doppelter Zugriff (b) Subgraph: Zwei Benutzer

Abbildung 7.1: Modellierung der sicherheitskritischen Strukturen

Diese beiden Graphen dienen im Folgenden als die gesuchten Subgraphen des Automatenfunktors aus Kapitel 5.3. Wie in Kapitel 6.2 erläutert wurde, werden zunächst für jeden Subgraphen ein eigener Automatenfunktor erzeugt und diese im Nachhinein vereinigt, um den gesuchten Automatenfunktor zu erhalten.

Als Nächstes müssen die oben genannten Zugriffsregeln geeignet dargestellt werden. Dazu werden diese als Graphtransformationsregeln modelliert. Wie in Kapitel 5 erwähnt, können die linke bzw. die rechte Seite einer Graphtransformationsregel

$$p\colon L \xleftarrow{\varphi^L} I \xrightarrow{\varphi^R} R$$

als Cospans

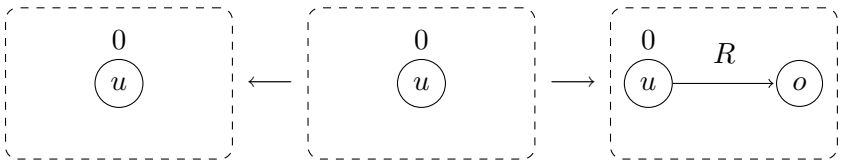
$$\ell\colon \emptyset \to L \xleftarrow{\varphi^L} I \quad \text{bzw.} \quad r\colon \emptyset \to R \xleftarrow{\varphi^R} I$$

aufgefasst werden. Diese Tatsache wird in Abschnitt 7.2 zur Verifikation der Zugriffsregeln benötigt. Im Folgenden werden die Spans zur Modellierung der Zugriffsregeln vorgestellt:

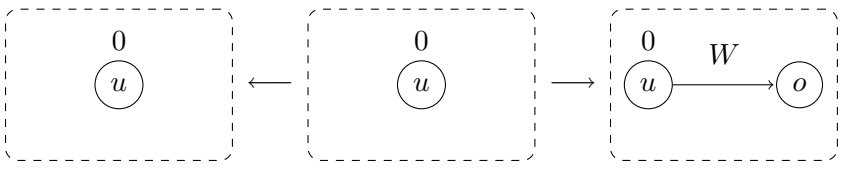
- Benutzer mit Leserecht auf Objekt erzeugen:

- Benutzer mit Schreibrecht auf Objekt erzeugen:

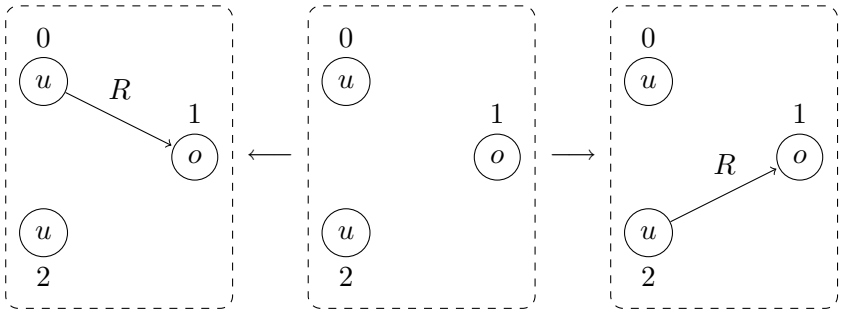
- Neues Objekt mit Leserecht erzeugen:



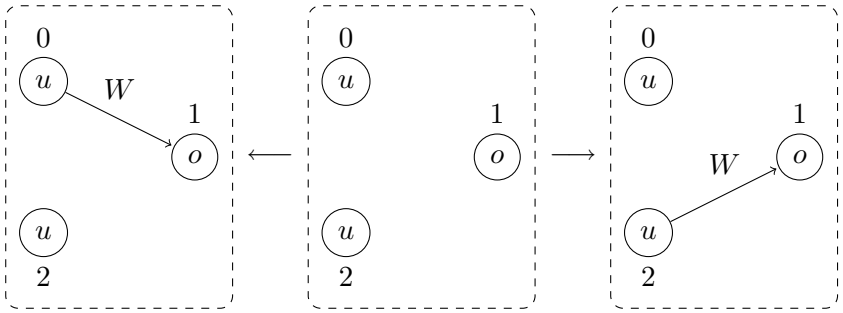
- Neues Objekt mit Schreibrecht erzeugen:



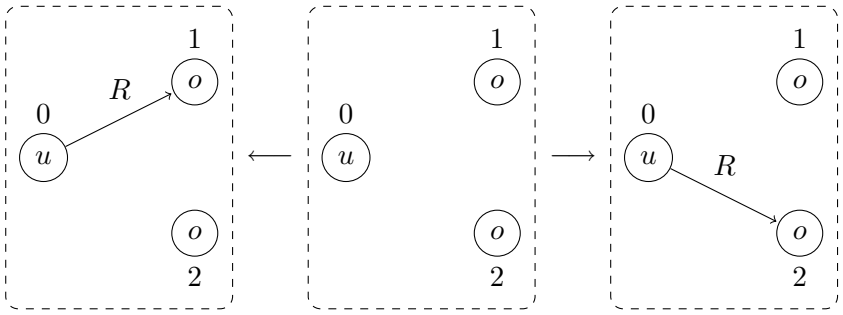
- Benutzer mit Leserecht wechseln:



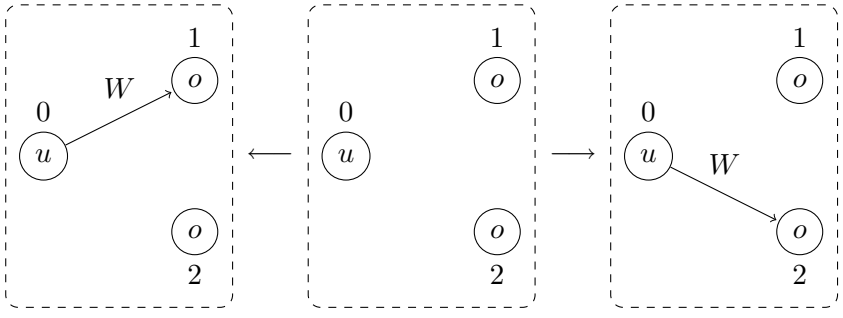
- Benutzer mit Schreibrecht wechseln:



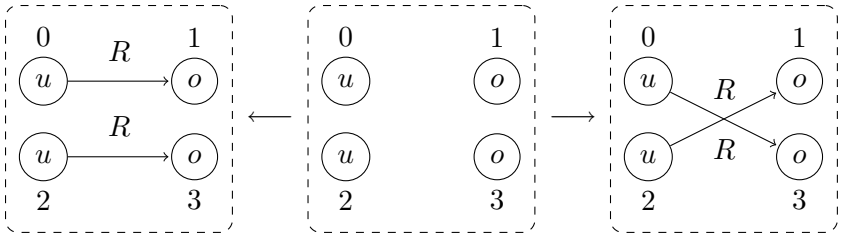
- Objekt mit Leserecht wechseln:



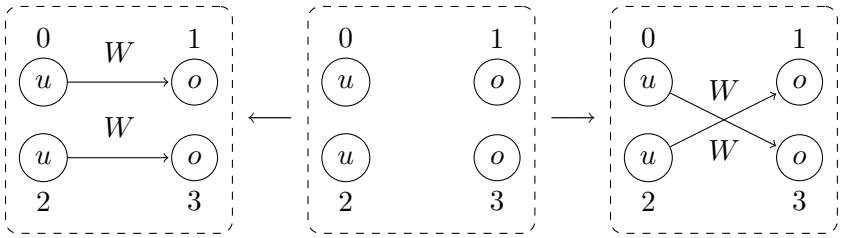
- Objekt mit Schreibrecht wechseln:



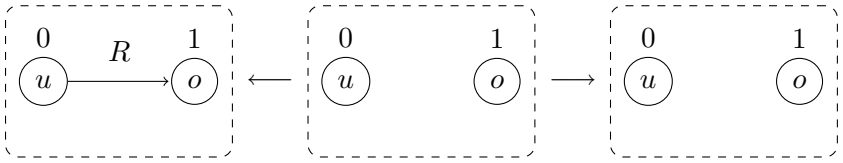
- Objekte mit Leserecht tauschen:



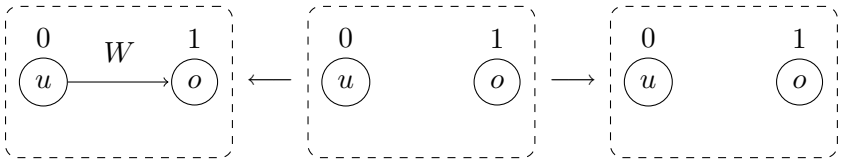
- Objekte mit Schreibrecht tauschen:



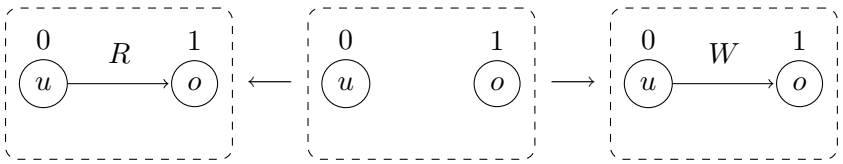
- Leserecht entziehen:



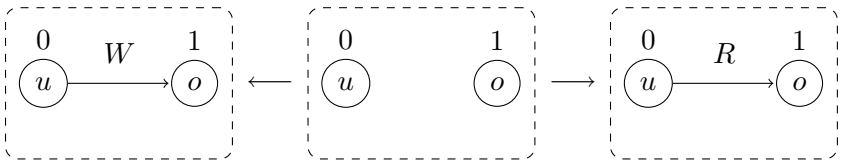
- Schreibrecht entziehen:



- Leserecht in Schreibrecht umwandeln:



- Schreibrecht in Leserecht umwandeln:



- Benutzer löschen:

- Objekt löschen:

Diese Zugriffsregeln bilden die Grundlage für weitergehende Betrachtungen hinsichtlich der Sicherheit des Mehrbenutzersystems.

7.2 Verifikation des Mehrbenutzersystems

Die Ergebnisse der Verifkation der in Abschnitt 7.1 definierten Zugriffsregeln sollen nun in diesem Abschnitt vorgestellt und diskutiert werden. Zunächst wird beschrieben, wie die Zugriffsregeln mit Hilfe des Algorithmus' 5.15 auf die vorgestellten Sicherheitsbedingungen überprüft werden können.

Um nachzuweisen, dass die Zugriffsregeln des Mehrbenutzersystems nicht gegen die oben definierten Sicherheitsanforderungen verstoßen, muss gezeigt werden, dass die Sprache aller Graphen, welche die Subgraphen „Doppelter Zugriff" und „Zwei Benutzer" enthalten, eine Invariante bezüglich der einzelnen Zugriffsregeln ist. Das bedeutet, dass eine Zugriffsregel die Sicherheitsanforderungen genau dann erfüllt, wenn es durch die Anwendung dieser Zugriffsregel nicht möglich ist, einen Graphen zu konstruieren, der mindestens einen der beiden sicherheitskritischen Subgraphen enthält.

Durch den in Abschnitt 5.3 beschriebenen Automatenfunktor können nur solche Graphen akzeptiert werden, die einen bestimmten Subgraphen enthalten. Für den Nachweis der Sicherheit muss allerdings gezeigt werden, dass bestimmte Graphen *nicht* enthalten sind. Daher ist ein indirekter Beweis der Sicherheit notwendig. Statt zu zeigen, dass die Graphen „Doppelter Zugriff" und „Zwei Benutzer" nicht durch die Zugriffsregeln konstruiert werden können, wird die folgende, stärkere Behauptung bewiesen: Falls ein Graph nach der Anwendung einer Zugriffsregel gegen die Sicherheit des Systems verstößt, hat dieser Verstoß bereits vor der Anwendung der Zugriffsregel vorgelegen.

Diese *Rückwärtsanalyse* funktioniert wie folgt: Wie in Kapitel 5 beschrieben, können die Graphtransformationsregeln in zwei Cospans ℓ und r für die linke bzw. rechte Seite aufgeteilt werden. Diese bilden die Eingabe für den Algorithmus 5.15 zur Überprüfung von Invarianten. Es wird nun geprüft, ob $r \leq_S^k \ell$ für eine feste Schranke $k \in \mathbb{N}$ gilt. Dass heißt, es wird geprüft, ob die Sprache aller Graphen, die die sicherheitskritischen Subgraphen enthalten, eine Invariante bezüglich der (umgekehrten,) getesteten Zugriffsregel ist.

Damit der Algorithmus anwendbar ist, muss allerdings sichergestellt sein, dass die Schranke k so gewählt ist, dass die zu überprüfenden Cospans mit Hilfe der in Definition 5.4 vorgestellten Graphoperationen konstruiert werden können. Für die obigen Zugriffsregeln bedeutet dies, dass $k \geq 4$ sein muss, weil zur Konstruktion

der Zugriffsregel „Objekte mit Leserecht tauschen“ bzw. „Objekte mit Schreibrecht tauschen“ eine Interfacegröße von vier benötigt wird.

An dieser Stelle werden die erzielten Ergebnisse für die einzelnen Zugriffsregeln vorgestellt und die nicht-verifizierbaren Zugriffsregeln genauer betrachtet, um die Gründe, die zu der Ablehnung der Zugriffsregeln durch den Algorithmus führten, zu erläutern.

Für die folgenden Zugriffsregeln konnte verifiziert werden, dass sie nicht gegen die in Abschnitt 7.1 definierten Sicherheitsbedingungen verstoßen. Das heißt, dass der Algorithmus zur Überprüfung von Invarianten für diese Zugriffsregeln festgestellt hat, dass die rechte Seite in (beschränkter) Simulationsrelation zur linken Seite steht:

- Benutzer mit Leserecht auf Objekt erzeugen
- Benutzer mit Schreibrecht auf Objekt erzeugen
- Neues Objekt mit Leserecht erzeugen
- Neues Objekt mit Schreibrecht erzeugen
- Benutzer mit Leserecht wechseln
- Objekt mit Leserecht wechseln
- Objekte mit Leserecht tauschen
- Leserecht entziehen
- Schreibrecht entziehen
- Schreibrecht in Leserecht umwandeln
- Benutzer löschen
- Objekt löschen

Somit ist die Sicherheit dieser Zugriffsregeln nachgewiesen. Für die folgenden Zugriffsregeln konnte dies nicht gezeigt werden:

- Benutzer mit Schreibrecht wechseln
- Objekt mit Schreibrecht wechseln
- Objekte mit Schreibrecht tauschen
- Leserecht in Schreibrecht umwandeln

Dafür gibt es verschiedene Ursachen. Die Zugriffsregeln „Leserecht in Schreibrecht umwandeln“ und „Objekt mit Schreibrecht wechseln“ verstoßen gegen die Sicherheitsanforderungen. Der Verstoß gegen die Sicherheitsanforderung „Zwei Benutzer“ kann wie folgt gezeigt werden:

- Für die Zugriffsregel „Leserecht in Schreibrecht umwandeln“:
  Nach zweifacher Anwendung der Zugriffsregel „Benutzer mit Leserecht auf Objekt erzeugen“ existieren zwei Benutzer, die jeweils lesenden Zugriff auf zwei unterschiedliche Objekte haben (Abbildung 7.2(a)). Anschließend kann einer der beiden Benutzer mit der Zugriffsregel „Objekt mit Leserecht tauschen“ einen lesenden Zugriff auf das Objekt des anderen Benutzers erhalten (Abbildung 7.2(b)). Nach zweifacher Anwendung der Zugriffsregel „Leserecht in Schreibrecht umwandeln“ für beide Benutzer haben diese schreibenden Zugriff auf dasselbe Objekt (Abbildung 7.2(c)). Dies sollte allerdings unterbunden werden. Der Verstoß gegen die Sicherheitsanforderung „Doppelter Zugriff“ kann in ähnlicher Weise gezeigt werden.

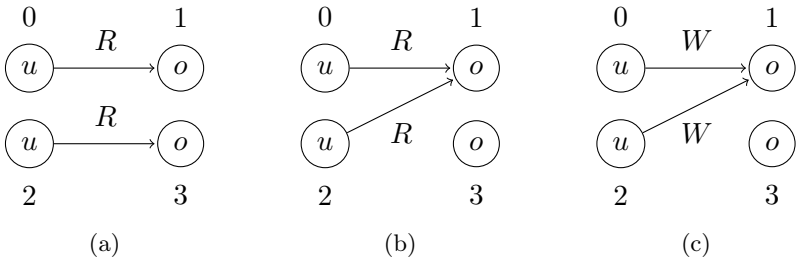


Abbildung 7.2: Verbotener Ablauf für die Zugriffsregel „Leserecht in Schreibrecht umwandeln“

- Für die Zugriffsregel „Objekt mit Schreibrecht wechseln“:
  Nach zweifacher Anwendung der Zugriffsregel „Benutzer mit Schreibrecht auf Objekt erzeugen“ existieren zwei Benutzer, die jeweils schreibenden Zugriff auf zwei unterschiedliche Objekte haben (Abbildung 7.3(a)). Wird nun die Zugriffsregel „Objekt mit Schreibrecht wechseln“ auf die beiden, soeben erzeugten Objekte angewandt, so haben zwei Benutzer wiederum schreibenden Zugriff auf dasselbe Objekt (Abbildung 7.3(b)).

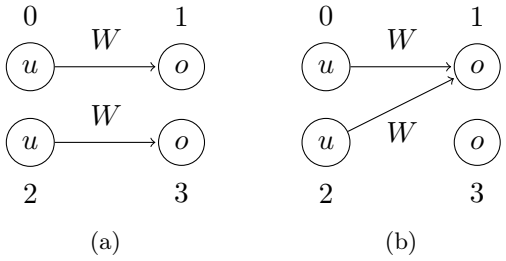
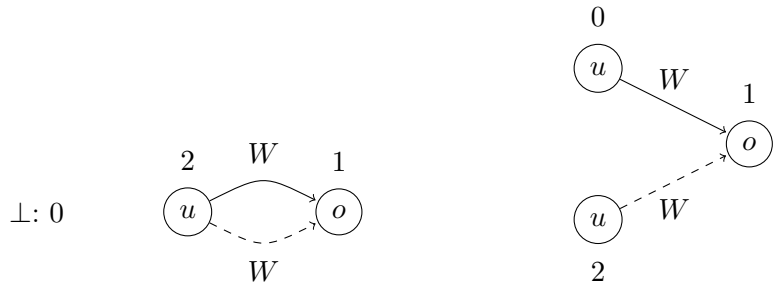


Abbildung 7.3: Verbotener Ablauf für die Zugriffsregel „Objekt mit Schreibrecht wechseln“

Die Zugriffsregel „Benutzer mit Schreibrecht wechseln“ verstößt nicht gegen die Sicherheitsanforderungen, dennoch wird sie von dem Algorithmus abgelehnt. Der Grund dafür ist in der verwendeten Unterapproximation zu suchen. Denn obwohl die rechte Seite der Zugriffsregel „Benutzer mit Schreibrecht wechseln“ in

Relation bezüglich der Myhill-Graphquasiordnung[1] zu der linken Seite steht, gilt dies nicht für die (beschränkte) Simulationsrelation. Das Problem besteht darin, dass der Automatenfunktor beim Einlesen der rechten Seite der Zugriffsregel in einen Zustand übergehen kann, der nicht durch die Zustände simuliert werden kann, die durch Einlesen der linken Seite der Zugriffsregel erreichbar sind. Dieses Problem soll anhand bestimmter Zustände, die beim Verifizieren der Zugriffsregel „Benutzer mit Schreibrecht wechseln“ erreicht werden können, genauer erläutert werden:

Der entscheidende Zustand, der beim Einlesen der rechten Seite erreicht, aber nicht simuliert werden kann, ist der in Abbildung 7.4(a) dargestellte Zustand s_R. Der Automatenfunktor hat sich dafür entschieden, den Subgraphen „Doppelter Zugriff“ zu erkennen – die gestrichelte Hyperkante soll andeuten, dass diese zum Erkennen des Subgraphen noch fehlt. Damit der Automatenfunktor auch beim Einlesen der linken Seite nach wie vor die Möglichkeit hat, einen Endzustand zu erreichen, muss dieser versuchen den Subgraphen „Zwei Benutzer“ zu erkennen. Dazu muss er in den in Abbildung 7.4(b) dargestellten Zustand s_L übergehen – durch die gestrichelte Hyperkante wird wiederum angedeutet, dass diese noch fehlt, um einen Endzustand zu erreichen.



(a) Zustand s_R für die rechte Seite (b) Zustand s_L für die linke Seite

Abbildung 7.4: Kritische Zustände der Zugriffsregel „Benutzer mit Schreibrecht wechseln“

Die restlichen Zustände, die beim Einlesen der linken Seite erreichbar sind, – bis auf den symmetrischen Zustand, bei dem die jeweils andere Hyperkante erkannt wird – sind auszuschließen. Es wäre in diesen Zuständen nicht für jeden Fall möglich, einen Endzustand zu erreichen, für den der Zustand s_R in einen Endzustand übergeht. Für den Zustand s_L ist dies anders: Zum Erreichen eines Endzustands muss zunächst die fehlende Hyperkante erkannt werden. Falls von dem Zustand s_R nun die fehlende Hyperkante erkannt wird, kann der Zustand s_L entsprechend darauf reagieren und ebenfalls die gesuchte Hyperkante erkennen. Falls die fehlende Hyperkante nicht erkannt wird, reagiert der Zustand s_L entsprechend dem Zustand s_R. Somit kann der Zustand s_L offensichtlich genau dann einen Endzustand erreichen, wenn dies auch der Zustand s_R kann. Die beiden Zustände s_L und s_R sind somit *sprachäquivalent*.

Allerdings kann der Zustand s_L den Zustand s_R nicht simulieren. Dies lässt sich

[1] Dies wird an dieser Stelle angenommen, da es mit Hilfe der in dieser Arbeit vorgestellten Mittel nicht bewiesen werden kann.

durch den folgenden Ablauf beweisen. Zunächst wird die Graphoperation vertex_3 ausgeführt. Dadurch geht der Zustand s_R unter anderem in den Zustand s'_R, der in Abbildung 7.5 dargestellt ist, über.

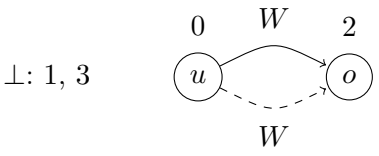
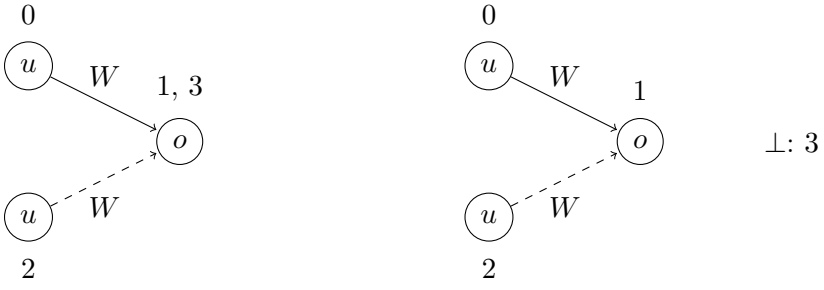


Abbildung 7.5: Nachfolgezustand s'_R für den Zustand s_R bei Anwendung von vertex_3

Der Zustand s_L geht für die Graphoperation vertex_3 beispielsweise in die beiden Nachfolgezustände s'_{L_1} und s'_{L_2} über, die in Abbildung 7.6 dargestellt sind.



(a) Nachfolgezustand s'_{L_1} (b) Nachfolgezustand s'_{L_2}

Abbildung 7.6: Nachfolgezustände für s_L bei Anwendung von vertex_3

Je nach dem in welchen Nachfolgezustand der Zustand s_L übergegangen ist, kann der Nachfolgezustand s'_R unterschiedlich reagieren. Es sei angenommen, dass der Nachfolgezustand von s_L der Zustand s'_{L_1} ist. Dann kann der Zustand s'_R darauf reagieren, indem er einen res-Übergang ausführt. Der Zustand s'_{L_1} kann darauf nicht reagieren, weil außer dem letzten Interfaceknoten noch ein weiterer Interfaceknoten auf denselben Subgraphknoten abgebildet wird (siehe auch Abschnitt 5.3).

Es sei nun angenommen, dass der Zustand s'_{L_2} der Nachfolgezustand von s_L ist. In diesem Fall kann der Zustand s'_R die beiden Knoten 1 und 3 fusionieren, indem er zuerst die Knoten 0 und 3 permutiert (nach Satz 5.7 ist dies mit den vorgestellten Graphoperationen möglich) und anschließend die beiden ersten Knoten fusioniert. Darauf kann der Zustand s'_{L_2} nicht reagieren. Zwar ist die Permutation der Knoten 1 und 3 möglich, allerdings kann ein Knoten des Subgraphen nicht mit einem undefinierten Knoten (außerhalb des gesuchten Subgraphen) fusioniert werden.

Die restlichen Nachfolgezustände, in die s_L durch die Graphoperation vertex übergehen kann, können ebenfalls nicht auf die Fusion der Knoten 1 und 3 reagieren. Folglich lässt sich der Zustand s_R durch keinen erreichbaren Zustand simulieren. Dieses Problem könnte durch den Übergang zu einem deterministischen Automatenfunktor behoben werden, da in diesem Fall die Simulations- und die Sprachäquivalenz identisch wären. Allerdings führt die Berechnung des äquivalen-

ten, deterministischen Automatenfunktors, wie in Abschnitt 5.2 bereits angedeutet, zu einer exponentiellen Explosion des benötigten Speichers.

Insgesamt zeigt dieses Anwendungsbeispiel, dass der in dieser Arbeit vorgestellte Algorithmus zur Überprüfung von invarianten Graphsprachen praktisch genutzt werden kann. Allerdings muss auch darauf hingewiesen werden, dass durch die (benötigte) Unterapproximation und dem damit verbundenen einseitigen Fehler, nicht das gesamte Beispiel korrekt verifiziert werden konnte. Zur Lösung dieses Problems sind weitere Untersuchungen notwendig, die den Rahmen dieser Arbeit sprengen würden und daher unterbleiben müssen.

Kapitel 8

Fazit und Ausblick

Abschließend sollen die Ergebnisse dieser Diplomarbeit nochmals zusammengefasst werden. Des Weiteren sollen ähnliche Arbeiten, die sich mit der Verifikation mittels Graphsprachen befassen, kurz vorgestellt und mit den Ergebnissen dieser Arbeit verglichen werden. Zum Schluss sollen Anknüpfungspunkte für zukünftige Arbeiten aufgezeigt werden.

8.1 Zusammenfassung

Ziel dieser Arbeit war es, zu überprüfen, ob Eigenschaften, die Invarianten dynamischer und verteilter Systeme bilden, als erkennbare Graphsprachen beschrieben werden können und ob diese Graphsprachen für die Verifikation der betrachteten Systeme genutzt werden können.

Um diese Fragestellung zu beantworten, musste zunächst gezeigt werden, dass erkennbare Graphsprachen bezüglich einer monotonen Wohlquasiordnung nach oben abgeschlossen sind. Aufbauend auf diesem Ergebnis konnte ein Algorithmus zur Überprüfung von Invarianten bei Graphsprachen entwickelt werden.

Aufgrund von Überlegungen der praktischen Umsetzbarkeit mussten an dieser Stelle jedoch Kompromisse bei der Entwicklung des Algorithmus' eingegangen werden. Aus diesem Grund musste auf eine Unterapproximation zurückgegriffen werden, weshalb es auch passieren kann, dass eine Verifikation fehlschlägt, obwohl kein Verstoß gegen die zu untersuchende Invariante vorliegt. Da es jedoch nicht möglich ist, dass der Algorithmus ein positives Ergebnis liefert, obwohl die untersuchte Invariante verletzt ist, kann die Verifikation dennoch als korrekt betrachtet werden. Daher wird diese Unterapproximation auch *einseitige Unterapproximation* genannt.

Parallel zu den theoretischen Untersuchungen wurde ein Programm entwickelt, welches einen Automatenfunktor, der die Sprache aller Graphen erkennt, die einen bestimmten Subgraphen enthalten, und den beschriebenen Algorithmus für Graphsprachen implementiert. Mit Hilfe dieses Programms konnte anhand eines Anwendungsbeispiel gezeigt werden, dass der vorgestellte Ansatz grundsätzlich für den praktischen Einsatz geeignet ist. Allerdings ist zu beachten, dass das entwickelte Programm nur für kleine Beispiele effizient einsetzbar ist, da sich die Größe des Zustandsraumes bei den berechnete Automatenfunktoren auf Grund des enormen Speicherbedarfs als problematisch erwiesen hat.

Zu den denkbaren Lösungsmöglichkeiten zählen die Benutzung einer besseren Unterapproximation, um den einseitigen Fehler einzudämmen, und die Suche nach einer besseren Darstellung des Zustandsraumes.

Dennoch konnte das Ziel dieser Arbeit, die Spezifikation von Invarianten mittels Graphsprachen, erfolgreich erreicht werden.

8.2 Vergleichbare Arbeiten

Damit die Ergebnisse dieser Arbeit besser eingeordnet werden können, soll auf vergleichbare Arbeiten aus dem Bereich der Verifikation mittels Graphsprachen eingegangen werden.

In der Arbeit von Koch et. al [16] ist ein anderer Ansatz zum Nachweis der Sicherheit bestimmter Zugriffsregeln beschrieben. Dieser Ansatz beruht ebenfalls auf der Idee, sicherheitskritische Strukturen als Graphen zu beschreiben und zu überprüfen, ob mit Hilfe der Zugriffsregeln ein Zustand erreichbar ist, der gegen die Sicherheitsanforderungen verstößt. Allerdings muss im Gegensatz zu dem in dieser Arbeit vorgestellten Ansatz das gesamte System modelliert werden. Außerdem ist in Kochs Ansatz für die Überprüfung der Sicherheit notwendig, alle Zustände zu betrachten, die nach einer (endlichen) Anzahl von Schritten durch Anwendung der Zugriffsregeln erreicht werden können. Das bedeutet, dass statt der Überprüfung von Invarianten wie in dieser Arbeit eine Erreichbarkeitsanalyse durchgeführt wird. Außerdem führt die Notwendigkeit, das gesamte System modellieren zu müssen, im Allgemeinen dazu, dass sehr große Zustände und auch Zustandsräume betrachtet werden müssen.

In den Arbeiten von Heckel et al. [13] und Habel et al. [12] wird die Korrektheit von Programmen bezüglich einer formalen Spezifikation bzw. einer Anwendungsbedingung, zum Beispiel einer Sicherheitsbedingung, untersucht. Das zu untersuchende System wird als Graphtransformationssystem dargestellt, wobei die Zustände des Systems den Graphen entsprechen und die Systemübergange als Transformationsregeln modelliert werden. Mit Hilfe der formalen Spezifikation, die zu erfüllen ist, können Vorbedingungen für die Anwendung der Transformationsregeln bestimmt werden. Durch die Überprüfung, ob diese Vorbedingungen erfüllt werden, kann die Sicherheit des Systems gezeigt werden. Ferner kann dieser Ansatz zur Überprüfung von Invarianten genutzt werden, da mit Hilfe der Anwendungsbedingung ebenfalls Mengen von Graphen spezifiziert werden können. Allerdings ist dabei zu beachten, dass auf diese Art nur eine Teilklasse der erkennbaren Sprachen beschrieben werden kann. Nach Rensink [22] stimmt die Klasse aller Graphen, die durch Anwendungsbedingungen spezifiziert werden können, mit der Klasse aller Graphen, die durch Prädikatenlogik erster Stufe beschrieben werden können, überein. Nach Courcelle [7] ist aber jeder Graph, der mit Hilfe der monadischen Logik zweiter Stufe definiert werden kann, ein erkennbarer Graph. Aufgrund der geringeren Ausdrucksmächtigkeit erscheint der in dieser Arbeit vorgestellte Ansatz als allgemeiner einsetzbar. Für eine abschließende Bewertung ist es allerdings erforderlich, die Ansätze von von Heckel et al. bzw. Habel et al. genauer zu analysieren und mit dem in dieser Arbeit vorgestellten Ansatz zu vergleichen.

Neben der Verifikation von Zugriffsregeln besteht ein weiteres Einsatzgebiet für die Überprüfung von Invarianten in der Untersuchung der Korrektheit von Zeigerstrukturen. In den Arbeiten von Fradet und Métayer [10] sowie Bakewell et al. [3] werden Techniken vorgestellt, bei denen die Zeigerstruktur eines Programms durch entsprechende Graphen und die zulässigen Zeigermanipulation durch Transformationsregeln dargestellt werden. Die modellierten Zeigermanipulationen sind genau dann sicher, wenn die Sprache aller Graphen, die gültige Zeigerstrukturen beschreiben, eine Invariante bezüglich der durch die Transformationsregeln beschriebenen Zeigermanipulationen ist. Im Gegensatz zu dem in dieser Arbeit

vorgestellten Ansatz werden für die Überprüfung und Beschreibung von Invarianten Graphgrammatiken (kontextfreier Graphsprachen oder Verallgemeinerungen kontextfreier Graphsprachen) genutzt. Allerdings kann dieser Ansatz nur bedingt mit dem Ansatz dieser Arbeit verglichen werden, da die Klasse der kontextfreien Graphsprachen und die Klasse der erkennbaren Graphsprachen (im Gegensatz zu Wortsprachen) nicht miteinander vergleichbar sind. Dies folgt umgehend aus der folgenden Überlegung: Die Sprache aller Graphen, die ebenso viele mit A wie mit B beschriftete Hyperkanten enthalten, ist zwar kontextfrei, aber nicht erkennbar. Daher kann die Klasse der kontextfreien Sprachen keine Teilklasse der Klasse der erkennbaren Sprachen sein. Umgekehrt ist die Sprache aller Graphen aber eine erkennbare Sprache, die nicht kontextfrei ist. Somit kann die Klasse der erkennbaren Sprachen auch keine Teilklasse der Klasse der kontextfreien Sprachen sein. Ein ausführlicher Beweis findet sich in [7].

8.3 Ausblick

Eine Schwierigkeit vorgestellten Ansatzes besteht darin, dass in Folge der Unterapproximation durch die Simulationsrelation ein einseitiger Fehler auftreten kann. Dass dieses Problem nicht nur von theoretischer, sondern auch von praktischer Bedeutung ist, hat das in Kapitel 7 vorgestellte Beispiel gezeigt. Daher wäre ein Ansatz zur Verbesserung des vorgestellten Verfahrens, die Suche einer besseren Unterapproximation, die gröber als die Simulationsrelation (aber dennoch feiner als die Myhill-Graphquasiordnung) ist. Des Weiteren wäre auch eine Implementierung eines Algorithmus' zum Auffinden von Abläufen, die zwei nicht in Relation stehende Zustände trennt, für die Überprüfung der negativen Verifikationsergebnisse hilfreich. Dies wären mögliche Ansätze zur Verbesserung des Verfahrens bzw. des entwickelten Programms.

Für die praktische Anwendbarkeit ist auch die Zustandsraumexplosion des berechneten Automatenfunktors entscheidend. Dabei hängt die Größe des Zustandsraums neben den gesuchten Subgraphen vor allem von der maximalen Interfacegröße ab, welche die eingegeben Cospans beschränkt. Selbst für den Fall, dass die zu untersuchenden Eigenschaften mit kleinen Graphen beschrieben werden können, erreicht der Zustandsraum des Automatenfunktors eine Größe, die einen praktischen Einsatz unmöglich macht.

Bereits für das in Kapitel 7 vorgestellte Beispiel findet selbst bei geringer Interfacegröße eine exponentielle Explosion des Zustandsraumes statt. Die Größe des Zustandsraumes für die Automatenfunktoren, welche die Subgraphen „Doppelter Zugriff“ und „Zwei Benutzer“ jeweils einzeln erkennen bzw. für den konstruierten Vereinigungsautomatenfunktor können aus der Tabelle 8.1 abgelesen werden.

	Maximale Interfacegröße							
	0	1	2	3	4	5	6	7
Doppelter Zugriff	7	24	69	194	551	1.588	4.633	13.638
Zwei Benutzer	13	51	173	589	2.067	7.475	27.697	104.553
Vereinigung	20	75	242	783	2.618	9.063	32.330	118.191

Tabelle 8.1: Zustandsraumgröße in Abhängigkeit der maximalen Interfacegöße

Betrachtet man außerdem die Größe der berechneten Simulationsrelation, wird das Problem noch deutlicher. In Tabelle 8.2 ist die Größe der Simulationsrelation, das heißt die Anzahl der Zustandspaare der Relation, in Abhängigkeit der maximalen Interfacegröße, dargestellt.

Maximale Interfacegröße							
0	1	2	3	4	5	6	7
400	3.425	31.314	323.995	≈3,7 Mio.	≈45 Mio.	≈587 Mio.	≈8 Mrd.

Tabelle 8.2: Größe der Simulationsrelation in Abhängigkeit der max. Interfacegöße

Zum Vergleich sind in Tabelle 8.3 die Laufzeiten des Programms dargestellt. Die ersten beiden Zeilen geben die Laufzeiten für die Berechnung der beiden Automatenfunktoren „Doppelter Zugriff“ und „Zwei Benutzer“ an. In der dritten Spalte wird die Laufzeit, die zur Vereinigung der beiden, zuvor konstruierten Automatenfunktoren benötigt wird, angegeben. Die letzte Spalte gibt die benötigte Laufzeit zur Ausführung des Algorithmus' zur Prüfung von Invarianten und zur Berechnung der Simulationsrelationstabelle an. Auf Grund der oben beschriebenen Zustandsraumexplosion kann die Relationstabelle nur für eine Interfacegröße kleiner oder gleich vier berechnet werden. Daher kann für größere Interfaces keine Laufzeit für die Berechnung des Algorithmus' angegeben werden. Dies zeigt, dass der Flaschenhals bei der Berechnung vorrangig der mangelnde Speicher ist.

	Maximale Interfacegröße							
	0	1	2	3	4	5	6	7
Doppelter Zugriff	<1s	<1s	<1s	<1s	1s	1s	6s	1 Min
Zwei Benutzer	<1s	<1s	<1s	<1s	1s	17s	4 Min	1 Std
Vereinigung	<1s	<1s	<1s	<1s	<1s	2s	2s	6s
Algorithmus	<1s	<1s	<1s	2s	26s	–	–	–

Tabelle 8.3: Laufzeit in Abhängigkeit der maximalen Interfacegöße

Für die Ermittlung der angegeben Laufzeiten wurde auf die folgende Hard- und Software zurückgegriffen:

CPU:	Intel Xeon Dualcore-Prozessor 5150, 2,66 GHz
Verfügbarer Speicher:	2 GB, 667 MHz
Front-Side-Bus:	1333 MHz
Betriebssystem:	GNU/Linux, 2.6.23
Java:	Java SE 1.6.0_03-b05, 32-Bit

Ein möglicher Anknüpfungspunkt für weitere Arbeiten wäre daher die Darstellung des Zustandsraumes mit Hilfe von *Binary Decision Diagrams (BDDs)*. Ein Beispiel für den Einsatz von BDDs zur Darstellung von Graphen kann in [5] gefunden werden. Einerseits könnte es durch die effizientere Darstellung des Zustandsraumes ermöglicht werden, dass auch Automatenfunktoren für eine größere, maximale Interfacegröße praktisch eingesetzt werden können. Andererseits könnte die BDD-Darstellung des Zustandsraumes auch eine Determinisierung und

Minimalisierung des berechneten Automatenfunktors ermöglichen. Dies müssten weitergehende Arbeiten zeigen.

Anhang A

Inhalt der CD-ROM

A.1 PDF-Dateien

Pfad: /

Diplomarbeit.pdf	PDF-Version dieser Arbeit

A.2 Programm-Dateien

Pfad: program/

doc/	Dieser Ordner enthält die JavaDoc-Dokumentation
operations/	Dieser Ordner enthält Graphoperations-Dateien
src/	Dieser Ordner enthält die Quelldateien dieses Programms
subgraphs/	Dieser Ordner enthält Subgraph-Dateien
tmp/	Dieser Ordner enthält temporäre Dateien
config.dtd	DTD-Datei für die Konfigurationsdatei
config.xml	Konfigurationsdatei
hypergraph.dtd	DTD-Datei für die Subgraphdateien
jdom.jar	JDOM-Bibliothek
README	Anleitung zum Starten des Programms
simulator.jar	Simulations-Programm

Literaturverzeichnis

[1] Adámek, Jiří, Horst Herrlich und George Strecker: *Abstract and concrete categories*. Wiley-Interscience, New York, NY, USA, 1990.

[2] Artin, Martin: *Algebra*. Birkhäuser Verlag, 1998.

[3] Bakewell, Adam, Detlef Plump und Colin Runciman: *Checking the Shape Safety of Pointer Manipulations*. In: Berghammer, Rudolf, Bernhard Möller und Georg Struth (Herausgeber): *RelMiCS*, Band 3051 der Reihe *Lecture Notes in Computer Science*, Seiten 48–61. Springer, 2003.

[4] Bauderon, Michel und Bruno Courcelle: *Graph Expressions and Graph Rewritings*. Mathematical Systems Theory, 20(2-3):83–127, 1987.

[5] Becker, Basil, Dirk Beyer, Holger Giese, Florian Klein und Daniela Schilling: *Symbolic invariant verification for systems with dynamic structural adaptation*. In: Osterweil, Leon J., H. Dieter Rombach und Mary Lou Soffa (Herausgeber): *ICSE*, Seiten 72–81. ACM, 2006.

[6] Bruggink, H.J. Sander und Barbara König: *On the Recognizability of Arrow and Graph Languages*. In: *Proc. of ICGT '08 (International Conference on Graph Transformation)*. Springer, 2008. LNCS, to appear.

[7] Courcelle, Bruno: *The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs*. Inf. Comput., 85(1):12–75, 1990.

[8] Ehrenfeucht, Andrzej, David Haussler und Grzegorz Rozenberg: *On Regularity of Context-Free Languages*. Theor. Comput. Sci., 27:311–332, 1983.

[9] Ehrig, Hartmut, Karsten Ehrig, Ulrike Prange und Gabriele Taentzer: *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] Fradet, Pascal und Daniel Le Métayer: *Shape Types*. In: *POPL*, Seiten 27–39, 1997.

[11] Griffing, Gary: *Composition-representative subsets*. Theory and Applications of Categories, 11(19):420–437, 2003.

[12] Habel, Annegret, Karl-Heinz Pennemann und Arend Rensink: *Weakest Preconditions for High-Level Programs*. In: Corradini, Andrea, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro und Grzegorz Rozenberg (Herausgeber): *ICGT*, Band 4178 der Reihe *Lecture Notes in Computer Science*, Seiten 445–460. Springer, 2006.

[13] Heckel, Reiko und Annika Wagner: *Ensuring consistency of conditional graph grammars – a constructive approach.* In: *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, Seite 2, 1995.

[14] Higman, Graham: *Ordering by Divisibility in Abstract Algebras.* Proc. London Math. Soc., s3-2(1):326–336, 1952.

[15] Hunter, Jason und Brett McLaughlin: *JDOM API Javadoc.* www.jdom.org/docs/apidocs, 2007, 18. November. Zugriff am 20. September 2008.

[16] Koch, Manuel, Luigi V. Mancini und Francesco Parisi-Presicce: *Decidability of Safety in Graph-Based Models for Access Control.* In: Gollmann, Dieter, Günter Karjoth und Michael Waidner (Herausgeber): *ESORICS*, Band 2502 der Reihe *Lecture Notes in Computer Science*, Seiten 229–243. Springer, 2002.

[17] Lawvere, William F. und Robert Rosebrugh: *Sets for Mathematics.* Cambridge University Press, January 2003.

[18] Luca, Aldo de und Stefano Varricchio: *Well Quasi-Orders and Regular Languages.* Acta Inf., 31(6):539–557, 1994.

[19] Myhill, John: *Finite automata and the representation of events.* Technischer Bericht WADD TR-57-624, Wright Patterson AFB, Ohio, 1957.

[20] Nerode, Anil: *Linear Automaton Transformations.* Proceedings of the American Mathematical Society, 9(4):541–544, 1958.

[21] Pierce, Benjamin C.: *Basic category theory for computer scientists.* MIT Press, Cambridge, MA, USA, 1991.

[22] Rensink, Arend: *Representing First-Order Logic Using Graphs.* In: Ehrig, Hartmut, Gregor Engels, Francesco Parisi-Presicce und Grzegorz Rozenberg (Herausgeber): *ICGT*, Band 3256 der Reihe *Lecture Notes in Computer Science*, Seiten 319–335. Springer, 2004.

[23] Rozenberg, Grzegorz (Herausgeber): *Handbook of Graph Grammars and Computing by Graph Transformations*, Band Volume 1: Foundations. World Scientific, 1997.

[24] Zimmermann, Karl-Heinz: *Diskrete Mathematik.* Books on Demand GmbH, 2006.

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Universität Duisburg-Essen, am 9. November 2008

Christoph Blume