UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Masterarbeit

Automatischer Nachweis von Bisimulationsäquivalenzen bei Graphtransformationssystemen

Dennis Nolte
Matrikelnummer: 2234055

Abteilung Informatik und angewandte Kognitionswissenschaft
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

4. November 2012

Prüfer:
Prof. Dr. Barbara König
Prof. Dr. Xinlong Zhou

Betreuer:
Mathias Hülsbusch

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Diese Arbeit beruht auf Grundlage der Forschungsergebnisse zur Gewinnung eines beschrifteten Transitionssystems in Graphtransformationssystemen. Unter Verwendung des DPO Ansatzes mit Borrowed Context und den in [EK04] erbrachten Kongruenzresultaten, werden in dieser Ausarbeitung Algorithmen definiert, die zum Automatischen Nachweis von Bisimulationsäquivalenzen bei Graphtransformationsystemen genutzt werden können. Diese Ausarbeitung wurde im Rahmen des DFG-Projekts *Behaviour Simulation and Equivalence of Systems Modelled by Graph Transformation* (kurz *Behaviour-GT*) geschrieben. Das Projekt ist eine Kooperation zwischen dem Lehrstuhl *Theoretische Informatik* von Frau Prof. Barbara König an der Universität Duisburg-Essen, sowie Herr Prof. Hartmut Ehrig an der Technischen Universität Berlin.

1.1. Motivation

In der heutigen Zeit ist die Modellierung von komplexen und nebenläufigen Prozessen die Grundlage für die Erforschung der darauf aufbauenden Systeme. In den letzten Jahren gab es zunehmende Erfolge in der Modellierung solcher Prozesse mit Prozesskalkülen aber auch mit Graphtransformationssystemen. Graphtransformationssysteme erlauben es dynamische und nebenläufige Systeme auf eine natürliche und intuitive Art zu modellieren. Einer der Standardansätze zur Modellierung ist der DPO Ansatz welcher es erlaubt, durch Graphtransformationsregeln, die das Systemverhalten modellieren, ein unbeschriftetes Transitionssystem zu erzeugen. Durch die Erweiterung dieses Ansatzes mit dem Borrowed Context gelingt es sogar beschriftete Transitionssysteme zu generieren. Mithilfe von Bisimulationsäquivalenzen die sich für die beschrifteten Transitionssysteme berechnen ließen, könnte man einzelne Komponenten des Systems austauschen ohne das beobachtete Verhalten des Gesamtsystems zu verändern. Es ist daher erstrebenswert ein Tool zur Verfügung zu stellen das prüft, ob eine modellierte Komponente (durch einen Graphen mit zugehörigem Interface) durch eine andere ohne Bedenken ersetzt werden kann.

1.2. Aufgabenstellung

Das Thema dieser Arbeit lautet 'Automatischer Nachweis von Bisimulationsäquivalenzen bei Graphtransformationssystemen'. Es werden im Folgenden Bisimulationsäquivalenzen aus Graphtransformationsregeln abgeleitet und untersucht. Am Ende soll eine Methode zur Verfügung gestellt werden mit denen die Verhaltensgleichheit von zwei gegebenen Graphen in Bezug auf ihre Interaktion mit der Umgebung festgestellt werden kann. Dabei interagieren die Graphen über ein festgelegtes Interface mit der Außenwelt. Dabei sollen folgende Unteraufgaben bearbeitet werden:

- Ableitung von Labels, die die möglichen Interaktionen eines Graphen mit seiner Umgebung beschreiben
- Überprüfung, ob eine gegebene Relation eine Bisimulation ist (mit Einsatz von Modulo-Techniken, vor allem Up-To-Context-Techniken)
- Ableitung einer Bisimulation, die ein gegebenes Paar von Graphen enthält (On-the-Fly-Technik, siehe [Hir99])

Die entsprechenden Konstruktionen sollen in Mathematica realisiert werden. Teilweise bereits vorhandene Pushout- und Pullback-Konstruktionen müssen gegebenenfalls überarbeitet werden.

1.3. Struktur dieser Arbeit

Diese Ausarbeitung ist folgendermaßen strukturiert: In Kapitel 2 wird eine Einführung in die grundlegenden Modelle gegeben auf denen diese Arbeit aufbaut. Kapitel 3 dient der Definition einer Bisimulationsrelation sowie der Beschreibung bekannter Algorithmen um diese zu berechnen. In Kapitel 4 wird das Kernthema sowie mögliche Optimierungsansätze erläutert, gefolgt von den Algorithmen die das Grundkonzept für die Implementierung beschreiben (Kapitel 5). Eine Einführung in Mathematica gibt es in Kapitel 6 und die letztendliche Implementierung wird in Kapitel 7 überschaubar zusammengefasst. Im Anschluss wurde eine Evaluation der Ergebnisse durchgeführt deren Ergebnisse in Kapitel 8 analysiert werden. Zum Abschluss folgt in Kapitel 9 eine Zusammenfassung der Ergebnisse sowie ein Ausblick auf mögliche Erweiterungen für die Zukunft.

2. Grundlegende Definitionen

Vorraussetzung für das Verständnis der folgenden Kapitel, ist die Kenntnis über grundlegende Definitionen aus der Graphentheorie, Morphismen und Graphen im allgemeinen. Desweiteren wird die Kenntnis über adhäsive Kategorien vorrausgesetzt. Eine Zusammenfassung der für die algebraische Graphenersetzung benötigten Kategorien findet sich in [Hac03]. Im übernächsten Kapitel dieser Arbeit wird die Borrowed-Context-Nutzung im DPO Ansatz beschrieben. Der DPO Ansatz ist Teil der algebraischen Graphenersetzung und wurde erstmals durch Ehrig in *Graph grammars: An algebraic approach* im Jahr 1973 beschrieben [CMR$^+$97]. Die algebraischen Graphenersetzung basiert auf Pushouts die mit Klebevorgängen die Graphenersetzung modellieren.

2.1. Transitionssysteme

Transitionssysteme bilden die Basis für den Bisimulationscheck des nächsten Kapitels. Die hier verwendeten Transitionssysteme (LTS) sind an ihren Kanten beschriftet und modellieren daher atomare Aktionen die nach aussen hin sichtbar sind. Die Definition der Transitionssysteme beruht auf der Definition in [BK08].

Definition 2.1.1 (Unbeschriftete Transitionssysteme) *Ein Transitionssystem* $T = (S, Act, \longrightarrow, I)$ *besteht aus den folgenen Komponenten:*

- S *ist der Zustandsraum*
- Act *ist eine Menge atomarer Aktionen*
- $\longrightarrow \subseteq S \times S$ *ist die Menge der Transitionsrelationen*
- $I \subseteq S$ *ist die Menge der Initialzustände*

Das Transitionssystem ist endlich *sofern S und Act endlich sind.*

Für den Rest der Ausarbeitung benötigen wir die Definition für beschriftete Transitionssysteme.

Definition 2.1.2 (Beschriftete Transitionsysteme) *Ein beschriftetes Transitionssystem* $LTS = (S, Act, \longrightarrow, I)$ *besteht aus den bereits definierten Elementen eines unbeschrifteten Transitionssystems. Die Menge der Transitionsrelation wird allerdings durch* $\longrightarrow \subseteq S \times Act \times S$ *definiert.*

Anstatt beschrifteten Transitionssystem wird im folgenden nur LTS (Labelled Transitionsystem) geschrieben. Außerdem wird für Transitionsrelationen die Schreibweise $s \xrightarrow{\alpha} s'$ anstatt $(s, \alpha, s') \in \longrightarrow$ verwendet.

Beispiel 2.1.1 (Transitionssystem) *Abbildung 2.1 zeigt* $LTS = (S, Act, \longrightarrow, I)$ *mit*

- $S = \{q_0, q_1, q_2, q_3\}$
- $Act = \{but, liq, choc\}$
- $\longrightarrow = \{q_0 \xrightarrow{but} q_1, q_1 \xrightarrow{liq} q_2, q_1 \xrightarrow{choc} q_3\}$
- $I = \{q_0\}$

Modelliert wird ein Getränkeautomat bei dem der Benutzer nach dem Drücken des Knopfes (but), eine Entscheidung zwischen Wasser (liq) oder Kakao (choc) treffen kann.

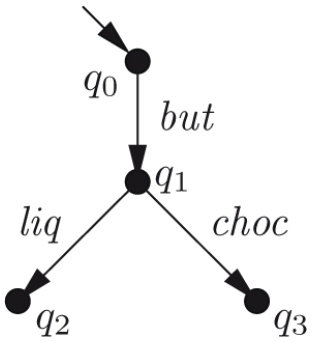

Abbildung 2.1.: LTS [Tre08]

2.2. Graphen

Die in dieser Arbeit verwendeten Graphen sind Bestandteil einer Graphstruktur die nach [EK04] definiert ist. Sie ist möglichst allgemein gehalten und umfasst verschieden Graphenarten wie gerichtete Graphen und Hypergraphen. Die dargestellten Abbildungen befinden sich in [EEPT06].

Definition 2.2.1 (Graphstrukturen) *Eine Graphstruktur* GS *ist ein Tupel aus den betrachteten Sorten* S*, den unären Operationssymbolen* $(OP_{s,s'})_{s,s' \in S}$ *und den Abbildungsalphabeten* $(\Sigma_s)_{s \in S}$*. Kurz* $GS = (S, OP, \Sigma)$ *geschrieben.*
Eine Graphstruktur A *aus* GS *ist eine Sorten indexierte Menge* $(A_s)_{s \in S}$ *mit einer Menge von Sorten indexierten Abbildungsfunktionen* $(l_s^A)_{s \in S}$ *sodass* $l_s^A : A_s \to \Sigma_s$ *gilt. Zusätzlich besitzt die A die Menge an OP indexierten Übergange* $(op^A)_{op \in OP}$ *sodass* $op^A : A_s \to A_{s'}$ *gilt sofern* $op \in OP_{s,s'}$*.*
Ein Morphismus $\varphi : A \to B$ *auf dieser Graphstruktur ist eine Sorten indexierte Menge von Übergängen der Form* $\varphi = (\varphi_s : A_s \to B_s)_{s \in S}$ *wobei für alle* $x \in A_s$ *sowohl* $l_s^A(x) = l_s^B(\varphi(x))$ *als auch* $op^B(\varphi(x)) = \varphi(op^A(x))$ *gilt. Der Morphismus* φ *ist injektiv sofern alle Übergänge injektiv sind und er ist ein Isomorphismus sofern alle Übergänge bijektiv sind. Ein Isomorphismus der Form* $\varphi : A \to A$ *ist ein Automorphismus. Auf alle Kategorien von Graphen und Graphmorphismen dieser Struktur sind Pushout und Pullback anwendbar, wie sie in der Kategorie* ***Set*** *verwendet werden [EK04].*

2.2.1. Graphen

Die verwendete Graphstruktur beinhaltet die Sorten $S = \{v, e\}$. Wir verwenden zwei Sorten indexierte Mengen $A_v = V$ (Vertices) für Knoten und $A_e = E$ (Edges) für Kanten, sowie zwei Operator Symbole $s, t \in OP_{E,V}$. Das Abbildungsalphabet $\Sigma = \Sigma_V \bigcup \Sigma_E$ ist in Knotenlabel Σ_V und Kantenlabel Σ_E unterteilt. Die in dieser Ausarbeitung verwendeten Graphen sind Multigraphen.

Definition 2.2.2 (Gerichteter Multigraph) *Ein Graph $G = (V, E, s, t)$ ist ein Tupel bestehend aus der Knotenmenge V (*Vertices*) und der Kantenmenge E (*Edges*). Zusätzlich existieren Funktionen $s, t : E \to V$ wobei s die Quellfunktion (*source*) und t die Zielfunktion (*target*) definieren wie in Abbildung 2.2 dargestellt.*

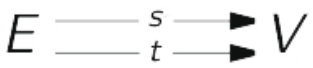

Abbildung 2.2.: Allgemeine Graphendarstellung

Beispiel 2.2.1 (Graph G_s) *Abbildung 2.3 zeigt den Graphen $G_s = (V_S, E_S, s_S, t_S)$ mit $V_S = \{u, v, x, y\}$, $E_S = \{a, b\}$, $s_S(a) = u$ $s_S(b) = u$ und $t_S(a) = v$ $t_S(b) = v$. Die Label der Knoten und Kanten sind in unserem Beispiel eindeutig und Identifizieren daher die Elemente eindeutig. Zusätzlich könnte man Labelfunktionen $l_{V_S} : V_S \to L$ und $l_{E_S} : E_S \to L$ definieren um gleiche Label aus einer Labelmenge L zu ermöglichen.*

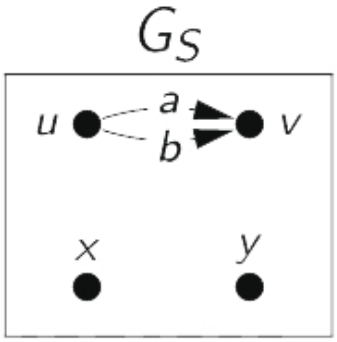

Abbildung 2.3.: Beispielgraph

Definition 2.2.3 (Teilgraph) *Ein Graph $N = (V_N, E_N, s_N, t_N)$ ist ein Teilgraph oder Subgraph des Graphen $G = (V_G, E_G, s_G, t_G)$ falls*

- *$V_N \subseteq V_G$ und*
- *$E_N \subseteq E_G$ gilt, sodass*
- *$\forall e \in E_N : s_N(e) \in V_N$ und $t_N(e) \in V_N$.*

Wir schreiben $N \subseteq G$ wenn N ein Teilgraph von G ist. Der Graph G wird als Supergraph oder Obergraph von N bezeichnet.

2.2.2. Morphismen

Graphmorphismen sind strukturerhaltende Abbildungen von Graphen in andere Graphen. Wir verwenden in dieser Ausarbeitung Monomorphismen und im Folgenden definieren wir die wichtigsten Begriffe.

Definition 2.2.4 (Graphmorphismus) *Ein Graphmorphismus $f : G_1 \to G_2, f = (f_V, f_E)$ besteht aus zwei Funktionen. Die Funktion $f_V : V_1 \to V_2$ zum abbilden der Knoten und der Funktion $f_E : E_1 \to E_2$ zum abbilden der Kanten. Die Allgemeindarstellung ist in Abbildung 2.4 zu sehen.*

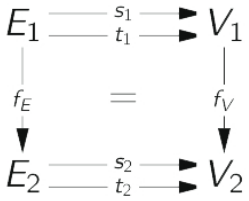

Abbildung 2.4.: Allgemeine Morphismusdarstellung

Die Komposition von zwei Graphmorphismen $f : G_1 \to G_2, f = (f_V, f_E)$ und $g : G_2 \to G_3, g = (g_V, g_E)$ mit $g \circ f : G_1 \to G_3, g \circ f = (g_V \circ f_V, g_E \circ f_E)$ ist wieder ein Graphmorphismus.

Definition 2.2.5 (Monomorphismus) *Ein Graphmorphismus $f : G_1 \to G_2, f = (f_V, f_E)$ ist genau dann ein Monomorphismus wenn es Morphismen $g, h : G_X \to G_1, g = (g_V, g_E), h = (h_V, h_E)$ gibt, so dass $f \circ g = f \circ h$ gilt. Monomorphismen sind in den Kategorien von algebraischen Strukturen genau die injektiven Morphismen.*

Beispiel 2.2.2 (Morphismus f) *Der in Abbildung 2.5 abgebildete Morphismus $f : G_S \to G_T$ besitzt die Eigenschaften $f_V(u) = s \quad f_V(x) = s \quad f_V(v) = t \quad f_V(y) = t \quad f_E(a) = e$ und $f_E(b) = e$.*

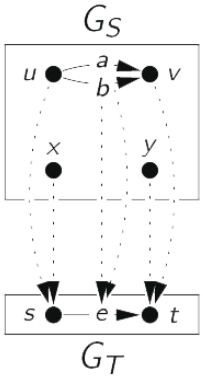

Abbildung 2.5.: Beispielmorphismus

Definition 2.2.6 (Identitäts-Graphmorphismus) *Für jeden Graph G existiert ein Identitäts-Graphmorphismus $id_G \in f : G \to G$ so dass $id_G \circ f = f$ und $f \circ id_G = f$ gilt.*

Der Identitäts-Graphmorphismus stellt somit das neutrale Element der Verkettung dar. Es gilt für einen Identitäts-Graphmorphismus $f : G \to G, f = (f_V, f_E)$, dass $f(V) = V$ und $f(E) = E$ ist. Im Zuge der Matchingtheorie benötigen wir noch die Definition von Graphisomorphismen.

Definition 2.2.7 (Graphisomorphismus) *Ein Morphismus $f : G_1 \to G_2$ heißt Graphisomorphismus, wenn ein Morphismus $g : G_2 \to G_1$ existiert, so dass $f \circ g = id_{G_2}$ und $g \circ f = id_{G_1}$ gilt.*

Ein Morphismus von G nach G heißt Endomorphismus von G und ein Endomorphismus, der gleichzeitig ein Isomorphismus ist, heißt Automorphismus.

2.2.3. Matching

Für die Nutzung des Borrowed Context werden Grundkenntnisse über Graphmatching benötigt. Die in dieser Ausarbeitung wichtigsten Matchings sind vollständige Matchings und partielle Matchings.

Definition 2.2.8 (Vollständiges Matching) *Gegeben seien zwei Graphen G und H. Die Graphen G und H besitzen ein vollständiges Match $f = (f_V, f_E)$, falls ein Teilgraph N mit $N \subseteq H$ existiert, sodass $f : G \to N$ ein Isomorphismus ist.*

Die Suche eines vollständigen Matches ist auch unter dem Namen Subgraph-Isomorphie-Problem bekannt. Die Komplexität des Problems ist als NP-Vollständig bekannt (siehe [Abd98]).

Definition 2.2.9 (Partielles Matching) *Gegeben seien zwei Graphen G und H. Sei $M \subseteq G$ und $N \subseteq H$. Ein partielles Matching $f = (f_V, f_E) : M \to N$ von G und H existiert genau dann wenn f ein Isomorphismus ist.*

Die Suche eines partiellen Matches besitzt die Komplexität P da der leere Graph immer Teilgraph eines Graphen ist. Erweitert man das Problem und sucht ein Match das mehr als k Kanten besitzt, ist das Problem NP-Vollständig. Jedes partielle Matching $f : G \to H$ lässt sich zu einem partiellen Matching $G \leftarrow D \to H$ erweitern. Die partiellen Matchings $G \leftarrow D \to H$ werden in dieser Ausarbeitung vorwiegend benötigt.

3. Berechnung der Bisimulationsrelation

Bisimulationsäquivalenzen (Verhaltensäquivalenz) wurden das erste mal in [Par81] definiert und später für Prozess Kalküle wie in [Mil99] für das π-Kalkül verwendet (siehe [HM01]). Viele dieser Kalküle lassen sich in Graphtransformationssysteme übersetzen. Wir sind vor allem an der Definition für Transitionssysteme interressiert.

3.1. Bisimulation und Bisimularität

Beim Vergleich von Transitttionssystemen spielt Sprachäquivalenz eine große Rolle.

Definition 3.1.1 (Sprachäquivalenz) *Sei $LTS = (S, Act, \longrightarrow, I)$ ein Transitionssystem und $\alpha, \beta \in S$. $L(\alpha) = \{\omega \in Act^* | \alpha \xrightarrow{\omega} \beta$ für ein $\beta \in S\}$, und $L(LTS) = L(\alpha_0)$ mit $\alpha_0 \in I$. Die Zustände α und β sind Sprachäquivalent (in Zeichen $\alpha \sim_L \beta$) falls $L(\alpha) = L(\beta)$.*

Allerdings reicht Sprachäquivalenz oftmals nicht aus, weshalb Transitionssysteme vor allem auf Bisimulationsäquivalenzen untersucht werden.

Definition 3.1.2 (Bisimulation) *Sei $LTS = (S, Act, \longrightarrow, I)$ ein Transitionssystem. Eine binäre Relation $\mathcal{R} \subseteq S \times S$ heißt genau dann Bisimulation, wenn für jedes Paar $(\alpha, \beta) \in \mathcal{R}$ und für jede Aktion $a \in Act$ gilt:*

Für jedes α' mit $\alpha \xrightarrow{a} \alpha'$ existiert ein β' mit $\beta \xrightarrow{a} \beta'$ und $(\alpha', \beta') \in \mathcal{R}$

Für jedes β' mit $\beta \xrightarrow{a} \beta'$ existiert ein α' mit $\alpha \xrightarrow{a} \alpha'$ und $(\alpha', \beta') \in \mathcal{R}$

Die graphische Darstellung dieser Aussage ist in Abbildung 3.1 dargestellt.

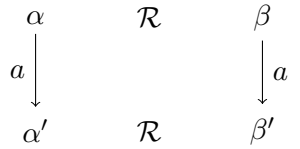

Abbildung 3.1.: Bisimulation

Definition 3.1.3 (Bisimularität) *Zwei Zustände α, β heißen bisimular (in Zeichen $\alpha \sim \beta$), wenn es eine Bisimulation $\mathcal{R}$ gibt mit $(\alpha, \beta) \in \mathcal{R}$. Die Relation $\sim$ heißt dann Bisimularität oder Bisimulationsäquivalenz.*

Satz 3.1.1 (Bisimularität ist eine Bisimulation) *Gegeben sei das Transitionssystem* $LTS = (S, Act, \longrightarrow, I)$. *Für eine Bisimularität* $\sim$ *gelten folgende Eigenschaften:*

$\sim = \bigcup\{\mathcal{R} | \mathcal{R} \subseteq S \times S, \mathcal{R} is teine Bisimulation\}$

$\sim$ *ist die größte Bisimulation und ist eine Äquivalenzrelation*

In [Sti95] wurde ein Spiel mit zwei Spielern beschrieben mit dem Bisimularität in Transitionssystemen überprüft werden kann. Beginnend mit einem Zustandspaar (α, β) halten sich beide Spieler abwechselnd an folgende Regeln:

1. Sollten beide gewählten Zustände keine Transitionen besitzen gewinnt Spieler II. Besitzt nur einer der beiden Zustände keine Möglichkeit einen Zug zu machen wird Spieler I zum Gewinner erklärt. Andernfalls wählt Spieler I einen der beiden Zustände aus und nutzt eine vorhandene Transition um einen Zug zu machen (Entweder $\alpha \xrightarrow{a} \alpha'$ oder $\beta \xrightarrow{a} \beta'$).
2. Spieler II muss nun reagieren und mit dem anderen Zustand dieselbe Aktion a ausführen wie Spieler I (Entweder $\beta \xrightarrow{a} \beta'$ oder $\alpha \xrightarrow{a} \alpha'$). Falls dies nicht möglich ist wird Spieler I zum Gewinner erklärt.
3. Das Spiel beginnt mit dem neuen Zustandspaar (α', β') von vorne. Sollte das Spiel unendlich lange fortgesetzt werden können gewinnt Spieler II.

Aus den Spielregeln können nun folgende Rückschlüsse gezogen werden:

- Falls Spieler II eine Gewinnstrategie für (α, β) besitzt so gilt $\alpha \sim \beta$
- Falls Spieler I eine Gewinnstrategie für (α, β) besitzt so gilt $\alpha \nsim \beta$

Beispiel 3.1.1 (Bisimularität) *Seihen die Transitionssysteme in Abbildung 3.2 gegeben und sei* $\alpha = 1, \beta = 5$. *Es gilt* $\alpha \sim \beta$ *und die Bisimulationsrelation lautet* $\mathcal{R} = \{(1,5), (2,6), (2,7), (3,8), (4,8)\}$.

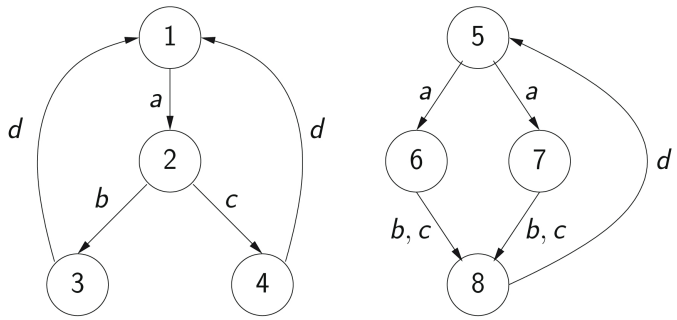

Abbildung 3.2.: Transitionssysteme für die Bisimularität

Satz 3.1.2 (Bisimularität und Sprachäquivalenz) *Gegeben sei das Transitionssystem* $LTS = (S, Act, \longrightarrow, I)$. *Es gilt:*

- *Falls* $\alpha \sim \beta$ *und* $\alpha \xrightarrow{\omega} \alpha'$ *für ein* $\omega \in Act^*$, *dann existiert* $\beta \xrightarrow{\omega} \beta'$ *und es folgt* $\alpha' \sim \beta'$.
- *Falls* $\alpha \sim \beta$ *so gilt auch* $\alpha \sim_L \beta$.

3.2. Klassische Algorithmen

In der Vergangenheit wurden verschiedene Algorithmen entwickelt die zur Automatischen Berechnung von Bisimulationen genutzt werden können. Im folgenden werden zwei dieser Algorithmen vorgestellt die sich dem Problem auf unterschiedliche Art und Weise annähern. Zum einen wird der Fixpunkt-Algorithmus betrachtet der komplette Bisimulationen errechnet, zum anderen der Hirschkoff-Algorithmus der die Bisimularität zweier Zustände verifiziert.

Fixpunkt-Algorithmus

Der Fixpunkt-Algorithmus ist ein iteratives Verfahren mit dem die vollständige Bisimularität $\sim$ berechnet werden kann.

Definition 3.2.1 (Fixpunkt-Algorithmus) *Gegeben sei das beschriftete Transitionssystem* $LTS = (S, Act, \longrightarrow, I)$.

- *Initialisiere* $\sim_0 = S \times S$
- $\sim_{n+1} \subseteq S \times S$, *wobei für* $\alpha, \beta \in S$ *genau dann* $\alpha \sim_{n+1} \beta$ *ist, wenn* $\forall a \in Act$ *gilt:*
  - *Für jedes* α' *mit* $\alpha \xrightarrow{a} \alpha'$ *existiert ein* β', *so dass* $\beta \xrightarrow{a} \beta'$ *und* $\alpha' \sim_n \beta'$
  - *Für jedes* β' *mit* $\beta \xrightarrow{a} \beta'$ *existiert ein* α', *so dass* $\alpha \xrightarrow{a} \alpha'$ *und* $\alpha' \sim_n \beta'$
- *Das Verfahren terminiert sobald* $\sim_n = \sim_{n+1}$.

Der Algorithmus selbst terminiert immer, kann aber im worst-case eine exponentielle Laufzeit besitzen.

Hirschkoff-Algorithmus

Der in [Hir99] definierte Hirschkoff-Algorithmus, überprüft die Bisimularität zweier Prozesse und berechnet auf Grundlage eines On-the-Fly Algorithmus aus [FM91] eine Bisimulation $\mathcal{R}$. On-the-Fly beschreibt die Eigenschaft, die Bisimulation $\mathcal{R}$ während des traversierens des LTS zu berechnen. Der zusammengefasste Algorithmus in Pseudocode:
$\mathcal{W} := \emptyset$;
$(*)$ $\mathcal{R} = \{(\mathcal{P}, \mathcal{Q})\}$, $\mathcal{V} := \emptyset$, $\mathcal{R} := \emptyset$, *status* := *true*; insert $(\mathcal{P}, \mathcal{Q})$ in $\mathcal{S}$;
while $\mathcal{S}$ is not empty and *status* = *true* do
  choose a pair $(\mathcal{P}_0, \mathcal{Q}_0)$ in $\mathcal{S}$ and remove it from $\mathcal{S}$;
  if $(\mathcal{P}_0, \mathcal{Q}_0)$ **succeeds** then (add $(\mathcal{P}_0, \mathcal{Q}_0)$ to $\mathcal{V}$ (or $\mathcal{R}$)); **propagate**) else
  if $(\mathcal{P}_0, \mathcal{Q}_0)$ **fails** then
    if $(\mathcal{P}_0, \mathcal{Q}_0) \in \mathcal{R}$
      then (remove $(\mathcal{P}_0, \mathcal{Q}_0)$ from $\mathcal{V}$ and insert it in $\mathcal{W}$, status:=false; **propagate**)
      else (insert $(\mathcal{P}_0, \mathcal{Q}_0)$ in $\mathcal{W}$; **propagate**);
  else *(* neither succeeds nor fails *)*
    compute the successors of $(\mathcal{P}_0, \mathcal{Q}_0)$ and insert them in $\mathcal{S}$;
endwhile;
if $\mathcal{P} \sim \mathcal{Q}$ then (if status then true else loop back to $(*)$) else false

Die Datenstruktur $\mathcal{S}$ ist die Worklist des Algorithmus. Die drei Listen $\mathcal{V}, \mathcal{R}$ und $\mathcal{W}$ beinhalten diejenigen Tupel von denen man vermutet das sie bisimular sind ($\mathcal{V}$), diejenigen von denen man weiß das sie bisimular sind ($\mathcal{R}$) und jene die nicht-bisimular sind ($\mathcal{W}$). Wenn es sich heraustellt, dass ein Paar bisimular ist, wird es von $\mathcal{V}$ nach $\mathcal{R}$ kopiert. Sind zwei Prozesse nicht bisimular, wird das Paar aus $\mathcal{V}$ entfernt und in $\mathcal{W}$ gespeichert. Es kann passieren, dass ein als bisimular markiertes Paar im späteren Verlauf des Algorithmus als nicht-bisimular eingeordnet wird. In einem solchen Fall bricht der Algorithmus den aktuellen Suchlauf ab und das status Flag wird auf false gesetzt. Der Algorithmus merkt sich in einem solchen Fall alle nicht-bisimularen Paare und beginnt von vorne. Die Methoden **succeed**,**fails** und **propagate** erfüllen folgende Funktionen:

succeed Gibt true zurück falls beide Prozesse trivial bisimular oder als bisimular eingestuft werden können.

fails Liefert true zurück falls das betrachtete Paar bereits in $\mathcal{W}$ liegt oder falls ein Prozess einen Schritt machen kann den der andere nicht nachmachen kann.

propagate Propagiert neue Erkenntnisse über die Relation an alle Listen

Der Hirschkoff-Algorithmus ist im Gegensatz zum Fixpoint Algorithmus ein schnelleres Verfahren um die Bisimularität zweier Prozesse zu untersuchen. Dabei wird nicht die größte Bisimulation verfeinert, sondern eine Bisimulation auf Grundlage des Ausgangspaares gebildet. In Kapitel 5 wird der klassische Hirschkoff-Algorithmus modifiziert.

4. Graphtransformation mit Borrowed Context

Der Hauptnutzen des Boworred Kontext liegt in der Gewinnung eines LTS, zur Verhaltensbestimmung von Graphenersetzungssystemen. Graphen repräsentieren hierbei die Zustände. Die Transitionen werden durch die Ersetzungsregeln mit ihren jeweiligen *Borrowed Context* gewonnen [Hül10]. Der DPO Ansatz liefert unbeschriftete Transitionen deren einzig sinnvolle Beschriftung die angewandten Regeln wären. Diese Beschriftungen wären allerdings nicht sinnvoll, da es sich um interne Aktionen handelt die nicht beeinflusst werden könnten. Um sinnvolle beschriftete Transitionen zu Gewinnen wird der DPO Ansatz im Folgenden zum Borrowed Context erweitert.

4.1. Pushout

Pushout Konstruktionen werden für den Double-Pushout Ansatz (DPO) benötigt.

Definition 4.1.1 (Pushout) *Ein Pushoutobjekt ist das Ergebnis wenn zwei Graphen mittels eines gemeinsamen Subgraphen zusammengeklebt werden. Wie in Abbildung 4.1 dargestellt wird ein Pushout (D, f', g') über den gegebenen Morphismen $f : A \to B$ und $g : A \to C$ einer Kategorie C gewonnen. Der Pushout beinhaltet:*

- *Das Pushoutobjekt D*
- *Die Morphismen $f' : C \to D$ und $g' : B \to D$ mit $f' \circ g = g' \circ f$*

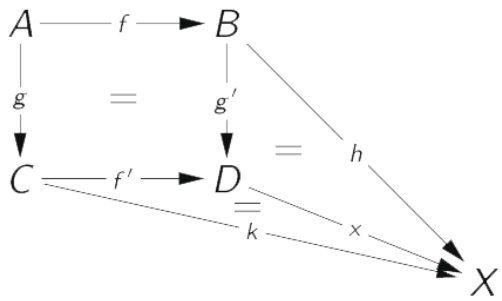

Abbildung 4.1.: Pushout Konstruktion

Für alle Objekte X und Morphismen $h : B \to X$ sowie $k : C \to X$ mit $k \circ g = h \circ f$ existiert ein eindeutiger Morphismus $x : D \to X$ sodass $x \circ g' = h$ und $x \circ f' = k$ gilt. Das Pushoutobjekt D ist das Ergebnis des Verklebens von B und C über A [EEPT06].

Beispiel 4.1.1 (Pushout) *Gegeben sei folgende Situation in Abbildung 4.2. Die Morphismen zwischen den Graphen sind durch die Nummerierung der Knoten erkennbar.*

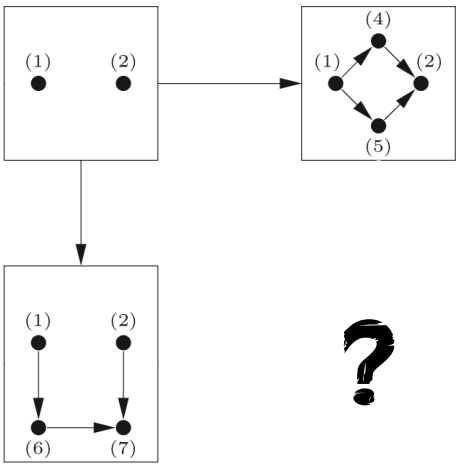

Abbildung 4.2.: Pushout Beispiel Aufgabe

Mittels der Pushout Konstruktion gewinnen wir das Pushoutobjekt mit den dazugehörigen Morphismen die in Abbildung 4.3 dargestellt sind.

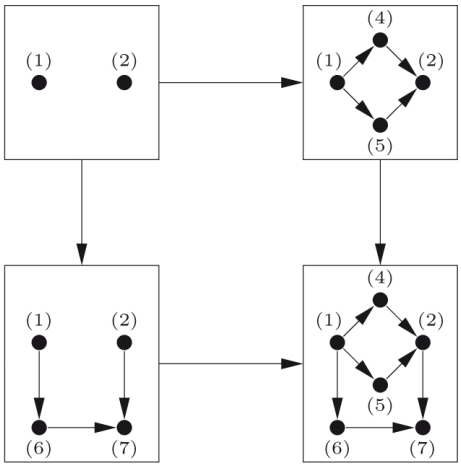

Abbildung 4.3.: Pushout Beispiel Lösung

4.2. Pullback

Pullback Konstruktionen werden im Borrowed Context Ansatz benötigt.

Definition 4.2.1 (Pullback) *Ein Pullbackobjekt ist das Ergebnis eines Schnitts von zwei Graphen über einem gemeinsamen Graphen. Wie in Abbildung 4.1 dargestellt wird ein Pullback (A, f', g') über den gegebenen Morphismen $f : C \to D$ und $g : B \to D$ einer Kategorie C gewonnen.*

Der Pullback beinhaltet:

- *Das Pullbackobjekt A*
- *Die Morphismen $f' : A \to B$ und $g' : A \to C$ mit $g \circ f' = f \circ g'$*

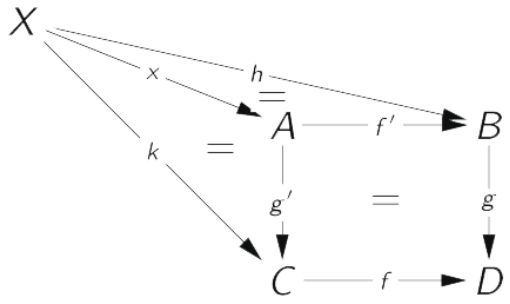

Abbildung 4.4.: Pullback Konstruktion

Für alle Objekte X und Morphismen $h : X \to B$ sowie $k : X \to C$ mit $f \circ k = g \circ h$ existiert ein eindeutiger Morphismus $x : X \to A$ sodass $f' \circ x = h$ und $g' \circ x = k$ gilt. Das Pullbackobjekt A ist das Ergebnis des Schnitts von B und C über D [EEPT06].

Beispiel 4.2.1 (Pullback) *Gegeben sei folgende Situation in Abbildung 4.5. Die Morphismen zwischen den Graphen sind durch die Nummerierung der Knoten erkennbar.*

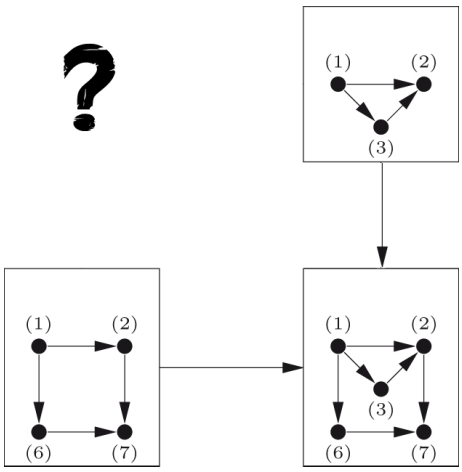

Abbildung 4.5.: Pullback Beispiel Aufgabe

Mittels der Pullback Konstruktion gewinnen wir das Pullbackobjekt mit den dazugehörigen Morphismen die in Abbildung 4.6 dargestellt sind.

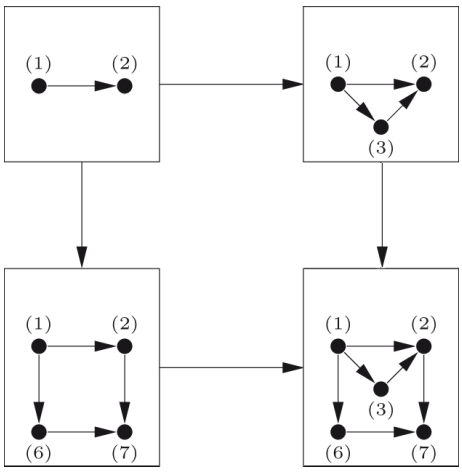

Abbildung 4.6.: Pullback Beispiel Lösung

4.3. Double-Pushout

Als einer der wichtigsten Ansätze der algebraischen Graphenersetzung, benötigen wir den DPO Ansatz um unbeschriftete Transitionssysteme zu gewinnen. Wir benötigen den DPO Ansatz da er die Grundlage für den Borrowed Context Schritt darstellt.

Definition 4.3.1 (DPO Ansatz) *Beim DPO Ansatz handelt es sich um ein Graphenersetzungssystem bestehend aus einem Paar von Regeln* $(\varphi_L : I \to L, \varphi_R : I \to R)$. *Alternativ kann auch* $(L \overset{\varphi_L}{\leftarrow} I \overset{\varphi_R}{\rightarrow} R)$ *geschrieben werden. L steht hierbei für die Vorbedingung der Regel und R für die geltende Nachbedingung. I steht für den Bereich des Graphen der vorhanden sein muss, um die Regel anzuwenden und der bei der Ersetzung nicht verändert wird. Ein Graph G kann in einen Graphen H überführt werden, sofern es einen injektiv passenden Morphismus* $\varphi : L \to G$ *gibt* [1]. *Zur Veranschaulichung dient Abbildung 4.7. Den Namen Double-Pushout verdankt dieser Ansatz dem Vorgehen über den Linken Pushout den Zwischengraphen C zu finden und aus dem Verkleben von C mit R über I, das zweite Pushoutobjekt H zu gewinnen.*

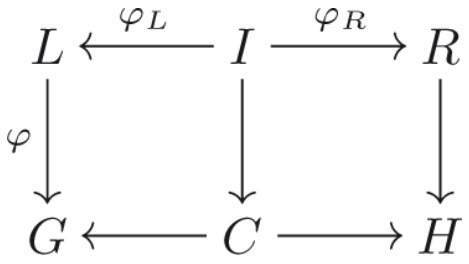

Abbildung 4.7.: Der DPO Ansatz [EK04]

[1] L muss vollständig in G enthalten sein

Um einen Graphen G in einen Graphen H zu überführen erzeugt man als erstes einen Zwischengraphen C. Von G werden die passenden Knoten und Kanten von L entfernt die kein vorkommen in I haben. Dieser Schritt darf nur durchgeführt werden sofern nach dem entfernen von L über I keine Kanten in der Luft hängen bleiben (die sogenannte *Dangling Edge Condition*). G bildet das Pushoutobjekt von C und L über I. Anschließend wird das Pushoutobjekt H durch das Verkleben von C und R über I gebildet [EK04].

Beispiel 4.3.1 (Double-Pushout) *Gegeben sei folgende Situation in Abbildung 4.8. Die Morphismen zwischen den Graphen sind durch die Nummerierung der Knoten erkennbar.*

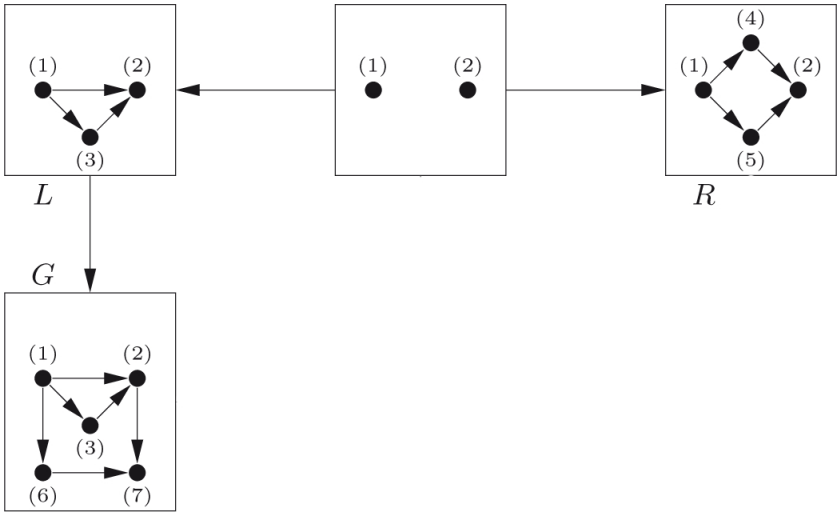

Abbildung 4.8.: Double-Pushout Beispiel Aufgabe

Wie in Abbildung 4.8 dargestellt wird mittels des Pushoutcomplements der Zwischengraph C mit den dazugehörigen Morphismen erzeugt.

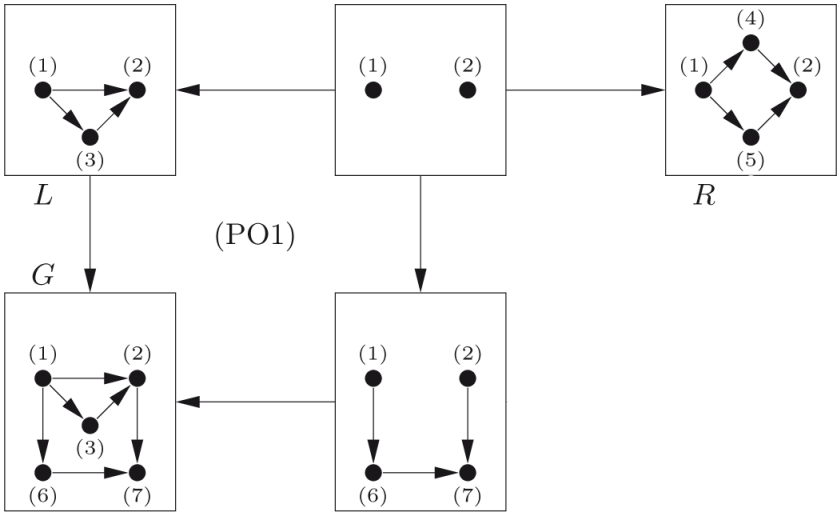

Abbildung 4.9.: Double-Pushout Beispiel Zwischenschritt

Zuletzt folgt der zweite Pushout zwischen den Graphen C und R über gemeinsamen dem Graphen I. Der resultierende Graph H mit den fehlenden Morphismen ist in Abbildung 4.10 abgebildet.

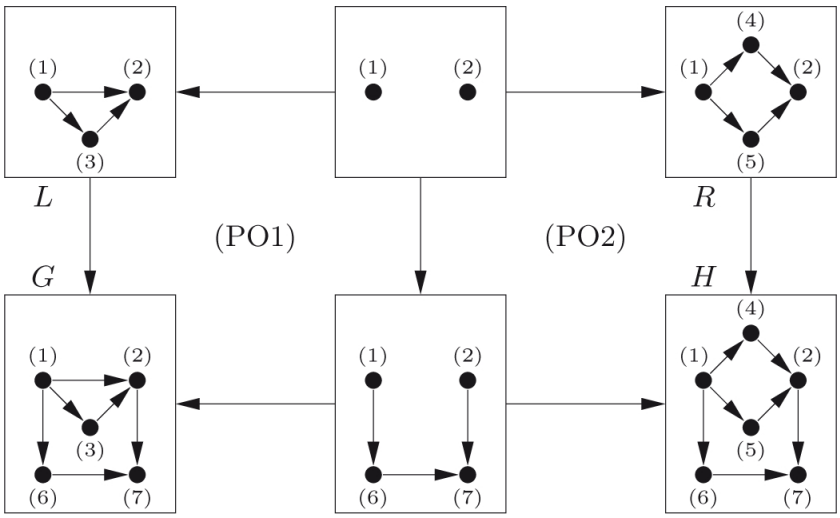

Abbildung 4.10.: Double-Pushout Beispiel Lösung

Mit dem DPO Ansatz gelingt es nun ein unbeschriftetes Transitionssystem zu definieren. Die Graphen G, H beschreiben die Zustände und durch die vom System gegebene Regel existiert eine unbeschriftete Transitionsrelation $G \to H$. Um ein LTS zu generieren muss der DPO Ansatz nun zu Borrowed Context erweitert werden.

4.4. Interface und Kontext

Der *Borrowed Context* wird im Zuge von offenen Systemen erläutert. Offene Systeme bieten dem Betrachter Schnittstellen (sogenannte Interfaces) an, über die Graphen angesprochen werden können. Desweiteren existieren Kontexte die auf bestimmte Graphen und ihre Interfaces einfluss nehmen können.

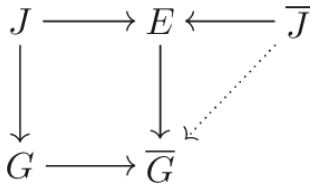

Abbildung 4.11.: Interface und Kontext [EK04]

Definition 4.4.1 (Interface und Kontext) *Ein Interface J ist ein injektiver Morphismus $J \to G$. Diese Schreibweise bedeutet, das der Graph G das Interface J besitzt. Ein Kontext E besteht aus zwei injektiven Morphismen der Form $J \to E \leftarrow \bar{J}$. Zusammengenommen erhält man über den Pushout von G und E über dem Interface J den Graphen $\bar{G}$ mit dem Interface $\bar{J}$, folglich $\bar{J} \to \bar{G}$ (siehe Abbildung 4.11). Der Graph $\bar{G}$ mit Interface $\bar{J}$ modelliert den Zustand in den der Graph G mit Interface J im Kontext E wechseln kann [EK04].*

4.5. Borrowed Context Ansatz

Nun wollen wir den minimalen Kontext berechnen der als Transitionslabel für die berechneten LTS genutzt werden kann.

Definition 4.5.1 (Borrowed Context) *Ein Graph G mit Interface J $(J \to G)$ kann zu einem Graphen H mit Interface K $(K \to H)$ mittels einem Transitionsabbild $(J \to F \leftarrow K)$ überführt werden, sofern es eine Ersetzungsregel $(L \leftarrow I \to R)$ gibt, bei der L teilweise in G passt. Dazu werden die Graphen D, G^+ und C benötigt sowie weitere Morphismen, sodass die Quadrate in Abbildung 4.12 Pushouts und Pullbacks sind.*

Zu Beginn wird der Graph D aufgestellt, der alle passenden Knoten und Kanten von L in G beinhaltet. Liegt L total in G, könnte der reguläre DPO Ansatz durchgeführt werden um G in H zu überführen. Mit Hilfe des Borrowed Kontext ist diese Überführung möglich, sofern L nur teilweise in G vorkommt. Die passenden Teilstücke sind beschrieben durch $(G \leftarrow D \to L)$. G^+ ist das Pushoutobjekt von G und L über D. L kommt in G^+ total vor, wodurch der DPO Ansatz genutzt werden kann um C und H zu gewinnen. F kann über einen Pushout konstruiert werden. F ist der minimale Kontext der fehlt damit L total in G passt und wird als *Borrowed Context* bezeichnet. Zusätzlich wird über einen Pullback mittels der Graphen F und C über G^+ das Interface K für H ermittelt. Zusammengefasst wird eine Transition der Form $(J \to G) \overset{J \to F \leftarrow K}{\to} (K \to H)$ gewonnen [EK04].

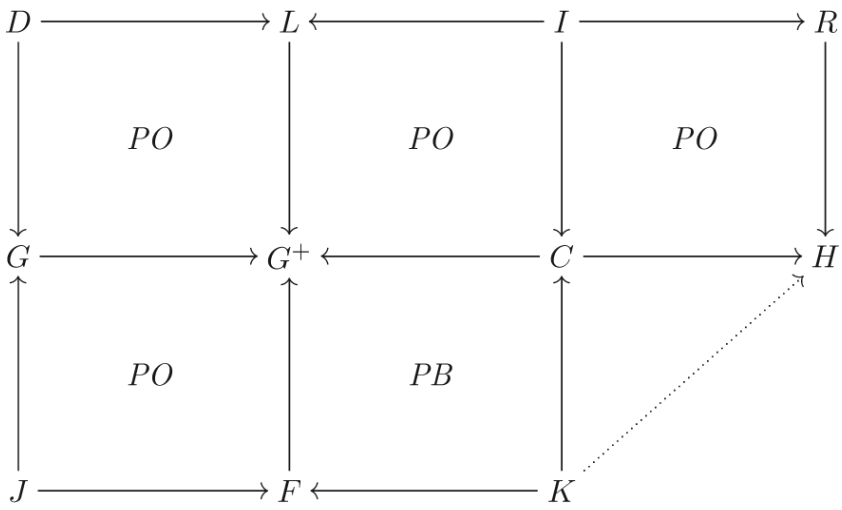

Abbildung 4.12.: Graphenersetzen mit Borrowed Context [EK04]

Beispiel 4.5.1 (Onlinemarkt) *Als Beispiel soll ein Onlinemarkt aus [BEK06] dienen. Ein User soll in der Lage sein dort Objekte die er besitzt anzubieten, oder Objekte im Markt anzufragen und zu kaufen. Abbildung 4.13 repräsentiert einen Satz von Ersetzungsregeln die das Verhalten des Prozesses modellieren. Alle Regeln basieren auf den Knoten U, A, O, Buy und Sell. Der Knoten U repräsentiert einen* User*, der verschiedene* Objekte *O besitzt und mit dem* Markt *A interagieren kann. Die Kanten sind in diesem Beispiel unbeschriftet. Die Knoten Buy und Sell stehen für eine Aktion, die sich auf beteiligte Knoten und Kanten auswirkt. So fügt Buy eine Kante zwischen einem User U und dem gekauften Objekt O hinzu, während Sell eine solche Kante nach durchführung der Aktion entfernt. Die Knoten Buy und Sell sind als Zeiger zu verstehen die verdeutlichen, dass die jeweilige Aktion auf dem Graphen durchgeführt werden kann.*

Es folgt eine kurze Beschreibung der Bedeutungen aller Ersetzungsregeln in Abbildung 4.13:

Offer *In der Vorbedingung besitzt der User U ein Objekt O und eine Verbindung zum Markt A. Die Regel* Offer *besagt, dass der User U sein Objekt O zum Verkauf anbieten kann. Dargestellt wird dies durch den Graphen in der Nachbedingung von* Offer*.*

Ask *In der Vorbedingung besitzt der User U eine Verbindung zum Markt A. Der User U kann nun verschiedene Objekte O anfragen die er wie in der Nachbedingung festgehalten kaufen kann.*

Buy *Ein Objekt O kann über einen Markt A seinen Besitzer/User U wechseln. In der Vorbedingung müssen zwei User U mit dem Markt A verbunden sein. Der User U der das Objekt O besitzt, muss es verkaufen wollen und der User U der es gerne hätte muss die Anfrage an den Markt A stellen es zu kaufen. In der Nachbedingung wechselt das Objekt O den User U.*

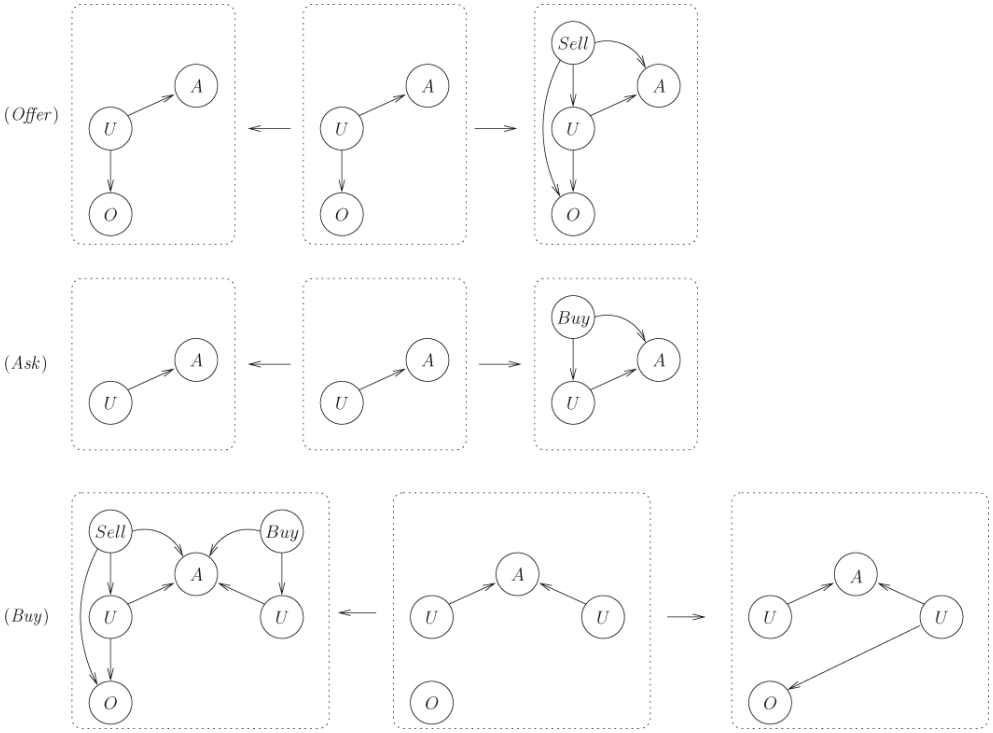

Abbildung 4.13.: Ersetzungsregeln für den Onlinemarkt

Nun wird von folgendem Beispielscenario ausgegangen das in Abbildung 4.14 dargestellt ist. Der Graph G modelliert einen User U, der im Besitz von zwei Objekten O ist. Über die Ersetzungsregel Buy *aus Abbildung 4.13 soll eines der beiden Objekte O über einen Onlinemarkt A verkauft werden. Als Interface wird ein Markt A vorrausgesetzt über den interagiert werden soll.*

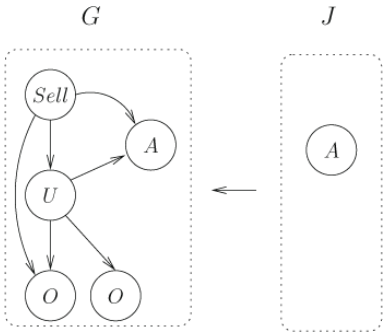

Abbildung 4.14.: Graph G mit Interface J

Soll der Graph G nun mit der Ersetzungsregel Buy *in einen Graphen H überführt werden, wird mit dem Vergleich passender Knoten und Kanten zwischen dem Graphen und der Vorbedingung begonnen. Die Vorbedingung L der Ersetzungsregel* Buy *passt nur teilweise in den Graphen G. Die Überführung in den Graphen H muss deshalb mittels dem Borrowed Context erfolgen. Läge L total in G, könnte der reguläre DPO Ansatz für die Überführung benutzt werden.*

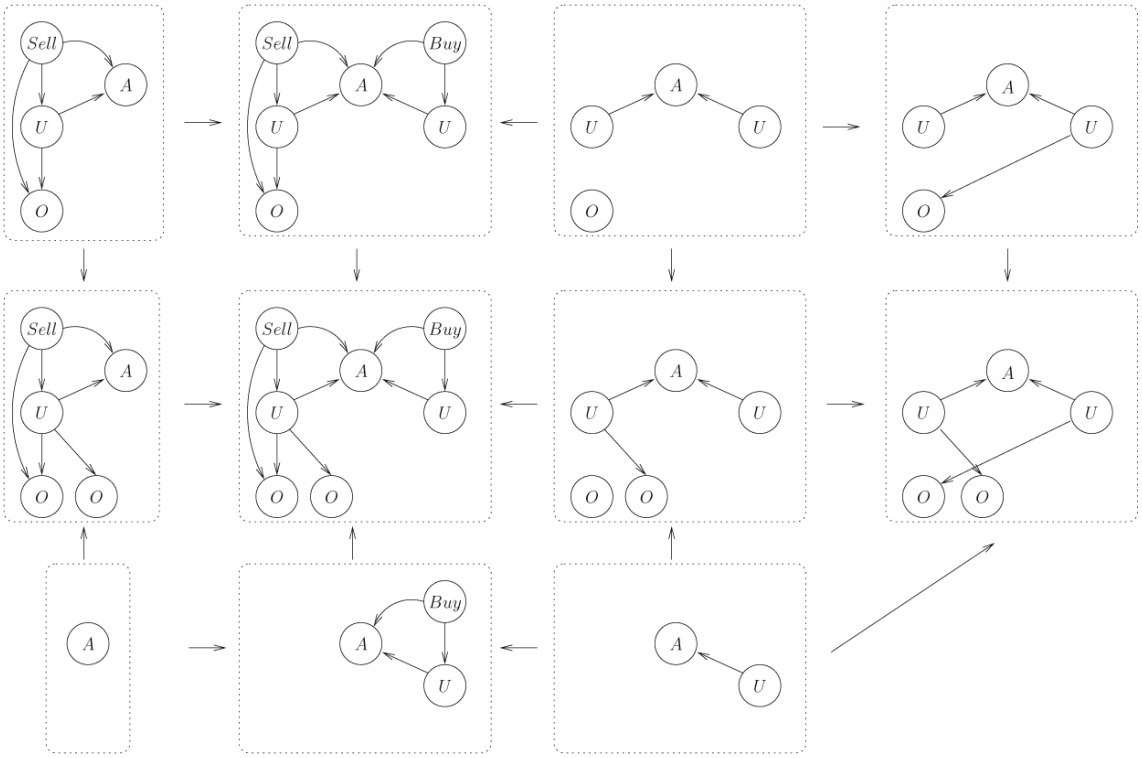

Abbildung 4.15.: Borrowed Context Schritt mit Onlinemarktregel *Buy*

Anhand Abbildung 4.15 lässt sich die Überführung erläutern. Beginnend mit den partiellen Übereinstimmungen wird der Graph D (ganz oben links) konstruiert. Aus dem Pushout vom Verkleben der Graphen G und L über D wird der Graph G^+ gewonnen. Der Graph G^+ beinhaltet die vollständige Vorbedingung aus Buy *sowie den Graphen G. Über den Standard DPO Ansatz wird G^+ in den Graphen H (mitte ganz rechts) überführt. Dazu wird über das Interface I, der Ersetzungsregel* Buy*, der Zwischengraph C erzeugt. Mittels einem Pushout durch das Verkleben von C mit der Nachbedingung R über I wird der Graph H gewonnen. In H sieht man nun ein Modell für die Situation, das der verkaufende User U sein Objekt O verloren hat und der kaufende User U auf der rechten Seite dieses Objekt besitzt [BEK06].*

Die unterste Zeile der Abbildung 4.15 zeigt $(J \rightarrow F \leftarrow K)$, wobei

- *J das Interface des Graphen G ist,*
- *F den* Borrowed Context *angibt,*
- *K das Interface des resultierenden Graphen H ist.*

Der Borrowed Context *F zeigt, dass die Ersetzungsregel nur angewendet werden kann wenn der Kontext einen weiteren User U zur Verfügung stellt der zum Markt A verbunden ist und das Objekt O kaufen möchte.*

Aus dieser Information kann nun ein LTS aufgestellt werden. Ein User mit zwei Objekten von denen er nur eines verkaufen möchte kann ebenfalls über die Regel Buy *den Zustand wechseln sofern der Kontext ihm einen anderen User zur Verfügung stellt der das Objekt kaufen möchte (siehe Abbildung 4.16). Gleichzeitig ist dieser gefundene fehlende Kontext minimal und liefert zusätzlich Bisimularität was aus [EK04] entnommen werden kann. Dadurch können Verhaltensäquivalenzen in Graphenersetzungssystemen auf Basis von LTS untersucht werden.*

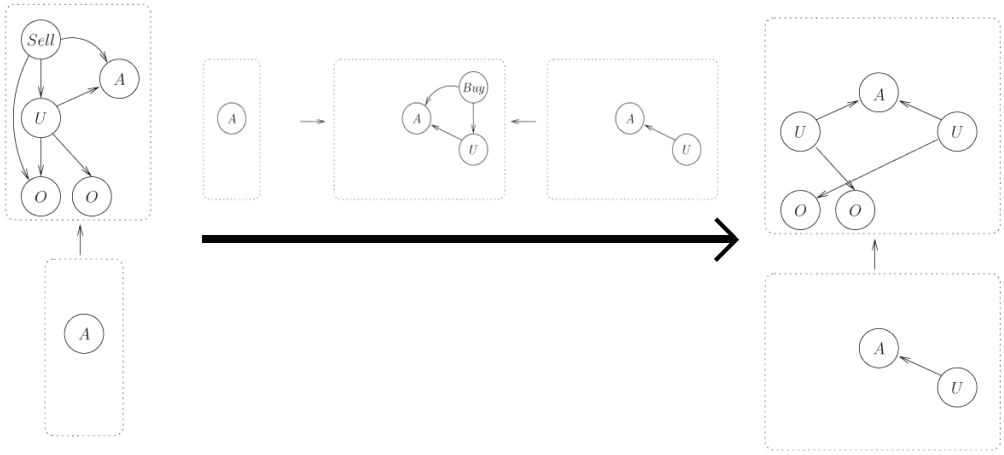

Abbildung 4.16.: Resultierende Transition

In [EK04] wurde die Kongruenz der Bisimularität für Relationen mit Graphenpaaren bewiesen. Sofern $(J \to G) \sim (J \to G')$ gilt, folgt mit der Abgeschlossenheit unter einem Kontext $c[(J \to G)] \sim c[(J \to G')]$. Sofern $\mathcal{R}$ eine Bisimulation ist, folgt dass jede unter Kontext abgeschlossene Relation $\hat{\mathcal{R}}$ ebenfalls eine Bisimulation ist. Daraus folgt die Kongruenz der Bisimularität $\sim$.

4.6. Optimierungen

Aus den vorherigen Kapiteln ist nun ersichtlich, dass für jeden Teilgraphen D eine neue Transition durch den Borrowed Context berechnet werden kann. Um die Bisimulation möglicherweise zu verkleinern, werden nun Optimierungen durchgeführt. Pruning hilft bereits beim Berechnen der Transitionen Ergebnisse ohne Aussage zu filtern während Up-To-Context dazu genutzt wird die resultierenden Relationspaare zu beschränken.

4.6.1. Pruning

Interessant im Sinne des Borrowed Context sind nur partielle Matches mit denjenigen Teilgraphen D, bei denen mehr als das definierte Interface vorhanden ist. Ursprünglich entstammt der Begriff dem Prozess des Optimierens von Entscheidungsbäumen. Wir adaptieren diesen Begriff und vereinfachen die Relation, indem die berechneten Transitionen auf diejenigen Borrowed Context reduziert werden, bei denen nicht das Interface geborgt werden muss. In Abbildung 4.17 ist die Situation dargestellt.

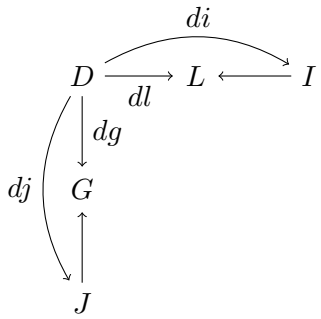

Abbildung 4.17.: Teilweises Match Situation

Definition 4.6.1 (Pruning im Borrowed Context) *Jedes durch das partielle Match berechnete Tripel (D, dl, dg) ist nur dann für den Borrowed Context relevant sofern es keine Morphismen $dj : D \to J$ und $di : D \to I$ gibt, so dass das Diagramm 4.17 kommutiert.*

Grundsätzlich ermöglicht der Borrowed Context jeden Graphen durch jede mögliche Regel zu transformieren da die Precondition komplett geborgt werden könnte. Durch das Prunen der Teilgraphen D werden nur interessante Borrowed Context betrachtet.

4.6.2. Up to Context

Beim Berechnen der Relationspaare unserer Bisimulation kann es sein das Paare gefunden werden die bereits in der Relation enthalten sind jedoch um einen Context erweitert wurden. Diese erweiterten Paare müssen nicht weiter beachtet werden da sie durch ihre kleineren Vertreter ohnehin in der Relation beschrieben sind. Die Kongruenzsätze erlauben es an dieser Stelle diese größeren Paare rauszunehmen. Die Situation wird in

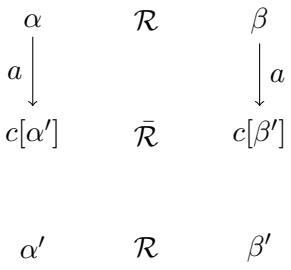

Abbildung 4.18.: Bisimulation Up-to-Context

Abbildung 4.18 veranschaulicht. Das Paar $(c[\alpha'], c[\beta'])$ kann aus der Relation entfernt werden, da (α', β') bereits enthalten ist.

Um diesen Context zu bestimmen, müssen nun die Relationspaare für Graphen genauer betrachtet werden. Ein zusätzlicher Borrowed Context ist genau dann gegeben wenn folgende Situation existiert: Das Relationstupel $(J \to G, J \to G')$ wurde mit Borrowed Context zu $(K \to H, K \to H')$ transformiert. Existiert nun ein Kontextgraph C sodass die Abbildung 4.19 existiert ist, kann das berechnete Paar $(K \to H, K \to H')$ aus der Relation entfernt werden.

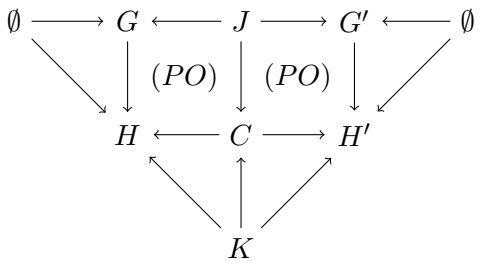

Abbildung 4.19.: Up-to-Context Diagramm für Graphenpaare

5. Algorithmen

Im diesem Kapitel geht es um die Beschreibung iterativer Algorithmen, die alle bisher benötigten Kriterien dieser Ausarbeitung so automatisieren, dass sie in Mathematika implementiert werden können. Die Algorithmen in dieser Ausarbeitung werden zuerst in Pseudocode definiert. Zum Start definieren wir eine Syntax der Algorithmen.

Für If-Verzweigungen legen wir folgende Syntax fest:

Definition 5.0.2 (Syntax der alternativen Auswahl mit zwei Zweigen)
FALLS *Bedingung*
DANN
Algorithmusschritte für Bedingung erfüllt
SONST
Algorithmusschritte für Bedingung nicht erfüllt
ENDE FALLS

While-Schleifen definieren wir mit:

Definition 5.0.3 (Syntax der While-Schleife im Algorithmus)
SOLANGE *Bedingung erfüllt ist*
FÜHRE AUS
Algorithmusschritte
ENDE SOLANGE

Um die Elemente einer Liste vollständig zu iterieren benötigen wir For-Schleifen der folgenden Form:

Definition 5.0.4 (Syntax der For-Schleife im Algorithmus)
FÜR $\forall\ x \in X$
Algorithmusschritte
ENDE FÜR

Für Rückgabeparameter verwenden wir das Schlüsselwort (**RÜCKGABE** Parameter) und für einen allgemeinen Algorithmustitel (**Algorithmus** Name Übergabeparameter). Hilfsfunktionen werden vor der Algorithmusdefinition im Abschnitt **HILFSFUNKTIONEN** erläutert. Kommentare zum Algorithmus dokumentieren in [Text] Blöcken die jeweiligen Zeilen und tragen so zum Verständniss bei. Wir beginnen mit dem Problem der Bisimulationsberechnung und beschreiben im Anschluss algorithmische Ansätze für die Graphenersetzung und ihre Optimierungen bezüglich der gewonnenen Relationen. Dabei arbeiten wir vorwiegend mit Listen.

5.1. Algorithmen für Transitionssysteme

Ausgangspunkt für die Berechnung der Bisimulationen, sind die aus der Graphersetzung gewonnenen LTS. Gegeben $LTS_1 = (S_1, Act_1, \longrightarrow_1, I_1)$ und $LTS_2 = (S_2, Act_2, \longrightarrow_2, I_2)$. Sofern weitere Funktionen benötigt werden um kleinere Teilprobleme zu lösen, werden diese stets angegeben.

5.1.1. Bisimulationscheck

Die Grundidee des Bisimulationschecks, basiert auf dem Erweitern einer Anfangsrelation (α, β) zu einer Bisimulation $\mathcal{R}$. Wir benötigen einen Algorithmus der überprüft ob für ein gegebenes Zustandspaar $\alpha \sim \beta$ gilt. Hierbei ist $\alpha \in S_1$ und $\beta \in S_2$. Zusätzlich soll der Algorithmus alle direkten Folgetupel (α', β') berechnen. Hilfsfunktionen in diesem Algorithmus sind die Funktionen:

HILFSFUNKTIONEN
$sources : Act \to S$ [Berechnet eine Liste aller Quellknoten für $a \in Act$]
$targets : Act \times S \to S$ [Berechnet eine Liste aller Zielknoten für $a \in Act$]
$acts_i : S_i \to Act_i$ [Liefert die Label-Liste, die $s \in S_i$ als Quellknoten besitzen]
$childs_1 : \mathcal{R} \to C$ [Liefert Menge $C := \{\alpha \in S_1 \mid \exists \beta \in S_2$ und $(\alpha, \beta) \in \mathcal{R}\}$]
$childs_2 : \mathcal{R} \to C$ [Liefert Menge $C := \{\beta \in S_2 \mid \exists \alpha \in S_1$ und $(\alpha, \beta) \in \mathcal{R}\}$]

Wir beginnen mit einem Algorithmus für eine Methode die berechnet ob ein Tupel (α, β) einen simulierenden Schritt machen können. Die Variable isBisim gibt an ob dieser Schritt existiert und die Liste results speichert alle erreichbaren Tupel.

ALGORITHMUS BisimulationsCheckfürTuple(LTS1,LTS2,(α, β)**)**
01 Liste $act1 := acts_1(\alpha)$; [Liste der möglichen Züge in α]
02 Liste $act2 := acts_2(\beta)$; [Liste der möglichen Züge in β]
03 Liste result := $\emptyset$; [Menge mit allen erreichbaren Ergebnistupeln]
04 Boolean isBisim := false; [Bisimulations Flag]
05 **FALLS** $act1 == act2$
06 **DANN**
07 isBisim := true;
08 **FÜR** $\forall a \in Act_1$
09 $children_1 := targets(a, \alpha)$;
10 $children_2 := targets(a, \beta)$;
11 $results := results \cup \{(\alpha', \beta') \mid \alpha' \in children_1$ und $\beta' \in children_2\}$;
12 **ENDE FÜR**
13 **ENDE FALLS**
14 **RÜCKGABE** $(isBisim, results)$

Wir erweitern den Algorithmus zur Überprüfung kompletter Bisimulationen. Der bisherige Algorithmus überprüft lediglich ein einzelnes Tupel. Nun definieren wir eine Methode die überprüft ob die Nachfolgetupel eines gegebenen Zustandspaare noch ausreichen um eine Bisimulation aufzustellen. Diese Methode nennen wir AusreichendNachfolger. Als Eingabe erhält sie die zu betrachteten Nachfolgertupel in der Liste *result* sowie eine Liste *notbisim* aller bekannten Paare die nicht-bisimular sind.

ALGORITHMUS AusreichendNachfolger(results,notbisim)
01 $(hit1, hit2) := (childs_1(results), childs_2(results)$; [Erreichbare Zustände]
02 $restchilds := results \setminus notbisim$; [Nicht bisimulare Paare filtern]
03 $(rest1, rest2) := (childs_1(restchilds), childs_2(restchilds))$; [Restzustände]
04 **FALLS** $(hit1 == rest1)$ und $(hit2 == rest2)$
05 **DANN** [Ausreichend Folgetupel vorhanden]
06 **RÜCKGABE** true;
07 **SONST** [Nicht ausreichend Folgetupel vorhanden]
08 **RÜCKGABE** false;
09 **ENDE FALLS**

Wir überprüfen die erreichten Zustände vor einer Filterung (Tupel $(hit1, hit2)$) und nach einer Filterung (Tupel $(rest1, rest2)$), aller nicht-bisimularen Nachfolger. Ist die Menge der Zustände konstant geblieben sind alle vorher erreichten Zustände immernoch erreichbar und es sind noch ausreichend Nachfolger für eine Bisimulation vorhanden.

Wir definieren nun den Bisimulations-Check-Algorithmus der auf Grundlage unseres Tupel-Ansatzes arbeitet. Dabei ist die Liste *relation* unsere mögliche Bisimulation, die durch den Algorithmus verifiziert wird. Der Boolean isBisim kennzeichnet für jeden Schritt, ob das gerade betrachtete Tupel einen simulierenden Schritt ausführen kann.

ALGORITHMUS BisimulationsCheckKomplett(LTS1,LTS2,$\mathcal{R}$)
01 Set $relation := \mathcal{R}$; [Zu überprüfende Relation]
02 Set $notBisim := \emptyset$; [Menge der Nachfolger die nicht in der Relation vorkommen]
03 Boolean isBisim := true; [Gleiches Verhalten Flag]
04 **FÜR** $\forall r \in relation$
05 $(isBisimT, results)$:= BisimulationsCheckfürTuple(LTS1,LTS2,r);
06 **FÜR** $\forall\ (\alpha, \beta) \in results$
07 **FALLS** $(\alpha, \beta) \notin relation$
08 **DANN** [Alle Nachfolger die nicht in der Relation stehen werden gemerkt]
09 $notbisim := notbisim \cup \{(\alpha, \beta)\}$;
10 **ENDE FALLS**
11 **ENDE FÜR**
12 isBisim := isBisim $\wedge$ AusreichendNachfolger(results,notbisim);
13 **FALLS** $\neg$isBisim
14 **DANN**
15 **RÜCKGABE** $(False)$;
16 **ENDE FALLS**
17 **ENDE FÜR**
18 **RÜCKGABE** $(isBisim)$

Der Bisimulations-Check erwartet als Eingabe zwei LTS sowie eine zu verifizierende Bisimulation $\mathcal{R}$. Für jedes Tupel wird der Bisimulations-Check für Tupel durchgeführt [Zeile 5]. Es wird überprüft welche Nachfolger in der Bisimulation $mathcalR$ enthalten sind [Zeilen 6-11] und ob diese Nachfolger ausreichen, um für das betrachtete Tupel einen simularen Schritt zu garantieren [Zeile 12]. Wir betrachten nun verschiedene Algorithmen die zur Überprüfung der Bisimularität zweier Zustände genutzt werden.

5.1.2. Greedy-Algorithmus

Wir definineren nun den Greedy Algorithmus, bei dem ein Paar (α, β) mit $\alpha \in S_1$ und $\beta \in S_2$ genau dann bisimular $((\alpha \sim \beta))$ ist, sofern ALLE erreichbaren Kinder in der Bisimulation liegen.

ALGORITHMUS GreedyBisimulation(LTS1,LTS2,(α, β)**)**
01 Set $relation := (\alpha, \beta)$; [Ergebnisrelation]
02 Boolean isBisim := true; [Gleiches Verhalten Flag]
03 **FÜR** $\forall r \in relation$
04 $(isBisimT, results)$:= BisimulationsCheckfürTuple(LTS1,LTS2,r);
05 $relation := relation \cup results$;
06 isBisim := isBisim $\wedge$ isBisimT;
07 **FALLS** $\neg$isBisim
08 **DANN**
09 **RÜCKGABE** $(False, (\alpha, \beta))$;
10 **ENDE FALLS**
11 **ENDE FÜR**
12 **RÜCKGABE** $(isBisim, relation)$

Im Gegensatz zum Bisimulations-Check-Algorithmus bauen wir beginnend mit einem Starttupel (α, β) dessen Bisimularität wir zeigen wollen, eine Bisimulation, in der sämtliche gefundene Nachfolger enthalten sein sollen. Besonders zu tragen kommt diese Variante in deterministischen LTS, bei denen alle Nachfolger in der Bisimulation enthalten sein müssen. Laufzeit wird hierbei durch das Fehlen des Nachfolger-Checks gespart.

5.1.3. Fixpunkt-Algorithmus

Der Fixpunkt-Algorithmus berechnet die Bisimularität $\sim$ aus zwei gegebenen LTS. Die Hilfsfunktionen für den Fixpunkt-Algorithmus lauten:

HILFSFUNKTIONEN
$count : \mathcal{R} \to \mathbb{N}$ [Liefert die Gesamtanzahl der Tupel in der Relation $\mathcal{R}$]

Zu Beginn des Fixpunkt Algorithmus wird die größtmögliche Relation *bisimRelation* mit dem Kreuzprodukt initialisiert. Der Algorithmus startet mit dem Bisimulations-Check für Tuple [Zeile 08]. Kann ein simulierender Schritt gemacht werden, so werden alle erreichbaren Tupel mit Hilfe der Methode AusreichendNachfolger genauer betrachtet. Hat das Tupel nicht genug Nachfolger um eine Bisimulation aufzustellen wird es in *notbisim* aufgenommen [Zeile 15]. Kann kein simulierender Schritt ausgeführt werden so wird das betrachtete Tupel in die Liste *notbisim* aufgenommen [Zeile 19]. Im Anschluss entfernen wir alle nicht bisimularen Tupel aus unserer Bisimulation *bisimRelation*. Ist *bisimRelation* nicht kleiner geworden geben wir die gefundene Bisimulation aus, andernfalls durchlaufen wir den Algorithmus erneut.

ALGORITHMUS BaueFixpunktRelation(LTS1,LTS2)
01 Liste $bisimRelation := \{(\alpha, \beta) \mid \alpha \in S_1$ und $\beta \in S_2\}$; [$\sim_0$]
02 Liste $notbisim := \emptyset$; [Menge der Tupel die nicht Bisimular sind]
03 Boolean finished := false; [Algorithmus ist terminiert Flag]
04 **SOLANGE** ¬finished
05 **FÜHRE AUS**
06 $startAnzahl := count(bisimRelation)$;
07 **FÜR** ∀ r ∈ bisimRelation
08 $(isBisim, results)$:= BisimulationsCheckfürTuple(LTS1,LTS2,r);
09 **FALLS** isBisim [Das betrachtete Tupel kann einen simularen Schritt machen]
10 **DANN**
11 **FALLS** $results \neq \emptyset$ [Das Tupel ist nicht trivial bisimular]
12 **DANN**
13 **FALLS** ¬AusreichendNachfolger($results, notbisim$)
14 **DANN** [Nicht ausreichend Folgetupel vorhanden]
15 $notbisim := notbisim \cup \{r\}$;
16 **ENDE FALLS**
17 **ENDE FALLS**
18 **SONST** [Das betrachtete Tupel konnte keinen simularen Schritt machen]
19 $notbisim := notbisim \cup \{r\}$;
20 **ENDE FALLS**
21 **ENDE FÜR**
22 $bisimRelation := bisimRelation \setminus notbisim$;
23 $endeAnzahl := count(bisimRelation)$;
24 **FALLS** startAnzahl == endeAnzahl [$\sim_n == \sim_{n+1}$]
25 **DANN**
26 finished := true;
27 **ENDE FALLS**
28 **ENDE SOLANGE**
29 **RÜCKGABE** *bisimRelation*

Wie in der klassischen Variante terminiert der Algorithmus immer und betrachtet zusätzlich ob ausreichend Folgetupel für eine Bisimulation vorhanden sind. Als Ausgabe gibt der Algorithmus die Bisimularität *bisimRelation*. Existieren keine Bisimularen Zustandspaare entfernt der Algorithmus nacheinander die Tupel und gibt am Ende eine leere Menge zurück. Alternativ könnte man auch auf den Äquivalenzklassen der Zustände arbeiten. Nun wollen wir den Hirschkoff-Algorithmus betrachten, um die Bisimularität eines Zustandspaare ohne die Berechnung von $\sim$ zu verifizieren.

5.1.4. Hirschkoff-Algorithmus

Der Hirschkoff-Algorithmus berechnet eine Bisimulation $\mathcal{R}$ auf der Grundlage eines Starttuples (α, β) mit dem Ziel $\alpha \sim \beta$ zu zeigen. Der hier vorgestellte Algorithmus ist eine Abwandlung der Klassischen Version die in [Hir99] definiert und Kapitel 3.2 dieser Ausarbeitung erläutert wurde. Die Hilfsfunktionen für den Hirschkoff-Algorithmus lauten:

HILFSFUNKTIONEN
$first : \mathcal{S} \to (\alpha, \beta)$ [Gibt das erste Element der Liste zurück]
$rest : \mathcal{S} \to (\mathcal{S} \setminus \{(\alpha, \beta)\})$ [Entfernt das erste Element der Liste und gibt Rest zurück]
$childs_1 : \mathcal{R} \to C$ [Liefert Menge $C := \{\alpha \in S_1 \mid \exists \beta \in S_2$ und $(\alpha, \beta) \in \mathcal{R}\}$]
$childs_2 : \mathcal{R} \to C$ [Liefert Menge $C := \{\beta \in S_2 \mid \exists \alpha \in S_1$ und $(\alpha, \beta) \in \mathcal{R}\}$]

Der wichtigste Bestandteil des Algorithmus ist die Liste *childList*. Diese Liste besteht aus Tupeln der Form $(Vater_i, Kinder_i)$ wobei:

- $Vater_i$ das Vatertupel $(\alpha, \beta) \in S_1 \times S_2$ ist
- $Kinder_i$ die Menge der Kindertupel ist mit $Kinder_i := \{(\alpha', \beta') \in S_1 \times S_2 \mid \exists a \in (Act_1 \cap Act_2)$ so dass $\alpha \xrightarrow{a} \alpha'$ und $\beta \xrightarrow{a} \beta'$ wobei $(\alpha, \beta) = Vater_i\}$

Das erste Element jedes Tupels ist also ein Zustandspaar, dass vorerst einen simularen Schritt machen kann. Das zweite Element des Tupels ist die jeweilige Menge der Folgetupel. Diese Liste *childList* wird benötigt, um nach jedem betrachteten Tupel eine Aussage über die Bisimularität des Vatertupels machen zu können. Bevor wir den Hirschkoff-Algorithmus selbst spezifizieren benötigen wir einen Algorithmus für die Methode Propagate.

ALGORITHMUS Propagate$(\mathcal{V}, \mathcal{R}, \mathcal{W}, childList)$
01 Boolean status := true; [Berechnung bisher in Ordnung]
02 **FÜR** $\forall\ (Vater, Kinder) \in childList$
03 **FALLS** $Vater \notin \mathcal{R}$ [Vater noch nicht als bisimular eingestuft]
04 **DANN**
05 **FALLS** $\forall Kinder \in \mathcal{V}$ [Alle Kinder wurden überprüft]
06 **DANN**
07 **FALLS** AusreichendNachfolger$(Kinder, \mathcal{W})$
08 **DANN** [Ausreichend Folgetupel vorhanden]
09 $\mathcal{R} := \mathcal{R} \cup \{Vater\}$;
10 **SONST** [Nicht ausreichend Kinder übrig]
11 $\mathcal{W} := \mathcal{W} \cup \{Vater\}$;
12 status := false; [Algorithmus-Neustart erforderlich]
13 **ENDE FALLS**
14 **ENDE FALLS**
15 **ENDE FALLS**
16 **ENDE FÜR**
17 **RÜCKGABE** $(status, \mathcal{R}, \mathcal{W})$

Propagate soll alle Listen des Hirschkoff-Algorithmus aktualisieren, sobald neue Erkenntnisse über die Bisimularität eines Tupels gewonnen wurde. Dazu übernehmen wir vom klassischen Algorithmus das status Flag, sowie die Listen $\mathcal{R}$, $\mathcal{V}$ und $\mathcal{W}$ mit einer leichten Bedeutungsänderung gegenüber der klassischen Variante (siehe Kapitel 3.2). Die Liste $\mathcal{R}$ und $\mathcal{W}$ behalten ihre jeweilige Bedeutung, wobei $\mathcal{V}$ nun alle Tupel beinhaltet die einmal überprüft wurden, egal ob sie als bisimular ($\mathcal{R}$) oder nicht-bisimular($\mathcal{W}$) eingestuft wurden. Dadurch ermöglichen wir die Überprüfung der Liste *childList*, ob sämtliche Kindertupel eines Vatertupels untersucht wurden [Zeile 05]. Der Algorithmus für Propagate erhält als Eingabe den aktuellen Stand der Listen $\mathcal{R}$, $\mathcal{V}, \mathcal{W}$ sowie *childList*. Die Listen $\mathcal{R}$, $\mathcal{W}$ sowie das status Flag werden auf Grundlage von *childList* aktualisiert und zurückgegeben.

Der Hirschkoff-Algorithmus bekommt als Eingabe zwei LTS und ein Starttupel (α, β). Die Ausgabe ist ein Boolean *isBisim* und die Bisimulation $\mathcal{R}$ die das Ergebnis bezeugt. Die Liste $\mathcal{S}$ ist die Worklist.

ALGORITHMUS Hirschkoff($LTS_1, LTS_2, \{(\alpha, \beta)\}$**)**
01 $\mathcal{W} := \emptyset$; [Initialisierung der nicht-bisimular Liste]
02 $starttuple := (\alpha, \beta)$;
03 Boolean status := false; [Die Berechnung startet neu für erste Initialisierung]
04 **SOLANGE** $\neg(starttuple \in \mathcal{W}$ oder status)
05 **FÜHRE AUS** [Wiederhole falls Starttupel nicht in $\mathcal{W}$ und Berechnung fehlerhaft]
06 $\mathcal{R} := \emptyset$; $\mathcal{V} := \emptyset$; [Listen $\mathcal{R}$ und $\mathcal{V}$ initialisieren]
07 status := true;
08 $childList := \emptyset$
09 $\mathcal{S} := \{starttuple\}$;
10 **SOLANGE** $\mathcal{S} \neq \emptyset$ und status
11 **FÜHRE AUS**
12 $tuplecheck := first(\mathcal{S})$;
13 $\mathcal{S} := rest(\mathcal{S})$;
14 **FALLS** $tuplecheck \notin \mathcal{R}$ und $tuplecheck \notin \mathcal{W}$
15 **DANN** [Das zu untersuchende Paar ist nicht nicht eingestuft]
16 $(isBisimT, results)$:= BisimulationsCheckfürTuple(LTS1,LTS2,tuplecheck);
17
18 **FALLS** isBisimT und $results == \emptyset$
19 **DANN** [Ehemals succeeds-Fall]
20 $\mathcal{V} = \mathcal{V} \cup tuplecheck$;
21 $\mathcal{R} = \mathcal{R} \cup tuplecheck$;
22 (status,$\mathcal{R}$,$\mathcal{W}$) := Propagate($\mathcal{V}$,$\mathcal{R}$,$\mathcal{W}$,*childList*);
23 **ENDE FALLS**
24
25 **FALLS** ¬isBisimT
26 **DANN** [Ehemals fails-Fall]
27 $\mathcal{V} = \mathcal{V} \cup tuplecheck$;
28 $\mathcal{W} = \mathcal{W} \cup tuplecheck$;
29 (status,$\mathcal{R}$,$\mathcal{W}$) := Propagate($\mathcal{V}$,$\mathcal{R}$,$\mathcal{W}$,*childList*);
30 **ENDE FALLS**

```
31          FALLS isBisimT und results ≠ ∅
32          DANN [Ehemals neither succeeds nor fails Fall]
33            FALLS tuplecheck ∉ V
34            DANN
35              V = V ∪ tuplecheck;
36              FALLS AusreichendNachfolger(results, W)
37              DANN [Ausreichend Folgetupel vorhanden]
38                childList := childList ∪ {(tuplecheck, results)};
39                S := S ∪ (results \ W);
40              SONST [Nicht ausreichend Kinder übrig]
41                W := W ∪ {tuplecheck};
42              ENDE FALLS
43            ENDE FALLS
44            (status,R,W) := Propagate(V,R,W,childList);
45          ENDE FALLS
46
47        ENDE FALLS
48      ENDE SOLANGE
49  ENDE SOLANGE
50  FALLS starttupel ∉ W und status
51  DANN
52      isBisim := true;
53  SONST
54      isBisim := false;
55      R := {starttupel};
56  ENDE FALLS
57  RÜCKGABE (isBisim,R)
```

Der GOTO Befehl im klassischen Hirschkoff-Algorithmus ist durch die SOLANGE-Schleife realisiert [Zeile 4-49]. Im Anschluss werden die Listen initialisiert und die Worklist wird abgearbeitet [ab Zeile 10]. Die Adaption an die klassische Variante ist leicht ersichtlich:

- Verhalten im Falle eines trivial bisimularen Zustandspaares [Zeilen 18-23]
- Verhalten im Falle eines nicht-bisimularen Zustandspaares [Zeilen 25-30]
- Gewinnung der interessanten Folgetupel [31-45]

Ähnlich dem Fixpunkt-Algorithmus, werden sowohl in der Methode Propagate [Propagate Zeilen 7-12] als auch im Hirschkoff-Algorithmus [Zeilen 36-42], die betrachteten Tupel auf ihre Folgetupel untersucht. Der Algorithmus baut auf Grundlage des *starttuples* die Bisimulation $\mathcal{R}$ immer weiter auf, wobei er jedes Tupel das einmal in der Worklist $\mathcal{S}$ gespeichert wurde untersucht. Als bisimular (in $\mathcal{R}$ liegend), wird ein Zustandspaar dann eingeordnet wenn ausreichend Kinder betrachtet wurden, und eine Teil-Bisimulation berechnet werden konnte. Die Liste *childList* speichert die Zuordnung der Vaterpaare zu ihren Kindern und ist somit die datenreichste Liste während der Berechnung.

Beispiel 5.1.1 (Algorithmus Hirschkoff) *Seien die Transitionssysteme in Abbildung 5.1 gegeben und sei* $\alpha = 1, \beta = 5$.

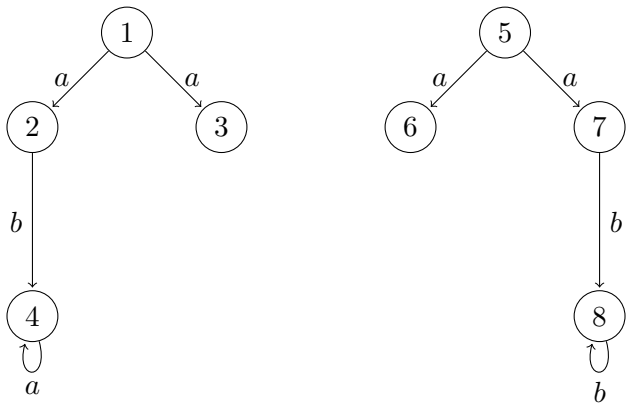

Abbildung 5.1.: Hirschkoff Beispiel

Der Hirschkoff-Algorithmus beginnt mit

- $\mathcal{V} = \emptyset$
- $\mathcal{R} = \emptyset$
- $\mathcal{W} = \emptyset$
- $\mathcal{S} = \{(1, 5)\}$

Der Algorithmus überprüft ob es ein simularer Schritt existiert und berechnet alle Nachfolger. Das Paar $(1, 5)$ *wird in* $\mathcal{V}$ *gespeichert da es betrachtet wurde, jedoch noch nicht in* $\mathcal{R}$ *oder* $\mathcal{W}$ *weil noch keine sichere Aussage über die Bisimularität des Paares gemacht werden kann. Die Nachfolgepaare werden in die Worklist aufgenommen.*

- $\mathcal{V} = \{(1, 5)\}$
- $\mathcal{R} = \emptyset$
- $\mathcal{W} = \emptyset$
- $\mathcal{S} = \{(2, 6), (2, 7), (3, 6), (3, 7)\}$

Das Paar $(2, 6)$ *wird untersucht und es kann festgestellt werden das dieses Paar nicht bisimular ist. Das Paar wird sofort in* $\mathcal{W}$ *und in* $\mathcal{V}$ *aufgenommen, da es nicht bisimular ist und untersucht wurde. Da noch nicht ausreichend Kinder des Tupels* $(1, 5)$ *untersucht wurden bleibt es immernoch uneingestuft.*

- $\mathcal{V} = \{(1, 5), (2, 6)\}$
- $\mathcal{R} = \emptyset$
- $\mathcal{W} = \{(2, 6)\}$
- $\mathcal{S} = \{(2, 7), (3, 6), (3, 7)\}$

Nun wird das Paar $(2,7)$ *betrachtet und es wird ein simularer Schritt gefunden. Das Paar wird in* $\mathcal{V}$ *aufgenommen und das berechnete Kindtupel* $(4,8)$ *in die Worklist* $\mathcal{S}$ *aufgenommen. Es existieren immernoch Kinder von* $(1,5)$ *die noch nicht betrachtet wurden und so bleibt dieses Paar immernoch uneingestuft.*

- $\mathcal{V} = \{(1,5),(2,6),(2,7)\}$
- $\mathcal{R} = \emptyset$
- $\mathcal{W} = \{(2,6)\}$
- $\mathcal{S} = \{(3,6),(3,7),(4,8)\}$

Als nächstes folgt das Paar $(3,6)$ *welches als Trivial bisimular eingestuft wird und daher sofort nach* $\mathcal{V}$ *und* $\mathcal{R}$ *verschoben wird.*

- $\mathcal{V} = \{(1,5),(2,6),(2,7),(3,6)\}$
- $\mathcal{R} = \{(3,6)\}$
- $\mathcal{W} = \{(2,6)\}$
- $\mathcal{S} = \{(3,7),(4,8)\}$

Mit dem Paar $(3,7)$ *wird das letzte Kindstupel von* $(1,5)$ *überprüft. Das Zustandspaar* $(3,7)$ *wandert nach* $\mathcal{W}$*.*

- $\mathcal{V} = \{(1,5),(2,6),(2,7),(3,6),(3,7)\}$
- $\mathcal{R} = \{(3,6)\}$
- $\mathcal{W} = \{(2,6),(3,7)\}$
- $\mathcal{S} = \{(4,8)\}$

Nun überprüft die Methode Propagate $(1,5)$ *auf eine mögliche Bisimularität da alle Nachfolger betrachtet wurden. Die verbleibenden und nicht in* $\mathcal{W}$ *liegenden Tupel sind* $(2,7)$ *und* $(3,6)$*, daher wird* $(1,5)$ *mit genügend Nachfolgern als Bisimular eingestuft und nach* $\mathcal{R}$ *kopiert.*

- $\mathcal{V} = \{(1,5),(2,6),(2,7),(3,6),(3,7)\}$
- $\mathcal{R} = \{(3,6),(1,5)\}$
- $\mathcal{W} = \{(2,6),(3,7)\}$
- $\mathcal{S} = \{(4,8)\}$

Das letzte Tupel $(4,8)$ *wird überprüft und nach* $\mathcal{V}$ *und* $\mathcal{W}$ *verschoben, da es keinen simularen Schritt ausführen kann.*

- $\mathcal{V} = \{(1,5),(2,6),(2,7),(3,6),(3,7),(4,8)\}$
- $\mathcal{R} = \{(3,6),(1,5)\}$
- $\mathcal{W} = \{(2,6),(3,7),(4,8)\}$
- $\mathcal{S} = \emptyset$

Es wurden ausreichend Kinder des Tupels $(2,7)$ *überprüft. Leider sind nicht genügend Nachfolger vorhanden, da das einzige Nachfolgepaar als nicht-bisimular eingestuft wurde. Somit wird* $(2,7)$ *nach* $\mathcal{W}$ *verschoben und der Algorithmus muss neu starten.*

- $\mathcal{V} = \emptyset$
- $\mathcal{R} = \emptyset$
- $\mathcal{W} = \{(2,6),(3,7),(4,8),(2,7)\}$
- $\mathcal{S} = \{(1,5)\}$

Im zweiten Durchlauf des Algorithmus berechnen wir wieder die Nachfolger des Paares $(1,5)$*. Aus dem vorherigen Durchlauf haben wir die Liste der nicht-bisimularen Zustandspaare* $\mathcal{W}$ *übernommen. Nach ein paar Schritten gelangen wir zu folgendem Zwischenstand.*

- $\mathcal{V} = \{(1,5),(2,6),(2,7),(3,6),(3,7)\}$
- $\mathcal{R} = \{(3,6)\}$
- $\mathcal{W} = \{(2,6),(3,7),(4,8),(2,7)\}$
- $\mathcal{S} = \{(4,8)\}$

Das letzte Kind von $(1,5)$ *wurde überprüft und Propagate liefert diesmal die Aussage, dass nicht genügend Nachfolger vorhanden sind um für* $(1,5)$ *eine Bisimulation zu bauen. Das Tupel* $(1,5)$ *wird nach* $\mathcal{W}$ *verschoben und der Algorithmus terminiert mit der Aussage das dieses Paar nicht bisimular ist.*

5.2. Algorithmen für Graphen

Nun wollen wir Algorithmen für die Graphtransformation mit Borrowed Context definieren. Der erste Schritt besteht in der Realisierung eines Algorithmus zur Berechnung aller partiellen Matchings. Die Graphenbezeichnungen sind gemäß der in Kapitel 4.5 vorgestellten Abbildung 4.12. Die Bezeichnung eines Morphismus wird durch die Graphen bestimmt zwischen denen er abbildet. So ist der Morphismus gl stets der Morphismus $gl = (gl_V, gl_E) : G \to L$ welcher die Abbildung der Graphen $G = (V_G, E_G, s_G, t_G)$ und $L = (V_L, E_L, s_L, t_L)$ beschreibt und möglicherweise partiell ist. Morphismen werden mit Kleinbuchstaben und Graphen mit Großbuchstaben beschrieben. Da wir auf injektiven Morphismen arbeiten lassen sich Morphismen invertieren und wir schreiben $gl^{-1} = lg$ für den invertierten Morphismus. Der invertierte Morphismus kann partiell sein.

5.2.1. Partielles Matching

Um alle Borrowed Context für einen Graphen $J \to G$ bezüglich einem Regelsatz, mit Regeln der Form $(L \leftarrow I \rightarrow R)$ zu bestimmen, müssen wir zunächst die partiellen Matchings $(D, D \to L, D \to G)$ bestimmen. Da Knoten und Kanten gleiche Labels haben können, benötigen wir eindeutige Identifier für die Graphenelemente. Daher verwenden wir ab sofort Knoten und Kanten IDs mit denen die Elemente eindeutig identifiziert werden können. Hilfsfunktionen sind für das Finden von partiellen Matchings:

HILFSFUNKTIONEN
$invertMorph : f \to f^{-1}$ [Invertiert den Morphismus f]
$ID_{V_G} : V_G \to \mathbb{N}$ [Gibt die ID des Knoten v $\in V_G$ zurück]
$ID_{E_G} : E_G \to \mathbb{N}$ [Gibt die ID der Kante e $\in E_G$ zurück]
$tuples : (Set_1, \ldots, Set_n) \to Set_1 \times \ldots \times Set_n$ [Berechne das Kreuzprodukt]
$legalMorphism : G \times (gl : G \to L) \times L \to Bool$ [True falls gl ein gültiger Morphismus ist]
$restrictToEdges : G \times IDListe \to G'$ [Behalte in G nur Kanten aus der ID-Liste]
$restrictToVertices : G \times IDListe \to G'$ [Behalte in G nur Knoten aus der ID-Liste]
$calculatePMonos_V : G \times L \to Set$ [Gibt alle partiellen Knoten Matchings $V_G \to V_L$]
$calculatePMonos_E : G \times L \to Set$ [Gibt alle partiellen Kanten Matchings $E_G \to E_L$]

Der Algorithmus zur Berechnung von (D, dl, dg) benötigt eine Vorverarbeitung in Form einer Berechnung des Morphismus $gl = (glList_V, glList_E) : G \to L$. Hierfür definieren wir die Methode $BerechneAlleMonoMorphismen$. Die Grundidee des Partiellen-Matching-Algorithmus ist die Berechnung eines partiellen injektiven Morphismus $gl : G \to L$ und aus diesem den Cospan $(G \leftarrow D \to L)$ abzuleiten.

ALGORITHMUS BerechneAlleMonoMorphismen(L, G)
01 Set $gl_V := \emptyset$; [Menge der injektiven $glList_V$ Tupel]
02 Set $gl_E := \emptyset$; [Menge der injektiven $glList_E$ Tupel]
03 Set $glSet := \emptyset$ [Menge der Monomorphismen gl]
04
05 $gl_V := calculateMonos_V(G, L)$; [Berechnung aller partiellen Knoten Matchings]
06 $gl_E := calculateMonos_E(G, L)$; [Berechnung aller partiellen Kanten Matchings]
07
08 $glSet := tuples(\ (gl_V, gl_E)\ )$; [Liste der vollständigen Mono-Morphismen]
09 **FÜR** $\forall\ gl \in glSet$
10 **FALLS** $\neg legalMorphism(G, gl, L)$
11 **DANN** [Falls eine Kondition nicht beachtet → lösche gl]
12 $glSet := glSet \setminus \{gl\}$
13 **ENDE FALLS**
14 **ENDE FÜR**
15 **RÜCKGABE** $glSet$

Wir beginnen mit Mengen disjunkter Listen für die Abbildung der Knoten [Zeile 5] und Kanten [Zeile 6]. Dabei wird verlangt, dass die beiden Elemente der Abbildung dasselbe Label besitzen. Mit der Methode *tupel* können wir nun die Morphismen $gl : G \to L$ generieren, von denen wir noch jene löschen die entweder die Dangling-Edge oder die Source/Target Bedingung nicht erfüllen und damit keine legitimen Morphismen sind [Zeilen 8-14]. Der Algorithmus liefert uns eine Menge $glSet$ von möglichen Morphismen $gl : G \to L$ die für die Gewinnung des Partiellen Matchings (D, dl, dg) genutzt werden können.

Als Eingabe erhält der PartialMatch-Algorithmus die Graphen G und L. Wir versuchen als erstes alle partiellen Matchings gl zu berechnen die direkt von G auf L abbilden. Anhand dieser Morphismen berechnen wir im Anschluss die Tupel (D, dl, dg), die als Eingabe für kommende Algorithmen gebraucht werden.

ALGORITHMUS PartialMatch(L, G)
01 Set $matchresult := \emptyset$ [Menge der partiellen Matching-Tripel (D, dl, dg)]
02 Set $glSet :=$ BerechneAlleMonoMorphismen(L, G);
03 **FÜR** $\forall\ gl \in glSet$
04 $\quad dl := gl \setminus \emptyset$; $[dl = (dlList_V, dlList_E)]$
05 $\quad dg := invertMorph(gl) \circ dl$;
06 $\quad D' := restrictToEdges(L, dlList_E)$; [Lösche nicht getroffene Kanten]
07 $\quad D := restrictToVertices(D', dlList_V)$; [Lösche nicht getroffene Knoten]
08 $\quad matchresult := matchresult \cup \{(D, dl, dg)\}$;
09 **ENDE FÜR**
10 **RÜCKGABE** *matchresult*

Mit den direkten Teilmorphismen $gl : G \to L$ gelingt die Berechnung der partiellen Matching-Tripel (D, dl, dg). Der Graph D ist ein Teilgraph von L der alle Knoten und Kanten besitzt die vom Morphismus $gl : G \to L$ getroffen wurden [Zeilen 6-7]. Der Morphismus von $D \to L$ zeigt somit auf den identischen Teilgraphen in L. Daher reicht es, aus dem Morphismus $gl : G \to L$ die Platzhalter in Form von $\emptyset$ zu entfernen und den übrig gebliebenen Morphismus zu verwenden [Zeile 4]. Diesen Morphismus $dl : D \to L$ können wir mit dem invertierten Morphismus von $gl : G \to L$ verketten um den eindeutigen Morphismus $dg : D \to G$ zu gewinnen [Zeile 5].

Beispiel 5.2.1 (Algorithmus Partielles Matching) *Wir beginnen mit der Berechnung der Menge der partiellen Matchings. Ein mögliches partielles Matching $gl : G \to L$ ist in Abbildung 5.2 dargestellt.*

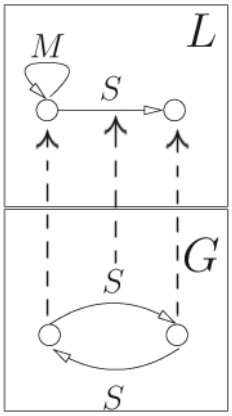

Abbildung 5.2.: Partielles Matching Morphismus $gl : G \to L$

Der Graph D unseres partiellen Matching Tripels (D, dl, dg) ist der vom Matching getroffene Teilgraph des Graphen G. Als Morphismus $dl : D \to L$ wählen wir die Identität des Teilgraphen. Der Morphismus $dg : D \to G$ wird durch die Verkettung von $dl : D \to L$ mit der inversen von $gl : G \to L$. Das Resultat ist in Abbildung 5.3 abgebildet.

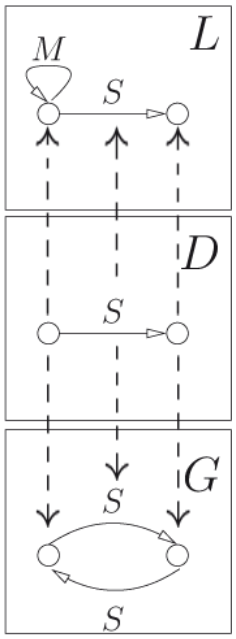

Abbildung 5.3.: Partielles Matching Cospan $G \leftarrow D \rightarrow L$

Mit der Menge der partiellen Matchings gelingt nun der nächste Schritt in Form der Berechnung aller Borrowed Context. Zunächst wollen wir die gefundene Menge der partiellen Matchings jedoch reduzieren. Dies geschieht mit der vorgestellten Optimierung des Prunings.

5.2.2. Pruning

Die in Kapitel 4.6.1 vorgestellte Optimierung des Pruning, wollen wir nun in einem Algorithmus beschreiben, um die für den Borrowed Context relevanten partiellen Matchings zu bestimmen. Der Algorithmus erhält als Eingabe die Morphismen $il : L \leftarrow I$ und $jg : J \rightarrow G$, sowie die Menge der partiellen Matching-Tripel *matchresult*.

ALGORITHMUS Pruning($matchresult, il, jg$**)**
01 **FÜR** $\forall\ (D, dl, dg) \in matchresult$
02 $djSet := \{dj : D \rightarrow J \mid invertMorph(dg) \circ jg \circ dj == id_D\}$;
03 $diSet := \{di : D \rightarrow I \mid invertMorph(dl) \circ il \circ di == id_D\}$;
04 **FALLS** $djSet \neq \emptyset$ oder $diSet \neq \emptyset$
05 **DANN** [Sofern einer der Morphismen di oder dj existiert, lösche das Tripel]
06 $matchresult := matchresult \setminus \{(D, dl, dg)\}$
07 **ENDE FALLS**
08 **ENDE FÜR**
09 **RÜCKGABE** *matchresult*

Als nächsten Schritt berechnen wir alle Borrowed Context $(I \rightarrow F \leftarrow K)$ sowie die resultierenden Graphen H mit ihren Interfaces K $(K \rightarrow H)$.

5.2.3. Borrowed Context

Zur Berechnung der LTS benötigen wir einen Algorithmus der für den Graphen G mit Interface J ($J \to G$) und dem gefundenen partiellen Matching $(D, \to, \downarrow)$ zu einer Regel $(L \leftarrow I \to R)$, den Borrowed Context $(J \to F \leftarrow K)$ berechnet. Die benötigten Hilfsfunktionen wurden aus einem Projekt von der Technischen Universität Berlin übernommen. Bennenungen der Variablen halten sich an die Pushout/Pullback-Konstruktionen die auf den Abbildungen 4.1 und 4.4 in Kapitel 4.1/4.2 vorgestellt wurden.

HILSFUNKTIONEN
$pushout : A \times f \times B \times g \times C \to D \times g' \times f'$ [Aus $(A, \to, B, \downarrow, C)$ berechne $(D, \downarrow, \to)$]
$pushoutC : A \times f \times B \times g' \times D \to C \times g \times f'$ [Aus $(A, \to, B, \downarrow, D)$ berechne $(C, \downarrow, \to)$]
$pullback : A \times g \times C \times f \times B \to D \times f' \times g'$ [Aus $(A, \uparrow, B, \leftarrow, C)$ berechne $(D, \leftarrow, \uparrow)$]

Die Eingabe des BorrowedContext-Algorithmus besteht aus dem Tripel ($(J \to G)$, $(L \leftarrow I \to R)$, $(D, \to, \downarrow)$). Als Ausgabe erwarten wir das Tupel ($(J \to F \leftarrow K)$, $(K \to H)$). Dabei arbeitet der Algorithmus die in Kapitel 4.5 beschriebenen Pushout und Pullback Schritte nacheinander ab.

ALGORITHMUS BerechneBorrowedContext$((J, jg, G), (L, il, I, ir, R), (D, dl, dg))$
01 $(G^+, lg, gg^+) := pushout(D, dl, L, dg, G)$;
02 $(C, ic, cg^+) := pushoutC(I, il, L, lg^+, G^+)$;
03 $(H, rh, ch) := pushout(I, ir, R, ic, C)$;
04 $(F, jf, fg^+) := pushoutC(J, jg, G, gg^+, G^+)$;
05 $(K, kf, kc) := pullback(G^+, fg^+, F, cg^+, C)$;
06
07 $kh := ch \circ kc$;
08 **RÜCKGABE** $((J, jf, F, kf, K), (K, kh, H))$

Der Algorithmus arbeitet die Schritte zur Gewinnung des Borrowed Context ab [Zeilen 1-5]. Daber verwendet er auch den Double Pushout Ansatz der jedoch keine eigene Methode darstellt [Zeilen 2-3]. Als letzter Schritt wird der Morphismus $kh : K \to H$ durch die Verkettung der Morphismen $kc : K \to C$ und $ch : C \to H$ berechnet [Zeile 7]. Bevor wir die berechneten Borrowed Context und die resultierenden Graphen für einen Bisimulations-Check verwenden, definieren wir einen Algorithmus der die Isomorphie zweier Graphen überprüft. Das ist notwendig um zu Überprüfen ob zwei Borrowed Context, ein identisches Tansitionslabel beschreiben.

5.2.4. Isomorphismus

Der Isomorphie-Check überprüft gegebene Borrowed Context Quintupel (J, jf, F, kf, K) auf Gleichheit, um eine Aussage für den Bisimulations-Check-Algorithmus zu gewinnen. Der Bisimulations-Check soll im folgenden Überprüfen ob ein Graphenpaar $(J \to G, J \to G')$ bisimular ist. Die Borrowed Context $(J, \to, F, \leftarrow, K)$ und $(J, \to, F', \leftarrow, K')$ sind nur dann als gleich anzusehen sofern Isomorphismen $f_{Iso} : F \to F'$ und $k_{Iso} : K \to K'$ existieren, so dass das Diagramm 5.4 kommutiert und das Quadrat einen Pushout beschreibt.

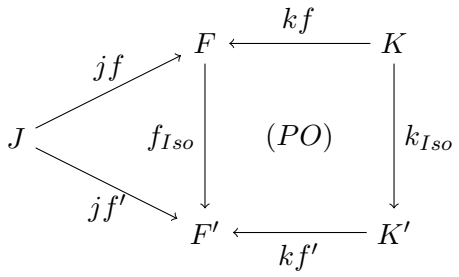

Abbildung 5.4.: Borrowed Context Isomorphie

Der erste Schritt um die Gleichheit der Borrowed Context zu zeigen liegt in der Berechnung des Isomorphismus $f_{Iso} : F \to F'$. Wichtig hierbei die Abbildung so zu definieren, dass die (bereits durch das Interface J) festgelegten Morphismen $jf : J \to F$ und $jf' : J \to F'$ beachtet werden. Für diesen Algorithmus existieren Hilfsfunktionen:

HILFSFUNKTIONEN
$length : Liste \to \mathbb{N}$ [Gibt die Anzahl der Elemente in der Liste zurück]
$getIdentity : F \to f$ [Gibt die sortierten Listen der Knoten und Kanten IDs]
$getLabelsV : Liste \times X \to V_X List$ [Gibt die Knoten Label der IDs im Graph X]
$getLabelsE : Liste \times X \to E_X List$ [Gibt die Kanten Label der IDs im Graph X]
$permutate : Liste \to Set$ [Gibt eine Menge mit allen Permutationen der Liste zurück]
$tuples : (Set_1, \ldots, Set_n) \to Set_1 \times \ldots \times Set_n$ [Berechne das Kreuzprodukt]
$soutar : f \times F \times F' \to Boolean$ [True falls source und target nicht vertauscht]

Die Morphismen sind wieder als Tupel mit $f = (fList_V, fList_E)$ definiert, wobei $fList_V$ und $fList_E$ Listen mit IDs definieren. Die Hilfsfunktion *getIdentity* berechnet den Identitätsmorphismus. Diese ID-Listen lassen sich mit der Methode *getLabels* auf eine sortierte Liste von zugehörigen Labels abbilden. Als ersten Schritt stellen wir nun einen Algorithmus zur Verfügung, der für die Eingabe des Quintupels (J, jf, jf', F, F') eine Menge von Isomorphismen $f_{Iso} : F \to F'$ berechnet.

ALGORITHMUS BerechneIsomorphismen(J, jf, jf', F, F')
01 $(fList_V, fList_E) := getIdentity(F)$; [Berechne die Identität von F]
02 $(f'List_V, f'List_E) := getIdentity(F')$; [Berechne die Identität von F']
03 **FALLS** $length(fList_V) \neq length(f'List_V)$ oder $length(fList_E) \neq length(f'List_E)$
04 **DANN** [Falls die Anzahl der Knoten oder Kanten verschieden ist]
05 **RÜCKGABE** $\emptyset$ [Kein Isomorphismus möglich]
06 **ENDE FALLS**
07
08 $isobase_V := permutate(fList_V)$ [Menge aller Knoten ID Permutationen]
09 $isobase_E := permutate(fList_E)$ [Menge aller Kanten ID Permutationen]
10
11 **FÜR** $\forall\ f_{Iso}List_V \in isobase_V$
12 **FALLS** $getLabelsV(f_{IsoList_V, F'}) \neq getLabelsV(fList_V, F)$
13 **DANN** [Die Labels des Knoten Isomorphismus stimmen nicht überein]
14 $isobase_V := isobase_V \setminus \{f_{Iso}List_V\}$ [Löschen dieser Permutation]
15 **ENDE FALLS**
16 **ENDE FÜR**

17 **FÜR** $\forall\ f_{Iso}List_E \in isobase_E$
18 **FALLS** $getLabelsE(f_{IsoList_E,F'}) \neq getLabelsE(fList_E, F)$
19 **DANN** [Die Labels des Kanten Isomorphismus stimmen nicht überein]
20 $isobase_E := isobase_E \setminus \{f_{Iso}List_E\}$ [Löschen dieser Permutation]
21 **ENDE FALLS**
22 **ENDE FÜR**
23
24 $f_{Iso}Set := tuples((isobase_V, isobase_E))$; [Menge aller Isomorphismen f_{Iso}]
25
26 **FÜR** $\forall\ f_{Iso} \in f_{Iso}Set$
27 **FALLS** $\neg soutar(f_{Iso}, F, F')$
28 **DANN** [Source und Target wurden beim Abbilden verdreht]
29 $f_{Iso}Set := f_{Iso}Set \setminus \{f_{Iso}\}$;
30 **ENDE FALLS**
31 **ENDE FÜR**
32
33 **FALLS** $(J, jf, jf') \neq (\emptyset, \emptyset, \emptyset)$
34 **DANN** [Falls Graph J und Morphismen $jf : J \to F, jf' : J \to F'$ gegeben]
35 **FÜR** $\forall\ f_{Iso} \in f_{Iso}Set$
36 **FALLS** $f_{Iso} \circ jf \neq jf'$
37 **DANN** [Isomorphismus f_{Iso} beachtet nicht den vordefinierten Morphismus]
38 $f_{Iso}Set := f_{Iso}Set \setminus \{f_{Iso}\}$;
39 **ENDE FALLS**
40 **ENDE FÜR**
41 **ENDE FALLS**
42
43 **RÜCKGABE** $f_{Iso}Set$

Zum Anfang überprüfen wir ob die Knoten und Kantenanzahl, der möglicherweise isomorphen Graphen übereinstimmt [Zeilen 3-6]. Anschließend beginnt der Algorithmus mit der Aufstellung permutierter Identitätsmorphismen, da diese als Isomorphismen auf andere Graphen genutzt werden können [Zeilen 8-9]. Dabei müssen die abgebildeten Elemente mit ihren getroffenen Knoten und Kanten Labeln übereinstimmen [Zeilen 11-16 und 17-22]. Wir kombinieren alle gefundenen Knoten und Kanten Abbildungen zu vollständigen Isomorphismen. Als nächstes überprüfen wir ob die Quellknoten und Zielknoten der Kanten richtig abgebildet wurden, ansonsten entfernen wir sie aus der Menge der möglichen Isomorphismen [Zeilen 26-31]. An der folgenden Stelle halten wir unseren Algorithmus möglichst flexibel. Sofern wie in unserem Fall zusätzlich der Graph J mit seinen Morphismen $jf : J \to F, jf' : J \to F'$ existiert, verwerfen wir alle gefundenen Isomorphismen die diesen festgelegten Abbildungen nicht genügen [Zeilen 33-41]. Sofern kein Graph und Morphismen definiert sind, berechnet der Algorithmus sämtliche Isomorphismen zwischen den Graphen F und F'. Die nun berechneten Isomorphismen $f_{Iso} : F \to F'$, müssen auf ihre Verträglichkeit mit den Isomorphismen $k_{Iso} : K \to K'$ überprüft werden. Dazu definieren wir den Algorithmus *GleicherBorrowedContext* der die Isomorphismen berechnet und einen Boolean zurück gibt, der angibt ob die untersuchten Borrowed Context dasselbe Transitionslabel beschreiben.

ALGORITHMUS GleicherBorrowedContext$((J, jf, F, kf, K), (J, jf', F', kf', K'))$
01 $f_{Iso}Set := BerechneIsomorphismen(J, jf, jf', F, F')$; [Isomorphismen f_{Iso}]
02 $k_{Iso}Set := BerechneIsomorphismen(\emptyset, \emptyset, \emptyset, K, K')$; [Menge der Isomorphismen k_{Iso}]
03 **FÜR** $\forall\ f_{Iso} \in f_{Iso}Set$
04 **FÜR** $\forall\ k_{Iso} \in k_{Iso}Set$
05 **FALLS** $f_{Iso} \circ kf == kf' \circ k_{Iso}$
06 **DANN**
07 **RÜCKGABE** true
08 **ENDE FALLS**
09 **ENDE FÜR**
10 **ENDE FÜR**
11 **RÜCKGABE** false

Um zu überprüfen ob die Borrowed Context gleich sind, berechnen wir zuerst alle möglichen Isomorphismen $f_{Iso} : F \to F'$ und $k_{Iso} : K \to K'$. Bei $f_{Iso} : F \to F'$ beachten wir die mitgelieferten Morphismen $jf : J \to F, jf' : J \to F'$ [Zeile 1], während wir bei $k_{Iso} : K \to K'$ alle Möglichkeiten ohne Einschränkungen berechnen [Zeile 2]. Im Anschluss überprüfen wir ob die Morphismen einen eindeutigen Pushout beschreiben und geben true zurück, falls es diesen Pushout gibt. Wir sind nun bereit einen Algorithmus zu definieren der für ein gegebenes Paar von Graphen $(J \to G, J \to G')$ und eine Menge von Regeln *Ruleset* überprüft, ob $(J \to G) \sim (J \to G')$ gilt.

5.2.5. Bisimulations-Check

Wir beginnen mit der Definition eines Algorithmus BisimulationsCheckFürGraphTupel der überprüft ob ein gegebenes Paar $(J \to G, J \to G')$ zu einer Regelmenge *Ruleset*, bisimulare Schritte durchführen kann. Im Anschluss wollen wir den Algorithmus erweitern um komplette Relationen zu überprüfen und sogar selbst welche aufzubauen. Als Eingabe benötigen wir das Graphenpaar und die Regelmenge. Das Paar von Graphen kann auch als Cospan $G \leftarrow J \to G'$ angesehen werden, wobei wir als Übergabereihenfolge J, jg, G, jg', G' festlegen werden. Als Rückgabe erwarten wir wie auch beim Bisimulationscheck für Transitionssysteme einen Boolean *isBisim*, der angibt ob ein simulierender Schritt möglich ist, sowie eine Menge an Nachfolgepaaren (K, kh, H, kh', H'). Die Bisimularität der Nachfolgepaare wird in einem Algorithmus *BisimulationsCheckGraphKomplett* überprüft, mit dem wir komplette Bisimulationen $\mathcal{R}$ bauen werden, um die Bisimularität von $(J \to G) \sim (J \to G')$ zu zeigen. Die Algorithmen dieses Kapitels sind eine Adaption der Algorithmen für Transitionssysteme (siehe Kapitel 5.1.1).

ALGORITHMUS BisimulationsCheckGraphTupel$((J, jg, G, jg', G'), Ruleset)$
01 $Tupelset := \{(J, jg, G, jg', G'), (J, jg', G', jg, G)\}$;
02 $MatchingsG := \emptyset$; [Menge der Partiellen Matching Tripel (D, dl, dg) von G und L]
03 $MatchingsG' := \emptyset$; [Menge der Partiellen Matching Tripel (D, dl, dg) von G' und L]
04 Boolean $isBisim := true$; [Beschreibt ob ein simulierender Schritt möglich ist]
05 $result := \emptyset$; [Menge der Nachfolgepaare (K, kh, H, kh', H')]

06 **FÜR** $\forall\ (J, jg, G, jg', G') \in Tupelset$
07 $AllBorrowedContextG := \emptyset$; [Menge der Borrowed Context Schritte von G]
08 $AllBorrowedContextG' := \emptyset$; [Menge der Borrowed Context Schritte von G']
09 **FÜR** $\forall\ (L, il, I, ir, R) \in Ruleset$
10 $MatchingG := PartialMatch(L, G)$;
11 $MatchingG := Pruning(MatchingG, il, jg)$; [Pruning der Tripel]
12 $MatchingG' := PartialMatch(L, G')$;
13
14 **FÜR** $\forall\ (D, dl, dg) \in MatchingG$
15 $((J, jf, F, kf, K), (K, kh, H)) :=$ BerechneBorrowedContext$((J, jg, G), (L, il, I, ir, R), (D, dl, dg))$;
16 $AllBorrowedContextG :=$ $AllBorrowedContextG \cup \{((J, jf, F, kf, K), (K, kh, H))\}$;
17 **ENDE FÜR**
18
19 **FÜR** $\forall\ (D, dl, dg') \in MatchingG$
20 $((J, jf', F', kf', K'), (K', kh', H')) :=$ BerechneBorrowedContext$((J, jg, G'), (L, il, I, ir, R), (D, dl, dg'))$;
21 $AllBorrowedContextG' :=$ $AllBorrowedContextG' \cup \{((J, jf', F', kf', K'), (K', kh', H'))\}$;
22 **ENDE FÜR**
23 **ENDE FÜR**
24
25 **FÜR** $\forall\ ((J, jf, F, kf, K), (K, kh, H)) \in AllBorrowedContextG$
26 Boolean simulates := false
27 **FÜR** $\forall\ ((J, jf', F', kf', K'), (K', kh', H')) \in AllBorrowedContextG'$
28 **FALLS** GleicherBorrowedContext$((J, jf, F, kf, K), (J, jf', F', kf', K'))$
29 **DANN**
30 simulates := true;
31 **FALLS** $(K, kh, H, kh', H') \notin results$ und $(K, kh', H', kh, H) \notin results$
32 **DANN**
33 $results := results \cup \{(K, kh, H, kh', H')\}$; [Merken des Nachfolgepaares]
34 **ENDE FALLS**
35 **ENDE FALLS**
36 **ENDE FÜR**
37 **FALLS** ¬simulates
38 **DANN** [Für $((J, jf, F, kf, K), (K, kh, H))$ kein simulierender Schritt]
39 isBisim := false; [Graphenpaar kann nicht bisimular sein]
40 **RÜCKGABE** (isBisim,$\{(J, jg, G, jg', G')\}$);
41 **ENDE FALLS**
42 **ENDE FÜR**
43 **ENDE FÜR**
44 **RÜCKGABE** (isBisim,$results$)

Der Algorithmus berechnet zunächst für ein Graphenpaar $(J \to G, J \to G')$, alle partiellen Matchings bezüglich der Regeln in der Regelmenge *Ruleset* [Zeilen 10-12]. Dabei wird die Menge der partiellen Matching Tripel (D, dl, dg) des simulierenden Graphen gepruned. Als nächstes werden alle möglichen Borrowed Context Schritte für die Graphen $J \to G$ und $J \to G'$ berechnet und in den Mengen $AllBorrowedContextG$ [Zeilen 14-17] und $AllBorrowedContextG'$ [Zeilen 19-22] gespeichert. Für jeden Borrowed Context Schritt $(J \to G) \overset{J \to F \leftarrow K}{\to} (K \to H)$, muss es nun einen gleichen Borrowed Context Schritt $(J \to G') \overset{J \to F \leftarrow K}{\to} (K \to H')$ geben [Zeilen 25-42]. Das Paar $(K \to H, K \to H')$ wird in die Ergebnisliste *results* aufgenommen, falls es noch nicht vorhanden ist [Zeilen 31-34]. Sofern ein Borrowed Context Schritt für $J \to G$ existiert den $J \to G'$ nicht beantworten kann brechen wir den Algorithmus ab [Zeilen 37-41]. Anschließend betrachten wir das Paar $(J \to G', J \to G)$ und führen dieselben Schritte durch. Die Rückgabe des Algorithmus ist ein Flag *isBisim*, dass den simularen Schritt bezeugt, sowie eine Menge von erreichbaren Graphen $(K \to H, K \to H')$. Nun können wir einen Algorithmus definieren der für eine gegebene Relation $\mathcal{R}$ überprüft, ob sie eine Bisimulation ist.

ALGORITHMUS BisimulationsCheckGraphKomplett($\mathcal{R}$, *Ruleset*)
01 Set $relation := \mathcal{R}$; [Zu überprüfende Relation]
02 Boolean isBisim := true; [Bisimulations Flag]
03 **FÜR** $\forall(J, jg, G, jg', G') \in relation$
04 $(isBisimT, results) :=$ BisimulationsCheckGraphTupel$((J, jg, G, jg', G'), Ruleset)$;
05 **FÜR** $\forall\ (K, kh, H, kh', H') \in results$
06 **FALLS** $(K, kh, H, kh', H') \notin relation$ und $(K, kh', H', kh, H) \notin relation$
07 **DANN** [Überprüfung ob das Nachfolgepaar Up-To-Context enthalten ist]
08 isBisim := false; [Es wird angenommen der Nachfolger fehlt in der Relation]
09 **FÜR** $\forall(J, jg, G, jg', G') \in relation$
10 isBisim := isBisim $\vee$ UpToContext$((J, jg, G, jg', G'), (K, kh, H, kh', H'))$
$\vee$ UpToContext$((J, jg, G, jg', G'), (K, kh', H', kh, H))$;
11 **ENDE FÜR**
12 **ENDE FALLS**
13 **FALLS** ¬isBisim
14 **DANN**
15 **RÜCKGABE** $(False)$;
16 **ENDE FALLS**
17 **ENDE FÜR**
18 **ENDE FÜR**
19 **RÜCKGABE** $(isBisim)$

Der Algorithmus betrachtet nacheinander die Tupel $(J \to G, J \to G')$ und berechnet mithilfe des Borrowed Context, für die Menge der Regeln *Ruleset*, alle Nachfolgertupel $(K \to H, K \to H')$ [Zeile 4]. Anschließend wird überprüft ob die Nachfolger in der Relation enthalten sind. Sollte ein Nachfolger nicht direkt in der Relation enthalten sein überprüfen wir ob sie Up-To-Context enthalten ist [Zeilen 5-17]. Die Methode UpToContext müssen wir nun im Folgenden definieren.

5.2.6. Up-To-Context

Die Up-To-Context Optimierung wurde in Kapitel 4.6.2 erläutert und dient dazu, die unendlich große Relation die sich aus Borrowed Context Schritten berechnen ließe, möglicherweise endlich zu halten. Desweiteren verwenden wir sie dazu um Nachfolgepaare $(K \to H, K \to H')$ eines Graphenpaares $(J \to G, J \to G')$ in einer gegebenen Relation zu bestätigen. Lässt sich ein in der Relation befindendes Paar, um einen Kontext C erweitern, sodass das resultierende Tupel $(K \to H, K \to H')$ entspricht, so ist $(K \to H, K \to H')$ in der Bisimulation enthalten. Grundlage der Berechnung ist das Diagramm 5.5.

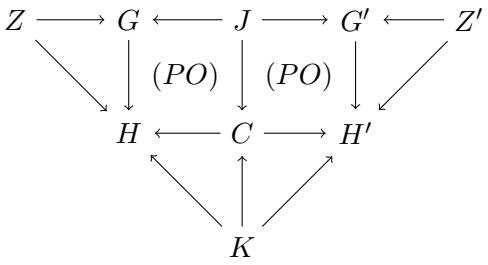

Abbildung 5.5.: Diagramm für Up-to-Context

Wir beginnen mit der Definition eines Algorithmus zur Berechnung aller Monomorphismen $gh : G \to H$. Diese werden benötigt, um im Folgenden den Kontextgraph C in der Mitte des Diagramms 5.5 zu bestimmen. Betrachtet wird in diesem Algorithmus der Span $G \leftarrow \emptyset \to K$ aus dem linken Teil des Diagramms. Die Hilfsfunktionen lauten:

HILFSFUNKTIONEN
$|X| : Set \to \mathbb{N}$ [Gibt die Anzahl der Elemente der Menge X zurück]
$tuples : (Set_1, \ldots, Set_n) \to Set_1 \times \ldots \times Set_n$ [Berechne das Kreuzprodukt]
$soutar : f \times G \times L \to Boolean$ [True falls source und targets nicht vertauscht]
$calculateMonos_V : G \times H \to Set$ [Gibt alle Monomorphismen $V_G \to V_H$]
$calculateMonos_E : G \times H \to Set$ [Gibt alle Monomorphismen $E_G \to E_H$]

Der Algorithmus BerechneMonosImKontext erhält als Eingabe das Quintupel (Z, zg, zh, G, H) und berechnet uns die Menge $gh_{Monos}Set : G \to H$, so dass das Diagramm 5.5 kommutiert. In unserem Fall wird der Graph Z nicht benötigt da der zusätzliche Graph Z wie in dem Diagramm 4.19 dargestellt $\emptyset$ entspricht. Der Algorithmus basiert auf der Idee des in Kapitel 5.2.4 vorgestellten Algorithmus BerechneIsomorphismen.

ALGORITHMUS BerechneMonosImKontext(Z, zg, zh, G, H)
01 **FALLS** $|V_G| > |V_H|$ oder $|E_G| > |E_H|$
02 **DANN** [Falls die Anzahl der Knoten oder Kanten in G größer sind]
03 **RÜCKGABE** $\emptyset$ [Kein Monomorphismus möglich]
04 **ENDE FALLS**
05
06 $monobase_V := calculateMonos_V(G, H)$ [Menge aller Knoten Monomorphismen]
07 $monobase_E := calculateMonos_E(G, H)$ [Menge aller Kanten Monomorphismen]
08
09 $gh_{Set} := tuples((monobase_V, monobase_E))$; [Menge aller Monomorphismen gh]

10 **FÜR** $\forall\ gh \in gh_{Set}$
11 **FALLS** $\neg soutar(gh_{Set}, G, H)$
12 **DANN** [Source und Target wurden beim Abbilden verdreht]
13 $gh_{Set} := gh_{Set} \setminus \{gh\}$;
14 **ENDE FALLS**
15 **ENDE FÜR**
16
17 **FALLS** $(Z, zg, zh) \neq (\emptyset, \emptyset, \emptyset)$
18 **DANN** [Falls Graph Z und Morphismen $zg : Z \to G, zh : Z \to H$ gegeben]
19 **FÜR** $\forall\ gh \in gh_{Set}$
20 **FALLS** $gh \circ cg \neq ch$
21 **DANN** [Monomorphismus gh beachtet nicht den vordefinierten Morphismus]
22 $gh_{Set} := gh_{Set} \setminus \{gh\}$;
23 **ENDE FALLS**
24 **ENDE FÜR**
25 **ENDE FALLS**
26 **RÜCKGABE** gh_{Set}

Znächst wird überprüft ob der Graph $K \to H$ wirklich einem durch Kontext $J \to C \leftarrow K$ erweiterten Graphen $J \to G$ entsprechen kann, indem die Anzahl der Knoten und Kanten untersucht werden [Zeilen 1-4]. Die Anzahl der Knoten und Kanten von H müssen jeweils größer oder gleich denen in G sein. Anschließend werden disjunkt die Mengen der Monomorphismen für Knoten $monobase_V$ und Kanten $monobase_E$ berechnet [Zeilen 6-7] und zu vollständigen Morphismen $gh : G \to H$ kombiniert [Zeile 9]. Es wird überprüft ob Quell und Zielknoten beim abbilden einer Kante nicht vertauscht wurden [Zeilen 10-15] und zuletzt welche der gefundenen Monomorphismen, die Kriterien des zusätzlich definierten Graphen Z erfüllen [Zeilen 17-25].

Mit der Menge der Monomorphismen gh_{Set} gelingt nun die Definition eines Algorithmus, der überprüft ob das Graphenpaar $(H \leftarrow K \to H')$ bereits durch ein in der Bisimulation vorhandenes Graphenpaar $(G \leftarrow J \to G')$ Up-To-Context vertreten wird. Die Hilfsfunktionen in diesem Algorithmus sind die folgenden:

HILSFUNKTIONEN
$invertMorph : f \to f^{-1}$ [Invertiert den Morphismus f]
$|X| : Set \to \mathbb{N}$ [Gibt die Anzahl der Elemente der Menge X zurück]
$pushout : A \times f \times B \times g \times C \to D \times g' \times f'$ [Aus $(A, \to, B, \downarrow, C)$ berechne $(D, \downarrow, \to)$]
$pushoutC : A \times f \times B \times g' \times D \to C \times g \times f'$ [Aus $(A, \to, B, \downarrow, D)$ berechne $(C, \downarrow, \to)$]

Die Eingabe des Algorithmus UpToContext sind die beiden Quintupel (J, jg, G, jg', G') und (K, kh, H, kh', H') die jeweils ihre Graphenpaare $(J \to G, J \to G')$, $(K \to H, K \to H')$ und somit die beiden Graphenpaare $(G \leftarrow J \to G')$, $(H \leftarrow K \to H')$ beschreiben. Da wir den zusätzlichen Graph Z mit seinen Morphismen $zg : Z \to G$ und $zh : Z \to H$ nicht benötigen fehlt er für diesen Algorithmus als Übergabeparameter und wir können eine vereinfachte Variante der Berechnung definieren. Die Ausgabe ist ein Boolean der angibt ob ein Kontextgraph C existiert, sodass das Diagramm 5.5 existiert.

ALGORITHMUS UpToContext$((J, jg, G, jg', G'), (K, kh, H, kh', H'))$
01 gh_{Set} := BerechneMonosImKontext$(\emptyset, \emptyset, \emptyset, G, H)$;
02 **FÜR** $\forall\ gh \in gh_{Set}$
03 $(C, jc, ch) := pushoutC(J, jg, G, gh, H)$; [Berechnung des Kontext C]
04 $kc := invertMorph(ch) \circ kh$; [Berechnung des fehlenden Morphismus $kc : K \to C$]
05 $(H'_C, g'h'_C, ch'_C) := pushout(J, jg', G', jc, C)$;
06 $kh'_C := ch'_C \circ kc$; [Vervollständigen von $K \to H'_C$];
07 **FALLS** |BerechneIsomorphismen$(K, kh', kh'_C, K', K'_C)| > 0$
08 **DANN** [Graphen $K \to H'$ und $K \to H'_C$ sind gleich]
09 **RÜCKGABE** true [Es konnte ein Kontext $J \to C \leftarrow K$ gefunden werden]
10 **ENDE FALLS**
11 **ENDE FÜR**
12 **RÜCKGABE** false [Es konnte kein Kontext $J \to C \leftarrow K$ gefunden werden]

Der Algorithmus berechnet zunächst die Menge der Monomorphismen $gh : G \to H$ und speichert diese in gh_{Set} [Zeile 1]. Für jeden Monomorphismus $gh : G \to H$ wird nun via Pushout Complement der Cospan $(J \to C \leftarrow K)$ vervollständigt [Zeilen 3-4]. Mithilfe eines Pushouts mit H und C über J wird ein möglicher Kandidat $K \to H'_C$ berechnet [Zeilen 5-6] dessen Isomorphie zu $K \to H'$ ausreicht um die Existenz des Diagramms 5.5 zu zeigen [Zeilen 7-10]. Wird eine solche Isomorphie gefunden gibt der Algorithmus true zurück und terminiert [Zeile 9]. Sofern kein isomorpher Kandidat $K \to H'$ existiert gibt der Algorithmus false zurück [Zeile 12].

Dieser einfache Ansatz funktioniert aufgrund des Wissens, dass an beiden Enden des Diagramms keine weiteren Graphen Z und Z' vorhanden sind. Wir betrachten im folgenden ein Verfahren mit dem bei zusätzlichen Spans $G \leftarrow Z \to H$ und $G' \leftarrow Z' \to H'$ überprüft werden kann, ob ein Graphenpaar $(K \to H, K \to H')$ Up-To-Context durch $(J \to G, J \to G')$ bereits in der Bisimulation enthalten ist.

Beispiel 5.2.2 (Up-To-Context mit zusätzlichen Graphen) *Eine Vorgehensweise für vorhandene Graphen Z und Z', ist mit den bereits definierten Algorithmen leicht umzusetzen. Die Methoden die hierfür benötigt werden sind* BerechneMonosImKontext *(Kapitel 5.2.6) und* BerechneIsomorphismen *(Kapitel 5.2.4).*

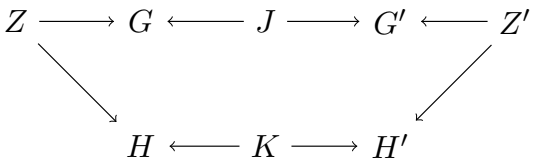

Abbildung 5.6.: Up-To-Context: Ausgangs-situation

Ausgangs-situation sind die beiden Cospans $(Z \to G \leftarrow J)$, $(Z \to H \leftarrow K)$ und die zusätzlichen Cospans $(J \to G' \leftarrow Z')$, $(K \to H' \leftarrow Z')$, so dass das Diagramm 5.6 existiert. Unser Ziel ist die Erweiterung dieser Ausgangs-situation bis zum Diagramm 5.5, wobei die beiden $\emptyset$ links und rechts durch die zuätzlichen Graphen Z und Z' ersetzt werden.

Für die Berechnung betrachten wir vorerst nur den linken Teil des Diagramms 5.6. Der relevante Ausschnitt wird in Diagramm 5.7 dargestellt. Unser Ziel ist die Berechnung des Kontext $J \to C \leftarrow K$ der für die Graphen $J \to G$ in diesem Kontext zu $K \to H$ transformiert wird. Diesen Kontext $J \to C \leftarrow K$ muss es genauso für die Graphen $J \to G'$ geben, sodass wir das Resultat $K \to H'$ erhalten.

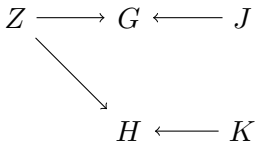

Abbildung 5.7.: Up-To-Context: Beginn linke Seite

Zunächst berechnen wir den Monomorphismus $gh : G \to H$ (siehe Diagramm 5.8). Dazu nutzen wir die Methode $BerechneMonosImKontext(Z, zg, zh, G, H)$ aus diesem Kapitel und berechnen somit die Menge der möglichen Monomorphismen, die den zusätzlichen Graphen Z und die Morphismen $zg : Z \to G$ sowie $zh : Z \to H$ erfüllen. Da die Methode mehrere Monomorphismen berechnen kann, müssen die folgenden Schritte für sämtliche Monomorphismen $gh : G \to H$ durchgetestet werden.

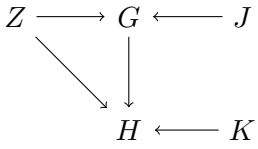

Abbildung 5.8.: Up-To-Context: Monomorphismus $gh : G \to H$

Nun wenden wir die Pushoutkonstruktion an und berechnen mit Hilfe des Pushout Komplements den Kontextgraphen C mit den dazugehörigen Morphismen $jc : J \to C$ und $ch : C \to H$. Dazu entfernen wir den Graphen G von H mit Beibehaltung der Knoten und Kanten welche durch das Interface J gegeben sind. Das entstandene Quadrat ist in Diagramm 5.9 abgebildet.

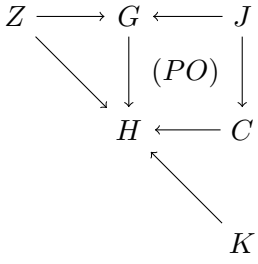

Abbildung 5.9.: Up-To-Context: Pushout

Um den Kontext $(J \to C \leftarrow K)$ zu vervollständigen benötigen wir den fehlenden Morphismus $kc : K \to C$. Um diesen zu berechnen bilden wir die Verkettung der Inversen von $ch : C \to H$ mit $kh : K \to H$. Der berechnete Morphismus $kc : K \to C$ muss total sein. So erhalten wir das Diagramm 5.10.

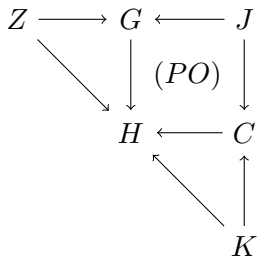

Abbildung 5.10.: Up-To-Context: Kontext $J \to C \leftarrow K$

Nun betrachten wir den in Diagramm 5.11 abgebildeten rechten Part. Ziel ist hier die Berechnung des Kontext $J \to C' \leftarrow K$, sodass die Graphen $J \to G'$ im Kontext zu $K \to H'$ transformiert werden. Sofern es uns gelingt zu zeigen das der berechnete Kontext $J \to C \leftarrow K$ der linken Seite, und der zu berechnende Kontext $J \to C' \leftarrow K$ der rechten Seite identisch sind, wird das Graphenpaar $(K \to H, K \to H')$ Up-To-Context durch $(J \to G, J \to G')$ in der Bisimulation vertreten.

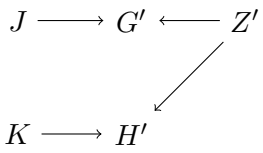

Abbildung 5.11.: Up-To-Context: Beginn rechte Seite

Symmetrisch zum linken Part berechnen wir den Kontext $(J \to C' \leftarrow K)$. Wir berechnen die Monomorphismen $g'h' : G' \to H'$ mit $BerechneMonosImKontext(Z', z'g', z'h', G', H')$. Via Pushout Komplement berechnen wir das rechte Quadrat und vervollständigen den Kontext $(J \to C' \leftarrow K)$ durch Berechnung des Morphismus $kc' : K \to C'$ durch $kc' := h'c' \circ kh'$. Der berechnete Morphismus $kc' : K \to C'$ muss total sein. Wir gelangen zum Zwischenstand, der in Diagramm 5.12 dargestellt ist.

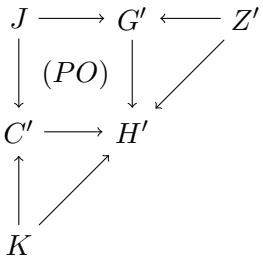

Abbildung 5.12.: Up-To-Context: Kontext $J \to C' \leftarrow K$

Wir kennen nun die Kontexte $(J \to C \leftarrow K)$ sowie $(J \to C' \leftarrow K)$, die für die Kontexterweiterung der linken Seite $(J \to G) \overset{J \to C \leftarrow K}{\to} (K \to H)$ und der rechten Seite $(J \to G') \overset{J \to C' \leftarrow K}{\to} (K \to H')$ existieren. Als nächstes zeigen wie die Gleichheit dieser Kontexte um so die Up-To-Context Kongruenz zu verifizieren.

Die Cospans $(G \leftarrow J \rightarrow G')$ und $(H \leftarrow K \rightarrow H')$ stehen genau dann in einer Up-To-Context Kongruenz sofern ein Isomorphismus $c_{Iso} : C \rightarrow C'$ gefunden werden kann, sodass für die Kontexte $(J \rightarrow C \leftarrow K)$ und $(J \rightarrow C' \leftarrow K)$ das Diagramm 5.13 existiert.

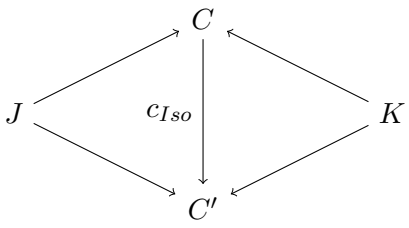

Abbildung 5.13.: Up-To-Context: Isomorphismus $c_{Iso} : C \rightarrow C'$

Der Isomorphismus kann mit der Methode $BerechneIsomorphismen(J, jc, jc', C, C')$ aus Kapitel 5.2.4 berechnet werden. Um den Zusammenhang zwischen den Verfahren leichter zu sehen kann das Interface K dupliziert werden. Für das Verständniss über die Gleichheit dieser Methodik, können die Diagramme 5.14 und 5.4 verglichen werden. Der Morphismus $id_K : K \rightarrow K$ beschreibt hierbei die Identität.

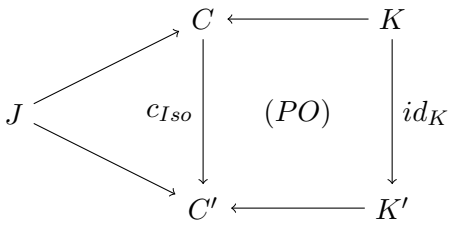

Abbildung 5.14.: Up-To-Context: Adaption für Isomorphismus-Algorithmus

Die nun definierten Algorithmen sollten im Rahmen dieser Arbeit implementiert werden. Das hierfür verwendete Programm heißt Wolfram Mathematica und basiert auf einer Programmiersprache die verschiedenste Programmierparadigmen unterstützt.

6. Mathematica

Das für diese Ausarbeitung gewählte Programm für eine Umsetzung der definierten Algorithmen ist *Wolfram Mathematica*. Mathematica ist ein kommerzielles Programm, erschaffen von *Stephen Wolfram* und seit 1988 auf dem Markt erhältlich. Vor der Veröffentlichung gab es verschiedenste Software Packete, die zur Lösung algebraischer, numerischer oder grafischer Probleme genutzt werden konnten. Jedoch existierte keines, dass wie Mathematica, sämtliche Lösungen dieser Probleme in sich vereinte. [Res12a]

6.1. Allgemeines

Mathematica wird vor allem in naturwissenschaftlichen Bereichen im Studium und in der Wirtschaft verwendet. Die Grundidee, besonders viele Konzepte zur Verfügung zu stellen, macht Mathematica zum Lösen von Problemen sehr attraktiv. Die wichtigsten Eigenschaften sind vor allem

Multiple Paradigmen Die Programmiersprache von Mathematica umfasst verschiedenste Programmierparadigmen. Der Nutzer ist nicht an einen Programmierstil gebunden und kann selbst entscheiden ob er Prozedural, Rekursiv, Funktional, Listen-, oder Regel-basiert arbeiten möchte.

Interaktive Erstellung Berechnungen und ihre mögliche Visualisierung sind nicht statisch sondern können dynamisch beim Programmieren erzeugt werden. Das Ändern von Variablen kann im Arbeitsfluss Schritt für Schritt verfolgt werden. Die Darstellung der Berechnung funktioniert dokumentenartig und ist leicht verständlich.

Automatisierung Mathematica besitzt in der Berechnung, Visualisierung und Entwicklung eine intelligente Automatisierung. Die Darstellung von Grafiken und Auswahl der Algorithmen zur berechnung diverser Probleme ist in Mathematica ausgefeilter, als in Programmen die sich beschränkter Mittel bedienen. Für die Entwicklung ist vor allem die Platformunabhängigkeit ein immenser Vorteil des Programms.

All-In-One Packet Mathematica besitzt keine Add-Ons. Das Programm besitzt Funktionalitäten in diversen naturwissenschaftlichen Bereichen und ist somit ein Komplettpacket, mit dem keine weiteren Kosten entstehen.

Die derzeitige Version von Mathematica lautet *Wolfram Mathematica 8*. Mathematica wurde über die Jahre immer reicher an Funktionen und Grafikvisualisierungen. Eine der neuesten Funktionen ist der *Free-Form-Input*, bei dem der Nutzer englische Sätze als Eingabe für Funktionsbeschreibungen verwenden kann, anstatt diese in einer Syntax aufzuschreiben.

6.2. Nutzung von Mathematica

In Mathematica nutzt man sogenannte Notebook Dateien zur Ausführung der Berechnungen. Zu Beginn lädt der Benutzer Pakete deren Funktionen benutzt werden sollen um spezifische Probleme zu lösen. Anschließend können nach Belieben eigene Funktionen auf Kriterien untersucht werden, sowie Tabellen/Berechnungen ausgewertet und dargestellt werden. Ein Beispielcode einer Notebook Datei ist in Abbildung 6.1 dargestellt.

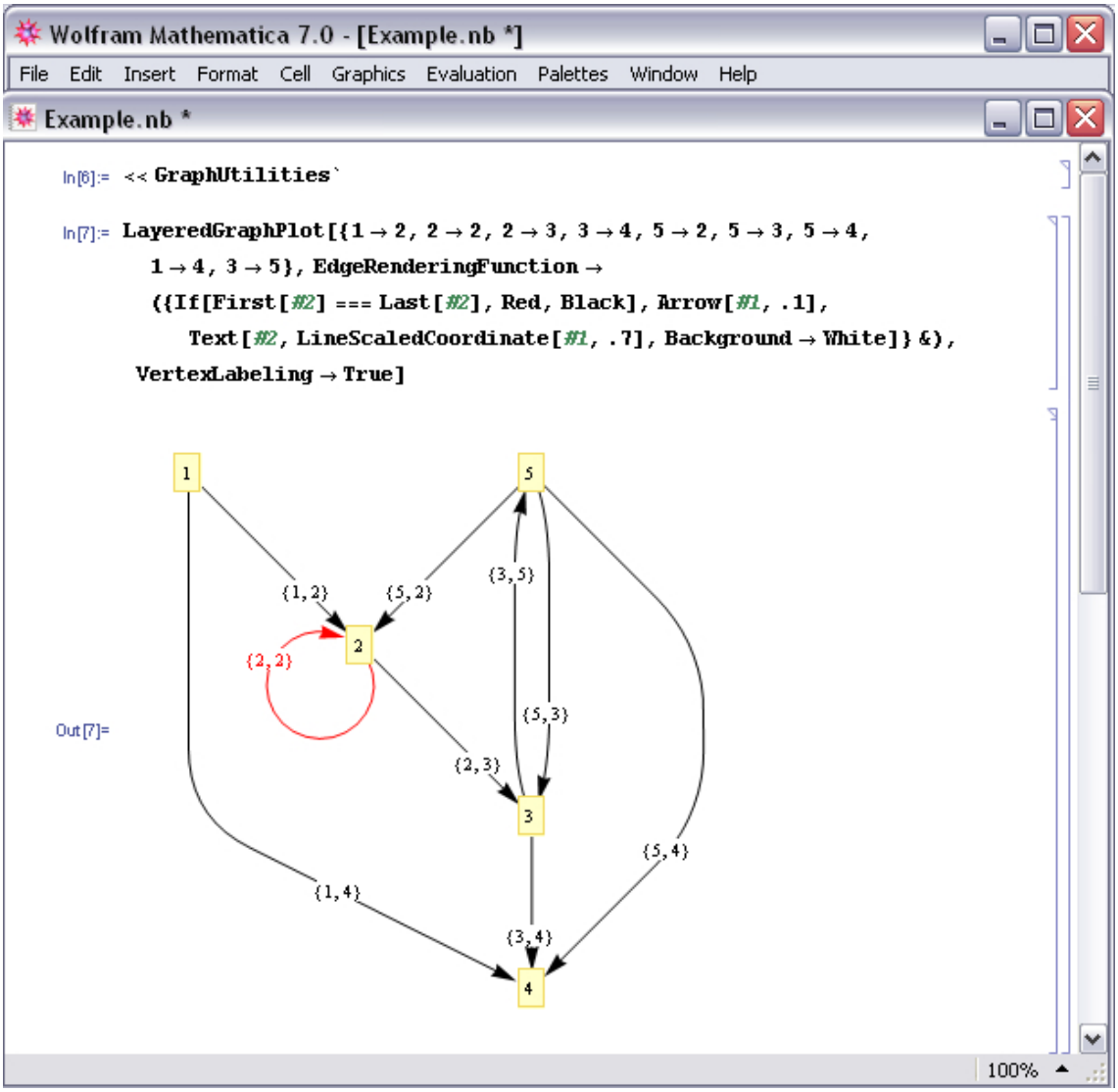

Abbildung 6.1.: Mathematica Notebook Datei

Eine Übersicht der in Mathematica vordefinierten Pakete und Methoden findet sich auf der Entwicklerseite (siehe [Res12b]). Dort finden sich zudem Tutorials und Beispiele um den Umgang mit Mathematica zu erlernen. Im folgenden Kapitel wird zusätzlich ein User Manual beschrieben um die Algorithmen dieser Ausarbeitung anwenden zu können.

7. Implementierung

In diesem Kapitel betrachten wir die Implementation der definierten Algorithmen. Wir beginnen mit einer Übersicht der erstellten Pakete und ihrer Methoden. Im Anschluss folgt ein kurzes User Manual zum Benutzen Methoden in Mathematica.

7.1. Übersicht

Es folgt eine Auflistung der Pakete und Methoden, die für die Umsetzung der Algorithmen geschrieben wurden. Die Variablenbezeichnungen im Quellcode und in den Methodenköpfen, sind den in Abbildung A.1, Abbildung A.2 und Abbildung A.3 angegebenen Diagrammen, im Anhang A nachempfunden.

7.1.1. Pakete

In dieser Ausarbeitung sind zwei Pakete entstanden, die einen umfangreichen Satz an Methoden für die Berechnung mit Graphtransformationssystemen unter Verwendung des Borrowed Context zur Verfügung stellen. Die Namen der Pakete und ihre Funktionen sind die folgenden:

BorrowedContext.m Das Paket BorrowedContext besitzt Methoden zur Grapherzeugung und Visualisierung. Graphen und Morphismen können definiert und für Pushout- und Pullback-Konstruktionen verwendet werden. Alle grundlegenden Methoden der slgebraischen Graphersetzung sind definiert. Die Borrowed Contexts für einen Graph mit Interface $(J \to G)$ können für selbstdefinierte Regeln berechnet werden. Die Berechnung von partiellen Matchings zwischen Graphen und Bisimulationsverifikation sind ebenfalls Bestandteil des Pakets. Zusätzlich wird die Up-To-Context-Methodik bereitgestellt womit Äquivalenzen für Cospans mit zusätzlichem Kontext berechnet werden können.

HischkoffBisim.m Das Paket HirschkoffBisim ermöglicht die Definition von LTS. Bisimulationen können verifiziert oder mit dem Fixpunkt/Hirschkoff Algorithmus berechnet werden.

Pakete besitzen in Mathematica die Endung *Dateiname*.m während Notebook-Dateien durch die Endung *Dateiname*.nb gekennzeichnet sind. Das Hauptpaket dieser Ausarbeitung ist das BorrowedContext Paket. Dieses stellt den Hauptteil der Methoden zur Verfügung. Für die Erzeugung von Bisimulationen ist das HirschkoffBisim Paket besser geeignet, da es die direkte Definition der LTS unterstützt.

7.1.2. Methoden

Die folgenden Tabellen geben eine Übersicht über den Großteil der implementierten Methoden sowie den Algorithmen die sie umsetzen. Tabelle 7.1 zeigt die wichtigsten implementierten Methoden aus dem HirschkoffBisim Paket die zur Umsetzung der Algorithmen verwendet werden können. Dabei werden für jede Methode, die im Quellcode verwendeten Übergabeparameter sowie die Rückgabeparameter spezifiziert. Für ein besseres Verständnis der Methoden folgen in Kapitel 7.2.4 Beispiele.

Kategorie	Parameter
Methodenname	isBisimularTransitionTuple
Übergabeparameter	Transitionsystem1,Transitionssystem2,nodeTuple_List
Rückgabeparameter	isBisim_Bool,answerTuples_List
Methodenname	isBisimularTransitionsysRelation
Übergabeparameter	Transitionsystem1,Transitionssystem2,relation_List
Rückgabeparameter	isBisim_Bool
Methodenname	buildFixpointRelation
Übergabeparameter	Transitionsystem1,Transitionssystem2
Rückgabeparameter	bisimRelation_List
Methodenname	buildHirschkoffRelation
Übergabeparameter	Transitionsystem1,Transitionssystem2,relationTuple_List
Rückgabeparameter	isBisim_Bool,R_List
Methodenname	Propagate
Übergabeparameter	V_List,R_List,W_List,child_List
Rückgabeparameter	status,R_List,W_List
Methodenname	transitionSystem
Übergabeparameter	KnotenIDs,KantenIDs,Source,Target,KnotenIDs,Kantenlabel
Rückgabeparameter	Transitionssystem
Methodenname	createTwoRandomTransitionsystem
Übergabeparameter	AnzahlKnoten,AnzahlKanten,Labelmenge
Rückgabeparameter	Transitionssystem1,Transitionssystem2

Tabelle 7.1.: HischkoffBisim Methoden

Die _Endungen hinter den Variablen Namen, deklarieren den Typ der Variablen, welcher von der Methode erwartet oder zurückgegeben wird. Die in dieser Ausarbeitung verwendeten Datentypen sind:

_List erwartet die Liste {Inhalt} die aus Elementen unterschiedlichen Typs besteht

_Graph erwartet die Graphstruktur aus Kapitel 7.2.2

_Integer erwartet einen Integer Wert

_Bool erwartet eine boolsche Variable (true oder false)

Ist keine _Endung vorhanden, wird die Variable in Mathematica als Integerwert interpretiert. Zum besseren Verständnis schreiben wir in dieser Ausarbeitung Integer aus.

Kategorie	Parameter
Methodenname	objectGraphR
Übergabeparameter	KnotenIDs,KantenIDs,Source,Target,Knotenlabel,Kantenlabel
Rückgabeparameter	Graph
Methodenname	pushoutComplement
Übergabeparameter	I_Graph,l_List,L_Graph,g_List,G_Graph
Rückgabeparameter	C_Graph,c_List,q_List
Methodenname	pushout
Übergabeparameter	I_Graph,r_List,R_Graph,c_List,C_Graph
Rückgabeparameter	H_Graph,h_List,t_List
Methodenname	pullback
Übergabeparameter	G_Graph,f_List,F_Graph,q_List,C_Graph
Rückgabeparameter	K_Graph,k_List,s_List
Methodenname	partialMatch
Übergabeparameter	l_List,L_Graph,n_List,G_Graph,Prune_Bool
Rückgabeparameter	D_Graph,o_List,m_List
Methodenname	extractPruneContext
Übergabeparameter	{D_Graph,o_List,m_List},l_List,n_List
Rückgabeparameter	{D_Graph,o_List,m_List}
Methodenname	borrowedContext
Übergabeparameter	{J_Graph,n_List,G_Graph}, {L_Graph,l_List,I_Graph,r_List,R_Graph},{D_Graph,o_List,m_List}
Rückgabeparameter	{J_Graph,j_List,F_Graph,k_List,K_Graph},{z_List,H_Graph}
Methodenname	calculateIsomorphism
Übergabeparameter	J_Graph,x_List,y_List,F_Graph,F2_Graph
Rückgabeparameter	f_{Iso}_List
Methodenname	sameBorrowedContext
Übergabeparameter	{J_Graph,j_List,F_Graph,k_List,K_Graph},{z_List,H_Graph} {J_Graph,j_List,F′_Graph,k′_List,K′_Graph},{z′_List,H′_Graph}
Rückgabeparameter	$\{f_{Iso}, k_{Iso}\}$
Methodenname	isBisimularRelation
Übergabeparameter	relation_List,ruleset_List,prune_Integer,upToContext_Bool
Rückgabeparameter	isBisim_Bool
Methodenname	checkBisimulation
Übergabeparameter	relationTuple_List,ruleset_List,prune_Bool
Rückgabeparameter	isBisim_Bool,results_List
Methodenname	contextMorphism
Übergabeparameter	Intern_Graph,d_List,e_List,G_Graph,H_Graph
Rückgabeparameter	w_List
Methodenname	contextExtension
Übergabeparameter	{Intern_Graph,d_List,G_Graph,n_List,J_Graph}, {Intern_Graph,e_List,H_Graph,z_List,K_Graph}
Rückgabeparameter	{J_Graph,u_List,C_Graph,v_List,K_Graph}

Tabelle 7.2.: BorrowedContext Methoden

Die Tabelle 7.2 beschreibt die Methoden des BorrowedContext Pakets. Die Übergabeparameter der Methode isBisimularRelation beinhalten den Integer *prune* sowie einen Boolean upToContext. Beim Prunen können verschiedene Arbeitsweisen definiert werden, je nachdem welche partiellen Matching-Tripel $(D, dl_D \rightarrow L, dg : D \rightarrow G)$ oder $(D, dl_D \rightarrow L, dg' : D \rightarrow G')$ gepruned werden sollen. Die Bedeutung dieser Übergabeparameter sind wie folgt:

Die Prune Variable darf die Werte 0,1 oder 2 annehmen.

- 0 bewirkt das kein partielles Matching-Tripel gepruned wird
- 1 pruned beide partiellen Matchings-Tripel
- 2 pruned nur die partiellen Matching-Tripel $(D, dl_D \rightarrow L, dg : D \rightarrow G)$

Der upToContext Boolean bestimmt ob bei gefundenen Nachfolgern Up-To-Context entschieden werden darf, ob diese durch andere Graphenpaare bereits in der Relation enthalten sind.

Die Tabelle 7.3 beschreibt die Zusammenhänge der definierten Algorithmen zu den implementierten Methoden. Zusätzlich wird angegeben in welchem Packet sich die Methoden befinden. Das Kürzel BC steht hierbei für BorrowedContext und HB für das HischkoffBisim Paket.

Algorithmus	Methodenname	Paket
BisimulationsCheckfürTuple	isBisimularTransitionTuple	HB
BisimulationsCheckKomplett	isBisimularTransitionsysRelation	HB
GreedyBisimulation	buildGreedyBisimularRelation	BC
BaueFixpunktRelation	buildFixpointRelation	HB
Hirschkoff	buildHirschkoffRelation	HB
PartialMatch	partialMatch	BC
Pruning	extractPruneContext	BC
BerechneBorrowedContext	borrowedContext	BC
BerechneIsomorphismen	calculateIsomorphism	BC
GleicherBorrowedContext	sameBorrowedContext	BC
BisimulationsCheckGraphKomplett	isBisimularRelation	BC
BisimulationsCheckGraphTupel	checkBisimulation	BC
BerechneMonosImKontext	contextMorphisms	BC
UpToContext	contextExtension	BC

Tabelle 7.3.: Algortihmen-Methoden

Wir betrachten nun einige Beispiele, um den Umgang mit den Methoden besser zu verstehen. Dafür beginnen wir mit den Definitionen von Graphen und Morphismen und betrachten anschließend Anwendungsbeispiele.

7.2. User Manual

Um die Verwendung der Methoden leichter zu verstehen, folgt eine Einleitung in die Arbeit mit Mathematica. Diese Einführung soll dazu dienen, ein Verständniss für die Definition von Graphen und Morphismen in Mathematica zu vermitteln. Die theoretischen Ansätze werden mit ausreichend Beispielen erklärt.

7.2.1. Notebook-Dateien und Pakete

Die Basis der Arbeit mit Mathematica ist das Erstellen einer Notebook-Datei. Notebook-Dateien sind Umgebungen in denen Variablen, Formeln, Methoden definiert und aufgerufen werden können. Der erste Schritt, um mit den vordefinierten Methoden eines Paketes arbeiten zu können, ist das Laden der Pakete. Die in dieser Ausarbeitung beschriebenen Pakete *BorrowedContext* und *HischkoffBisim* müssen in der ersten Zeile der Notebook-Datei je nach Bedarf geladen werden. Dies geschieht mit dem Eingeben der Zeile:

```
<< HirschkoffBisim‘
```

oder

```
<< BorrowedContext‘
```

Zeilen der Notebook-Datei können stets ausgeführt werden um den derzeitigen Fortschritt innerhalb einer Datei voranzutreiben. Jede Zeile kann Schritt für Schritt kompiliert werden, sodass auch kleine Änderungen in Zeilen nicht zu einer Neuberechnung der gesamten Notebook-Datei führen muss. Um eine Zeile zu kompilieren, muss der Benutzer die Tastenkombination Strg+Enter drücken. Um die gesamte Datei zu kompilieren und so nicht jede Zeile einzeln ausführen zu müssen, kann man alternativ in der Menüleiste den Punkt $Evaluation \rightarrow EvaluateNotebook$ auswählen. Nun können wir damit beginnen Funktionen, Graphobjekte, Morphismen oder sonstige Objekte zu definieren.

Die ‘ Symbole am Ende des Paket Namens ist notwendig, um zu Gewährleisten, dass Unterpakete dieses Paketes rekursiv geladen werden können. Sowohl das HirschkoffBisim als auch das BorrowedContext Paket basieren auf dem *Combinatorica* und *GraphUtilities* Paketen, welche mitgelieferte Paket von Mathematica sind. Einige Methoden arbeiten auf Basismethoden dieser Pakete, daher ist es notwendig diese Pakete rekursiv mit zu laden. Alternativ können Packete mit folgender Zeile geladen werden:

```
Import[Verzeichnispfad\\Paketname.m];
```

Wir betrachten nun Graph-, und Morphismus-Definitionen in Matematica. Für die Implementation der Algorithmen wurde ein iterativer und listenbasierter Programmierstil verwendet. Die Pushout und Pullback Methoden, welche von der Technischen Universität Berlin zur Verfügung gestellt wurden, basierten bereits auf diesem Programmierstil, sodass im Quellcode angeknüpft wurde.

7.2.2. Grapherzeugung und Visualisierung

In Mathematica wird ein Graph ähnlich den Definitionen aus Kapitel 2.2.1 definiert. Der Unterschied besteht in der Trennung der klaren Identifier für Knoten und Kanten in Form von IDs und ihrer Label. Um einen Graphen zu erzeugen verwenden wir die Methode *ObjectGraphR(KnotenIDs,KantenIDs,Source,Target,Knotenlabel,Kantenlabel)*. Die Methode erwarten als Eingabe die folgenden Parameter:

KnotenIDs ist eine Liste $(ID_V^1, ID_V^2, \ldots, ID_V^n)$ wobei n die Anzahl der Knoten des zu erzeugenden Graphen angibt. Die IDs für Knoten sind eine sortierte aufsteigende Reihe natürlicher Zahlen von 1 bis n.

KantenIDs ist eine Liste $(ID_E^1, ID_E^2, \ldots, ID_E^m)$ wobei m die Anzahl der Kanten des zu erzeugenden Graphen sind. Die IDs für Kanten sind eine sortierte aufsteigende Reihe natürlicher Zahlen von 1 bis n.

Source ist eine Abbildung $Source : KantenIDs \to KnotenIDs$, die jeder Kante mit $ID_E^x \in KantenIDs$ einen Quellknoten mit $ID_V^y \in KnotenIDs$ zuweist.

Target ist eine Abbildung $Target : KantenIDs \to KnotenIDs$, die jeder Kante mit $ID_E^x \in KantenIDs$ einen Zielknoten mit $ID_V^y \in KnotenIDs$ zuweist.

Knotenlabel ist eine Liste $(Label_V^1, Label_V^2, \ldots, Label_V^n)$ wobei n der Anzahl der Knoten des Graphen entspricht.

Kantenlabel ist eine Liste $(Label_E^1, Label_E^2, \ldots, Label_E^m)$ wobei m der Anzahl der Kanten des Graphen entspricht.

Durch Aufruf dieser Methode wird ein Graph erzeugt. Ein erzeugter Graph G ist ein Tripel $G = (E_G, V_G, Bool_{Digraph})$, wobei gilt:

- E_G ist eine Liste aus Tupeln $((ID_{V_G}^i, ID_{V_G}^j), (label_E : KantenIDs \to Kantenlabel))$ mit $1 \leq i, j \leq n$. Die Tupel $(ID_{V_G}^i, ID_{V_G}^j)$ sind Elemente aus der Relation (KnotenIDs $\times$ KnotenIDs) wobei $ID_{V_G}^i = Source(ID_{E_G}^k)$ und $ID_{V_G}^j = Target(ID_{E_G}^k)$ für eine Kante mit Position k in der Liste E_G. Das zweite Element $label_E : KantenIDs \to Kantenlabel$ ist eine Abbildung die einer Kante mit $ID_{E_G}^i$ ein Label $Label_E^i$ zuweist.
- V_G ist eine Liste aus Tupeln $((0,0), (label_V : KnotenIDs \to Knotenlabel))$. Da Knoten keine Abbildungen beschreiben bleibt das erste Tupel undefiniert. Das zweite Element $label_V : KnotenIDs \to Knotenlabel$ ist eine Abbildung die einem Knoten mit $ID_{V_G}^i$ ein Label $Label_V^i$ zuweist.
- $Bool_{Digraph}$ ist ein Boolean der angibt, ob der erzeugte Graph ein gerichteter Graph ist.

Diese Art der Graphinterpretation basiert auf einem Projekt der Technischen Universität Berlin. Zu Beginn dieser Ausarbeitung existierten bereits Methoden zur Berechnung von Pushout und Pullbackkonstruktionen. Die Graphstruktur wurde für die Algorithmen übernommen und um die Labelfunktionen erweitert. Es folgt ein Beispiel für einen Graphen in Mathematica.

Beispiel 7.2.1 (Mathematica Graphenerzeugung) *Wir erzeugen mit der Methode* ObjectGraphR(KnotenIDs,KantenIDs,Source,Target,Knotenlabel,Kantenlabel) *einen Beispielgraphen. Der Aufruf in der Notebook-Datei lautet:*

$G = objectGraphR[\{1,2\},\{1,2\},\{1 \to 1, 2 \to 1\},\{1 \to 2, 2 \to 1\},\{V,V\},\{S,M\}];$

Nach dem Kompilieren dieser Zeile erhalten wir den Graphen $G = (E_G, V_G, Bool_{Digraph})$ *mit den folgenden Eigenschaften:*

$E_G = \{\{\{1,2\}, ELabel \to S\},$
$\{\{1,1\}, ELabel \to M\}\}$

$V_G = \{\{\{0,0\}, VLabel \to V\},$
$\{\{0,0\}, VLabel \to V\}\}$

$Bool_{Digraph} = True$

In Mathematica wird ein solcher Graph als Graph $:< 2,2 >$ *ausgegeben, wobei die erste Zahl der Anzhal der Kanten und die zweite Zahl der Anzahl der Knoten entspricht. Um den Graphen genauer in Mathematica zu betrachten verwendet man die Methode* show. *Der Aufruf der Codezeile in Mathematica lautet*

show[G]

und erzeugt eine Visualisierung des Graphen. Unser Beispielgraph entspricht dem in Abbildung 7.1 gezeigten Graphen.

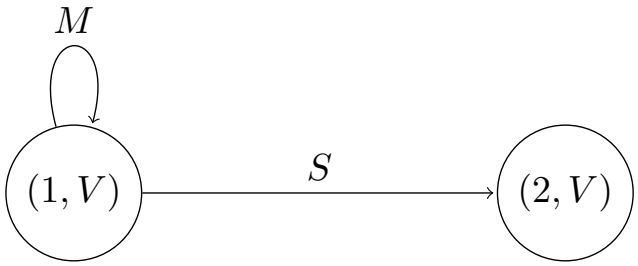

Abbildung 7.1.: Mathematica Beispielgraph

Um die Elemente eines Graphen oder jeglicher Listenbasierten Strukturen in Mathematica zu extrahieren verwendet man den Liste[[MainPosition,SubPosition,...,Position]] Befehl. Ein Beispiel der Verwendung dieses Befehls für den gerade erschaffenen Graphen $G = (E_G, V_G, Bool_{Digraph})$ *wäre:*

G[[1]] liefert E_G

G[[1,2]] liefert die Liste $\{\{1,1\}, ELabel \to M\}$

G[[1,2,1]] liefert das Tupel $\{1,1\}$

Das zweite wichtige Konstrukt zur Realisierung der Borrowed Context Berechnungen sind die Morphismen. Ähnlich den Graphen orientiert sich die Implementierung an der Definition von Listen, wobei Morphismen ohne umständlichen Methodenaufruf generiert werden können.

7.2.3. Morphismen definieren

Wir werden Morphismen ab sofort als Tupel von Listen definieren. Ein Morphismus $gl = (glList_V, glList_E) : G \to L$ besitzt die Listen

- $glList_V = (ID_{V_L}^{k_1}, ID_{V_L}^{k_2}, \ldots, ID_{V_L}^{k_n})$ und
- $glList_E = (ID_{E_L}^{l_1}, ID_{E_L}^{l_2}, \ldots, ID_{E_L}^{l_m})$

mit $1 \leq k_i \leq n'$ und $1 \leq l_j \leq m'$ wobei

n : Anzahl der Knoten in G $\quad$ m Anzahl der Kanten in G

n' : Anzahl der Knoten in L $\quad$ m' Anzahl der Kanten in L

Somit gilt $gl(ID_{V_G}^i) = ID_{V_L}^{k_i}$ und $gl(ID_{E_G}^j) = ID_{E_L}^{l_j}$. In Mathematica werden in den Codezeilen zum definieren von Morphismen Mengenklammern verwendet um Listen zu definieren.

Beispiel 7.2.2 (Mathematica Morphismus) *Der Morphismus $gl : G \to L$ mit der Mathematica Codezeile*

$gl = \{\{4, 5, 3\}, \{3, 2\}\};$

erzeugt einen Morphismus mit folgende Abbildung:

$gl_V(1) = 4, \quad gl_V(2) = 5, \quad gl_V(3) = 3,$

$gl_E(1) = 3, \quad gl_E(2) = 2$

Die Abbildung 7.2 zeigt den resultierenden Morphismus $gl : G \to L$. Die oberen Knoten mit ihren Kanten gehören zum Graphen L und die unteren Knoten mit ihren Kanten zum Graphen G. Der Morphismus $gl : G \to L$ wird durch die gestrichelten Pfeile dargestellt.

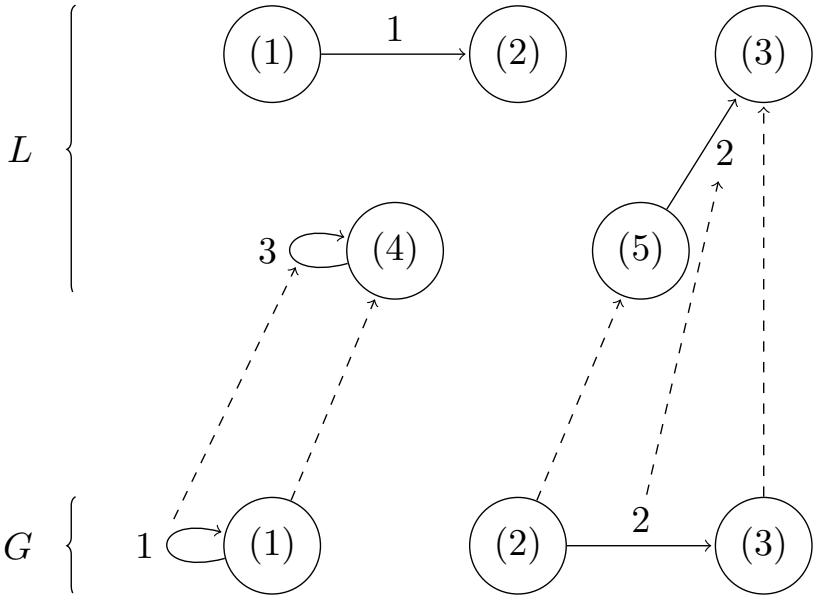

Abbildung 7.2.: Mathematica Beispielmorphismus

7.2.4. Methoden in Notebook-Dateien

Für jede entwickelte Methode existiert eine eigene Beispieldatei um den Umgang mit den Methoden in Mathematica besser zu veranschaulichen. Tabelle 7.4 zeigt für jede auf einem Algorithmus basierende Methode, die entsprechende Notebook-Datei in der diese enthalten ist. Das Kürzel BC steht für das BorrowedContext Paket und HB für das HischkoffBisim Paket.

Methodenname	Notebook-Datei	Paket
isBisimularTransitionTuple	LTSBisim	HB
isBisimularTransitionsysRelation	LTSBisim	HB
buildGreedyBisimularRelation	FlowerGreedy	BC
buildFixpointRelation	Graphcreator/FixpointLoop	HB
buildHirschkoffRelation	Hirschkoff	HB
partialMatch	BorrowedContext	BC
extractPruneContext	RelationPrune	BC
borrowedContext	BorrowedContext	BC
calculateIsomorphism	Isomorphismen	BC
sameBorrowedContext	DerivingBisimulation	BC
isBisimularRelation	Flower	BC
checkBisimulation	DerivingBisimulation	BC
contextMorphisms	ContextMorphism	BC
contextExtension	RelationUpTo/ContextMorphism	BC
createTwoRandomTransitionsystem	Graphcreator	HB
evaluation	MainEvaluation	HB

Tabelle 7.4.: Methoden-Dateien

Die zuletzt genannte Methode *evaluation* wird in Kapitel 8.2 vorgestellt und verwendet, um eine Datenerhebung für die Methoden *buildFixpointRelation* und *buildHirschkoffRelation* zu erhalten. Dazu werden zwei zufällige Transitionssysteme mit der Methode *createTwoRandomTransitionsystem* erzeugt. In der MainEvaluation Notebookdatei ist eine Evaluation für das Knotenintervall 150-200 berechnet worden, deren Ergebnisse unter anderem in Tabelle 8.4 zu sehen sind.

8. Evaluation

Die implementierten Methoden isBisimularRelation, buildGreedyBisimularRelation, buildFixpointRelation und buildHirschkoffRelation sollen auf ihre Performance und Korrektheit überprüft werden. Wir überprüfen die Ergebnisse für Graphtransformationssysteme mit zwei Aufgabentypen. Der erste Aufgabetyp ist die Berechnung einer Bisimulation für ein vorgegebenes Graphenpaar und die zweite Aufgabe ist die Verifikation einer bestehenden Bisimulation bezüglich ihrer Vollständigkeit und Korrektheit. Zusätzlich evaluieren wir den Fixpunkt- und Hirschkoff-Algorithmus unter dem Kriterium der Performance. Interessant ist dabei die Fragestellung mit welcher Wahrscheinlichkeit, für den Hischkoff-Algorithmus bei wachsender Anzahl an Knoten und Kanten, ein bisimulares Starttupel (α, β) gefunden werden kann.

8.1. Borrowed Context Auswertung

8.1.1. Automatische Berechnung einer Bisimulation

Wir betrachten die Ergebnisse der Methoden buildGreedyBisimularRelation. Die Evaluation findet auf Grundlage eines Beispiels statt, dass mit Mathematica umgesetzt wird. Gegeben sei das Graphenpaar $(J \to G, J \to G')$ wie in Abbildung 8.1 dargestellt und die Regeln $(L \leftarrow I \to R)$ und $(L' \leftarrow I' \to R')$ aus Abbildung 8.2.

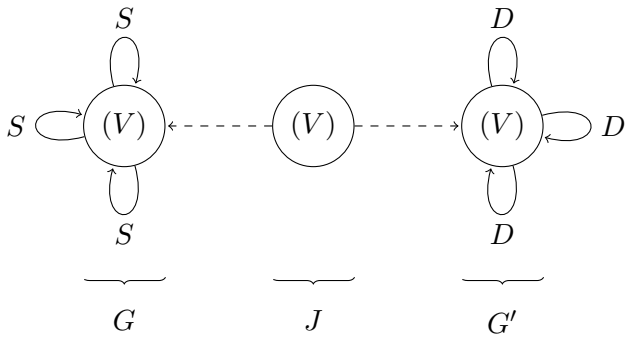

Abbildung 8.1.: Evaluation Greedy: $(J \to G, J \to G')$

Für das Graphenpaar soll soll eine Bisimulation berechnet werden in denen sämtliche Nachfolgepaare enthalten sind. Wir verwenden die folgenden Codezeilen:

$Relation = \{\{J, \{\{1\}, \{\}\}, G, \{\{1\}, \{\}\}, G'\}\};$

$Ruleset = \{\{L, \{\{1\}, \{\}\}, I, \{\{1\}, \{\}\}, R\}, \{L', \{\{1\}, \{\}\}, I', \{\{1\}, \{\}\}, R'\}\};$

$buildGreedyBisimularRelation[Relation, Ruleset, 2]$

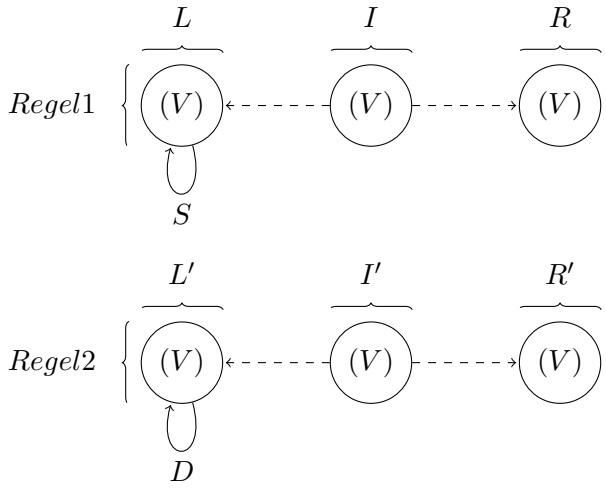

Abbildung 8.2.: Evaluation Greedy: Ruleset

Als Ausgabe erhalten wir das folgende Tupel:

$\{True,$
$\{$
$\{Graph :< 0,1 >, \{\{1\},\{\}\}, Graph :< 3,1 >, \{\{1\},\{\}\}, Graph :< 3,1 >\},$
$\{Graph :< 0,1 >, \{\{1\},\{\}\}, Graph :< 2,1 >, \{\{1\},\{\}\}, Graph :< 2,1 >\},$
$\{Graph :< 0,1 >, \{\{1\},\{\}\}, Graph :< 1,1 >, \{\{1\},\{\}\}, Graph :< 1,1 >\},$
$\{Graph :< 0,1 >, \{\{1\},\{\}\}, Graph :< 0,1 >, \{\{1\},\{\}\}, Graph :< 0,1 >\} \quad \}$

Die Ausgabe bestätigt durch den True Boolean, dass eine Bisimulation berechnet werden konnte. Der zweite Teil der Antwort ist die Bisimulation, wobei jede Zeile einem Graphenpaar mit der Darstellung (J, jg, G, jg', G') entspricht, mit $jg : J \to G$ und $jg' : J \to G'$. Die berechneten Graphenpaare entsprechen den in Abbildung 8.3 dargestellten Graphen.

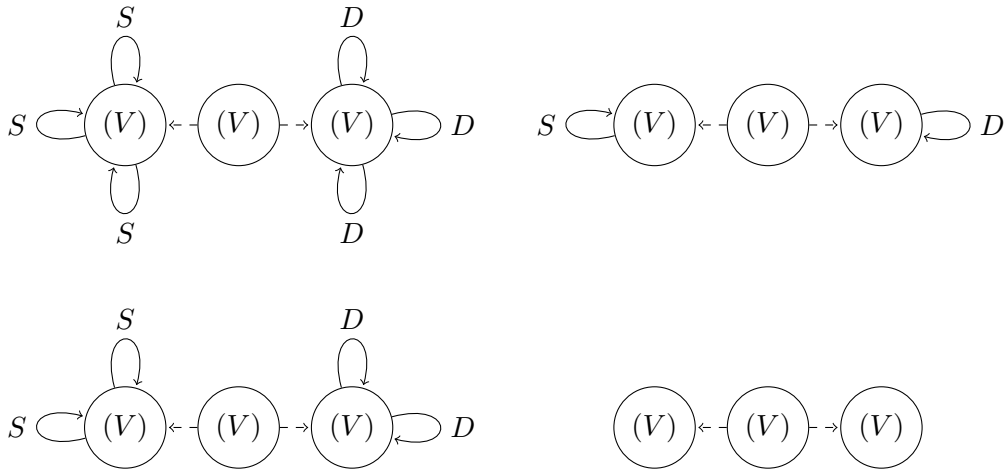

Abbildung 8.3.: Evaluation Greedy: Bisimulation-Graphenpaare

Der Teil des Graphenpaares der aus den S Kanten besteht verliert bei jedem Schritt durch die erste Regel eine Kante. Äquivalent dazu kann der Teil des Graphenpaares der aus den D Kanten besteht, die zweite Regel verwenden um denselben Borrowed Context Schritt zu tätigen. Der Borrowed Context ist dabei der Cospan der lediglich aus jeweils einem Knoten besteht, da die Precondition der Regel vollständig im jeweiligen Graphen enthalten ist. Die berechneten Graphenpaare sind korrekt und die Methode liefert folglich positive Ergebnisse.

8.1.2. Automatische Verifikation einer Bisimulation

Für die Evaluation der isBisimularRelation Methode verwenden wir ein Beispiel-Szenario, bei dem die Übertragung einer Nachricht, über einen Simplex und einen Duplex Kanal modelliert werden soll. Die Regeln sowie das Graphenpaar welches als einziges in unserer zu verifizierenden Bisimulation steht sind in Abbildung 8.4 abgebildet.

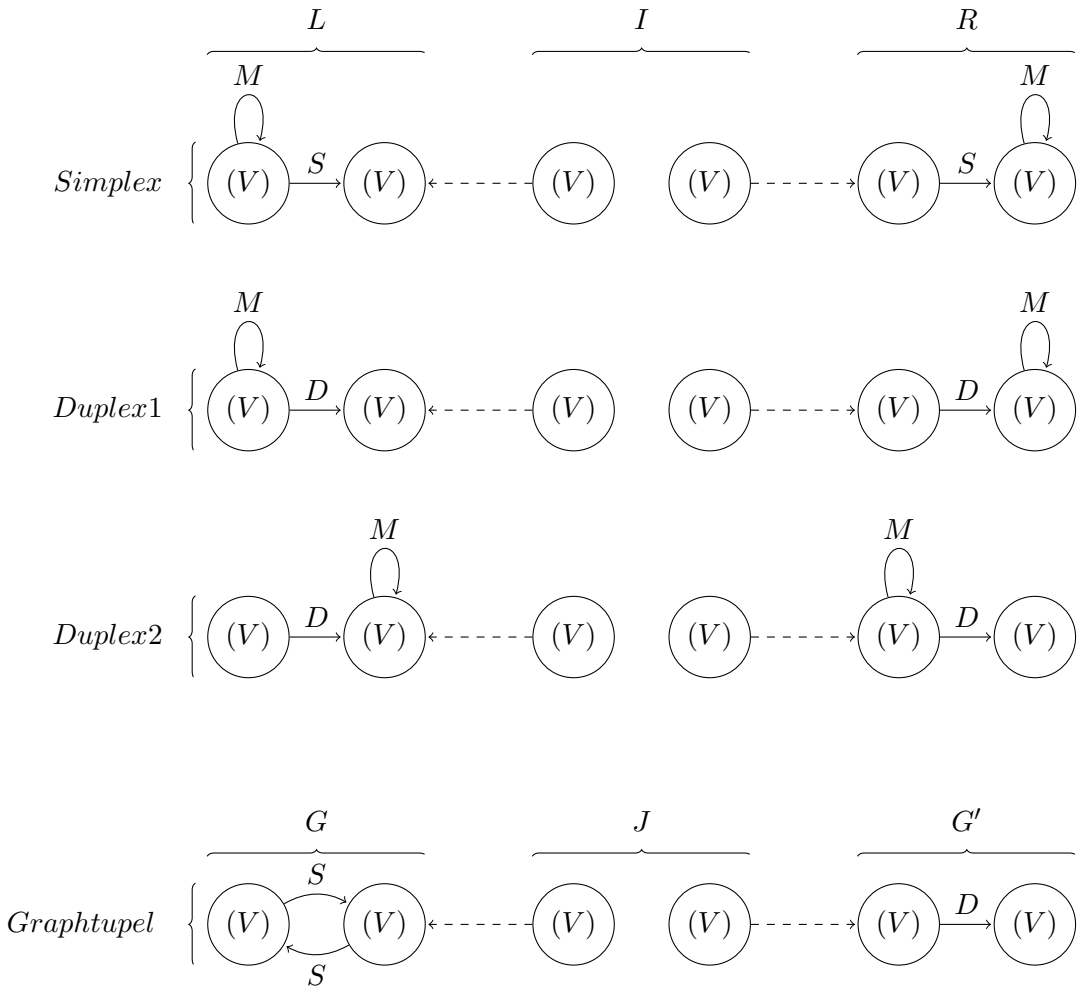

Abbildung 8.4.: Evaluation isBisimularRelation: Regeln und Graphenpaar

Wir wollen eine Bisimulation $\mathcal{R} = \{(G \leftarrow J \rightarrow G')\}$ verifizieren indem wir die Methode isBisimularRelation verwenden. Dazu betrachten wir das Ergebnis der Methode, wenn wir eine Up-To-Context Optimierung benutzen und das Ergebnis wenn wir die Optimierung ausschalten.

Die folgenden Variablen seien in Mathematica definiert:

$Relation = \{\{J, \{\{1,2\},\{\}\}, G, \{\{1,2\},\{\}\}, G'\}\}$;

$Ruleset = \{Simplex, Duplex1, Duplex2\}$;

Bei Verwendung der Methode ohne eine Up-To-Context Optimierung durch Eingabe der Codezeile

```
isBisimularRelation[Relation, Ruleset, 2, False];
```

liefert die Methode **False** als Antwort, da die berechneten Nachfolger (in Abbildung 8.5 dargestellt) nicht in der Bisimulation enthalten sind.

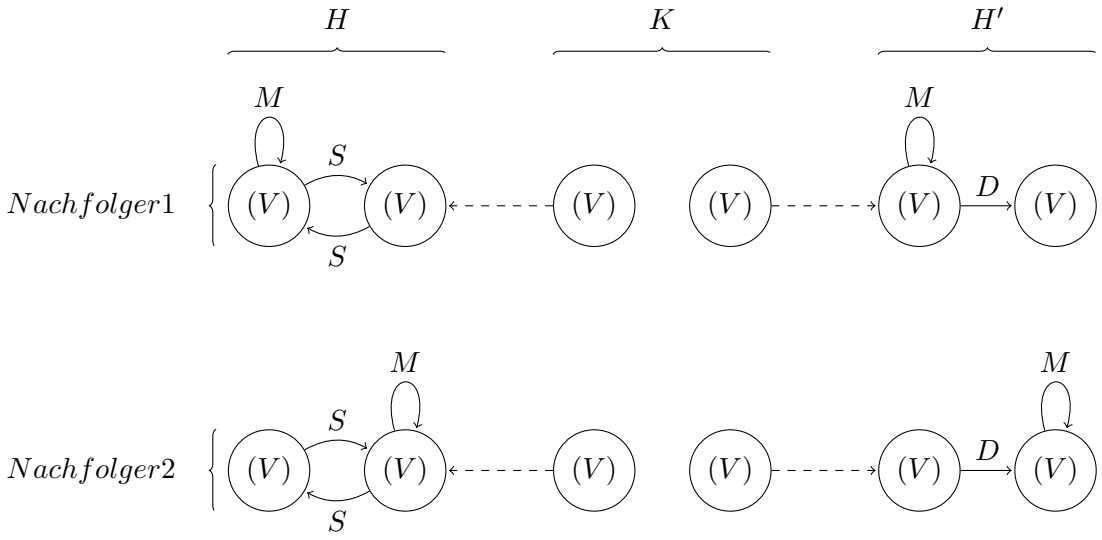

Abbildung 8.5.: Evaluation isBisimularRelation: Nachfolger ($H \leftarrow K \rightarrow H'$)

Verwendet man hingegen den Methodenaufruf

```
isBisimularRelation[Relation, Ruleset, 2, True];
```

liefert die Methode **True**. Grund dafür, ist die Repräsentation der Nachfolger durch das bereits in der Bisimulation enthaltene Graphenpaar ($G \leftarrow J \rightarrow G'$), welches lediglich um den Context einer Kante M erweitert wird.

8.2. Fixpunkt versus Hirschkoff

Wir führen nun eine Evaluation über die Performance und Korrektheit des Fixpunkt-Algorithmus, im Vergleich zum Hirschkoff-Algorithmus, durch. Beide Methoden können genutzt werden um die Bisimularität eines Paares (α, β) zu beweisen (siehe Kapitel 3.2). Der Fixpunkt-Algorithmus berechnet die größtmögliche Bisimulation $\sim$ die alle bisimularen Zustandspaare enthält. Der Hirschkoff-Algorithmus bedient sich einer On-The-Fly Technik mit der eine Bisimulation berechnen wird. Liegt ein betrachtetes Zustandspaar (α, β) in dieser kleineren Bisimulation $\mathcal{R}$ für die stets $\mathcal{R} \subseteq \sim$ ist, so gilt $\alpha \sim \beta$. Es handelt sich hierbei um zwei verschiedene Algorithmen die für unterschiedliche Probleme genutzt werden, die dennoch beide die Frage nach der Bisimularität eines Zustandspaares (α, β) beantworten können.

Wir führen die Evaluation mithilfe einer Methode durch, die zwei zufällige LTS erzeugt. Verfahren zur automaitschen Generierung von Automaten können in [TV05] nachgelesen werden. Mit der Methode *buildFixpointRelation* wird die größtmögliche Bisimulation $\sim$ berechnet. Im Anschluss berechnen wir für jedes in $\sim$ enthaltene Zustandspaar, mit Hilfe der *buildHirschkoffRelation* Methode, diejenigen Bisimulationen die ausreichen um die Bisimularität zu verifizieren. Die Ergebnisse halten wir in Tabellen fest, wie sie in Tabelle 8.1 dargestellt sind.

Einstellung		**Fixpunkt-Daten**		**Hirschkoff-Daten**		
Knoten Anzahl	Kanten Anzahl	Fixpunkt Elemente	Rechen-zeit(sek)	Hirschkoff Elemente	Rechen-zeit(sek)	Prozentsatz bisimularer Paare
XXX	XXX	XXX	X.XXs	XXX	X.XXs	XX.XX%

Tabelle 8.1.: Evaluations Tabelle

Insgesamt führen wir vier Datenerhebungen durch. Für die zufälligen LTS definieren wir eine Methode

$$\text{evaluation}[\{minNodes, maxNodes\}, edgePercent, labels, numberOfTimes]$$

mit der wir zwei LTS generieren. Die Übergabeparameter werden für folgendes Schema benötigt nach dem die LTS erzeugt werden:

1. Erzeuge eine Zufallszahl n in einem vordefinierten Intervall [minNodes,maxNodes] und setze Knotenanzahl := n
2. Berechne Kantenanzahl := n * edgePercent (Für unsere Berechnungen ist edgePercent = 1.85)
3. Lösche aus einem vollständigen LTS mit n Knoten solange Kanten bis die gewünschte Kantenanzahl erreicht ist
4. Belege jede Kante mit einem Kantenlabel aus der Liste label, wobei jedes Label gleich wahrscheinlich ist
5. Wiederhole die Schritte 3-4 ein zweites mal für ein zweites LTS und führe die Evaluation durch
6. Wiederhole die Schritte 1-5 insgesamt numberOfTimes mal

Für jedes so erzeugte LTS-Paar wird wie in Tabelle 8.1 gezeigt die Anzahl der Knoten und Kanten angegeben die für diese Berechnung per Zufallszahlengenerator ermittelt wurde. Die Kantenanzahl ergibt sich in unserer Testreihe aus 185% * Knotenanzahl. Diese Konstante wurde gewählt um für jeden Knoten mit recht hoher Wahrscheinlichkeit, zumindest eine ausgehende Kante zu besitzen. Trivial bisimulare Knotenpaare sollten auf diese Weise unterbunden werden ohne unnötig viele Kanten in dem LTS zu erzeugen. Für die Kantenlabel wurde die Menge $\{a, b\}$ gewählt, wobei für jedes Label eine Wahrscheinlichkeit von 50% besteht, für eine Kante gewählt zu werden. Wir führen Datenerhebungen für die folgenden Knoten Intervalle durch:

25 - 50 Knoten 50-100 Knoten

100-150 Knoten 150-200 Knoten

Dabei geben wir die Anzahl der Tupel in der Bisimulation $\sim$ an, welche durch den Fixpunkt-Algorithmus berechnet wurde. Dahinter folgt die Laufzeit des Programms in Sekunden, die benötigt wurde um $\sim$ zu berechnen. Von allen bisimularen Paaren in $\sim$, wird die Bisimulation $\mathcal{R}$ des Hischkoff-Algorithmus berechnet. Der Betrag der größten aller möglichen Bisimulationen $\mathcal{R}$ mittels Hirschkoff-Algorithmus ist in der fünften Spalte zu sehen, gefolgt von der Berechnungszeit für diese Bisimulation. Zuletzt folgt der Prozentsatz bisimularer Paare für alle möglichen Paare die mit den beiden LTS gebildet werden können. Die Ergebnisse sind in den Tabellen 8.2 (25-50 Knoten), 8.3 (50-100 und 100-150 Knoten) und 8.4 (150-200 Knoten) dargestellt. Für jedes Intervall wurden numberOfTimes := 20 Berechnungen durchgeführt.

Einstellung		**Fixpunkt-Daten**		**Hirschkoff-Daten**		
Knoten Anzahl	Kanten Anzahl	Fixpunkt Elemente	Rechen-zeit(sek)	Hirschkoff Elemente	Rechen-zeit(sek)	Prozentsatz bisimularer Paare
41	76	43	4.66s	2	0.05s	2.55%
46	85	30	6.02s	1	0.04s	1.41%
26	48	25	1.10s	1	0.02s	3.69%
49	91	101	8.05s	2	0.47s	4.20%
35	65	59	2.97s	2	0.15s	4.81%
25	46	9	1.09s	1	0.01s	1.44%
47	87	52	7.76s	2	0.25s	2.35%
39	72	24	3.97s	3	0.05s	1.57%
36	67	44	3.18s	2	0.07s	3.39%
25	46	5	1.13s	1	0.01s	0.80%
48	89	30	7.09s	2	0.21s	1.30%
32	59	20	2.21s	1	0.04s	1.95%
38	70	36	3.74s	1	0.05s	2.49%
37	68	42	3.48s	1	0.05s	3.06%
34	63	30	2.63s	1	0.05s	2.59%
31	57	13	1.91s	2	0.02s	1.35%
44	81	41	5.41s	3	0.08s	2.11%
30	56	32	1.73s	2	0.13s	3.55%
49	91	34	7.47s	2	0.04s	1.41%
42	78	14	5.15s	1	0.03s	0.79%

Tabelle 8.2.: Evaluation mit 25-50Knoten

Einstellung		**Fixpunkt-Daten**		**Hirschkoff-Daten**		
Knoten Anzahl	Kanten Anzahl	Fixpunkt Elemente	Rechen-zeit(sek)	Hirschkoff Elemente	Rechen-zeit(sek)	Prozentsatz bisimularer Paare
65	120	78	17.63s	2	0.39s	1.84%
78	144	140	30.99s	3	1.59s	2.30%
52	96	24	8.95s	1	0.02s	0.88%
55	102	66	11.36s	2	0.25s	2.18%
91	168	122	53.72s	2	0.09s	1.47%
54	100	96	10.36s	2	0.39s	3.29%
66	122	91	20.15s	2	0.50s	2.08%
57	105	65	12.85s	2	0.15s	2.00%
92	170	333	57.29s	2	1.42s	3.93%
93	172	182	57.90s	1	0.34s	2.10%
61	113	67	15.69s	2	0.12s	1.80%
62	115	164	17.02s	5	0.75s	4.26%
89	165	232	51.90s	1	0.11s	2.92%
58	107	82	13.19s	1	0.05s	2.43%
82	152	121	39.78s	1	0.83s	1.79%
87	161	177	48.40s	2	0.31s	2.33%
63	117	127	17.86s	3	1.02s	3.19%
59	109	119	14.78s	3	0.95s	3.41%
98	181	175	70.20s	4	1.23s	1.82%
85	157	195	42.16s	4	0.95s	2.69%
115	213	432	114.37s	3	7.02s	3.26%
112	226	364	141.50s	3	3.44s	2.44%
107	198	360	92.95s	3	1.70s	3.14%
143	265	486	253.50s	3	3.66s	2.37%
117	216	368	116.12s	2	2.90s	2.68%
103	191	282	91.23s	4	1.12s	2.65%
100	185	179	84.66s	3	0.35s	1.79%
115	213	422	125.27s	3	3.49s	3.19%
134	248	388	204.41s	2	3.48s	2.16%
144	266	582	256.42s	3	5.87s	2.80%
147	272	774	261.53s	3	1.71s	3.58%
119	220	341	130.69s	2	1.26s	2.40%
104	192	250	79.97s	2	0.86s	2.31%
111	205	375	103.50s	3	1.37s	3.04%
124	229	354	154.88s	2	0.43s	2.30%
104	192	197	86.54s	3	0.43s	1.82%
145	268	345	269.28s	2	2.65s	1.64%
127	235	256	175.81s	1	0.30s	1.58%
147	272	627	283.28s	2	1.80s	2.90%
109	202	298	107.25s	2	3.70s	2.50%

Tabelle 8.3.: Evaluation mit 50-100 und 100-150 Knoten

Einstellung		**Fixpunkt-Daten**		**Hirschkoff-Daten**		
Knoten Anzahl	Kanten Anzahl	Fixpunkt Elemente	Rechen-zeit(sek)	Hirschkoff Elemente	Rechen-zeit(sek)	Prozentsatz bisimularer Paare
186	344	744	535.68s	3	11.38s	2.15%
187	346	906	611.52s	3	2.44s	2.59%
162	300	651	407.29s	5	9.13s	2.48%
199	368	1116	816.55s	3	6.37s	2.81%
198	366	778	783.45s	2	3.25s	1.98%
195	361	1300	720.26s	3	8.76s	3.41%
168	311	865	388.66s	3	13.16s	3.06%
161	298	615	335.04s	4	9.11s	2.37%
184	340	910	530.32s	3	3.21s	2.68%
200	370	893	707.81s	3	5.16s	2.23%
182	337	585	503.87s	3	7.00s	1.76%
199	368	1160	727.11s	3	14.49s	2.92%
172	318	614	448.05s	3	1.11s	2.07%
151	279	554	285.27s	2	1.06s	2.42%
186	344	820	601.72s	3	3.19s	2.37%
198	366	1181	725.75s	4	19.91s	3.01%
169	313	873	418.46s	4	2.96s	3.05%
174	322	912	489.33s	1	0.13s	3.01%
192	355	866	674.21s	3	5.50s	2.34%
165	305	487	404.50s	2	0.47s	1.78%

Tabelle 8.4.: Evaluation mit 150-200Knoten

Für die Berechnungen wurde ein Terra Laptop mit Intel(R) Pentium(R) CPU B950 Prozessor 2,80GHz und 2,00 GB RAM verwendet, auf dem das Microsoft Windows 7 Home Premium Betriebssystem lief. Aus den Tabellen lassen sich die Ergebnisse in Tabelle 8.5 herleiten.

Kriterium	**Knoten Intervalle**			
Knoten	25-50	50-100	100-150	150-200
ØLaufzeit Fixpunkt	≈4.04s	≈30.6s	≈156.66s	≈555.74s
ØLaufzeit Hirschkoff	≈0.09s	≈0.57s	≈2.38s	≈6.39s
ØElemente Fixpunkt	≈33	≈133	≈384	≈842
ØElemente Hirschkoff	≈2	≈2	≈3	≈3
Øbisimulare Paare	≈2.34%	≈2.44%	≈2.53%	≈2.52%

Tabelle 8.5.: Evaluation Durchschnittliche Ergebnisse

Die Ergebnisse belegen das für die Frage, ob ein Zustandspaar (α, β) bisimular ist, der Hirschkoff-Algorithmus verwendet werden sollte. Soll für zwei gegebene LTS allerdings ein bisimulares Zustandspaar angegeben werden, sollte der Fixpunkt-Algorithmus verwendet werden, da der Prozentsatz bisimularer Paare selbst bei steigender Knotenanzahl sehr gering bleibt. Bei den Auswertungen der Bisimulationen $\mathcal{R}$ galt stets $\mathcal{R} \subseteq \sim$ was die Korrektheit der Ergebnisse stützt. Sofern bei einem Berechnungspaar die größte Hirschkoff-Bisimulation $\mathcal{R}$ aus nur einem Tupel bestand, bestand die Fixpunkt-Bisimulation $\sim$ folglich nur aus trivial bisimularen Paaren.

9. Abschluss

Wir wollen nun die erarbeiteten Ergebnisse zusammenfassen und einen Ausblick für kommende Arbeiten geben, die auf dieser Ausarbeitung aufbauen können. Desweiteren werden aktuelle Forschungen aufgezählt, die von den entstandenen Paketen in Mathematica profitieren könnten.

9.1. Zusammenfassung

Wir beginnen mit der Betrachtung der Unteraufgaben und ihren entstandenen Lösungsansätzen um die Vollständigkeit der geforderten Lösungen zu belegen.

Aufgabe Ableitung von Labels, die die möglichen Interaktionen eines Graphen mit seiner Umgebung beschreiben

Lösung Mit der Implementierung des Borrowed Context Ansatzes im Rahmen der algebraischen Graphtransformation und Konstruktionen der adhäsiven Kategorie, können Labels für beschriftete Transitionssysteme berechnet werden. Der Quellcode der Technischen Universität Berlin wurde modifiziert und erweitert, so dass Borrowed Context Berechnungen sowohl mit, als auch ohne Pruning Optimierung vollständig berechnet werden können.

Aufgabe Überprüfung, ob eine gegebene Relation eine Bisimulation ist (mit Einsatz von Modulo-Techniken, vor allem Up-To-Context-Techniken)

Lösung Der Bisimulations-Check wurde für Graphtransformationssysteme in Mathematica realisiert und an Beispielen verifiziert. Hierfür existieren nun Methoden die für ein Graphenpaar überprüfen ob ein simulierender Schritt existiert und sämtliche Nachfolger berechnen. Ebenso existiert eine Methode um eine gegebene Relation mit Graphenpaaren auf die Eigenschaften einer Bisimulation zu untersuchen. Bei diesen Berechnungen ist manuell einstellbar wie die Borrowed Contexte gepruned sein sollen und ob Up-To-Context Techniken zu verifizierung verwendet werden sollen.

Aufgabe Ableitung einer Bisimulation, die ein gegebenes Paar von Graphen enthält (On-the-Fly-Technik, siehe [Hir99])

Lösung Mit dem Greedy-Algorithmus wurde eine Methode bereitgestellt die aus einem gegebenen Graphenpaar eine Bisimulation berechnet in der sämtliche Nachfolger enthalten sein sollen. Damit stellt der Greedy-Algorithmus eine Variante eines On-the-Fly-Algorithmus dar, der in seinem Aufbaukriterium jedoch anspruchsvoller ist.

Zusätzlich zu diesen Anforderungen an die Arbeit wurden weitere Knozepte Implementiert und Analysiert. Die wichtigsten darunter sind folgende:

- Generierung von zufälligen LTS mit frei wählbaren Grundparametern
- Implementierung von Adaptionen der entwickelten Bisimulations-Checks für LTS
- Umsetzung und Evaluation des Fixpunkt und Hirschkoff Algorithmus
- Erweiterung der Up-To-Context Methode zur Anwendung von Conditions
- Bereitstellung eines User-Manuals mit 13 Beispieldateien zur Einarbeitung

Die Ergebnisse dieser Ausarbeitung belegen zudem die Kongruenzresultate für Bisimulationen in Graphtransformationssystemen mittels Borrowed Context aus [EK04]. Die implementierten Methoden können in verschiedenen Forschungsbereichen genutzt werden. Sie können für die Verifikation von Modelltransformationen genutzt werden, wie sie mit expliziten Bisimulationskonstruktionen und Borrowed Context in [HKR$^+$10] beschrieben wurden. Die erweiterung der Up-To-Context Methode für Conditions kann für Reaktive Systeme mit Bedingungen eingesetzt werden welche in [RKE08] sowie [HK12] erläutert werden.

9.2. Ausblick

Für die Anwendung der Methoden für Bisimulationen die mit Bedingungen verknüpft sind, müssen Erweiterungen erarbeitet werden die Conditional-Shifts umsetzen. Desweiteren entstand während der Entstehungsphase dieser Ausarbeitung, ein Ansatz einer Hischkoff Variante die direkt auf den Graphtransformationenssystemen arbeitete. Aufgrund der hohen Speicherauslastung und dem unendlich großen Zustandsraum der entstehenden LTS selbst mit Up-To-Context wurde diese Methode bisher nicht vervollständigt. Es könnte eine weitere Methode entworfen werden, der die entstehenden Transitionen und Zustände aus Graphtransformationen konvertiert, so dass sie von der nun existierenden Hirschkoff Methode für LTS zur Berechnung von Bisimulationen genutzt werden könnte. Zuletzt kann an der Effizienz der vorgestellten Algorithmen geschraubt werden. Die Korrektheit der Algorithmen wurde gezeigt, was in dieser Ausarbeitung im Vordergrund stand. So ist es zusätzlich möglich die Laufzeiten der Algorithmen, für größere als die von uns betrachteten Transitionssysteme, zu verbessern.

Anhang A.

Quellcode Variablenbezeichnungen

Die Variablenbezeichnungen des Quellcodes folgen dem in Abbildung A.1 abgebildeten Borrowed Context Diagramm.

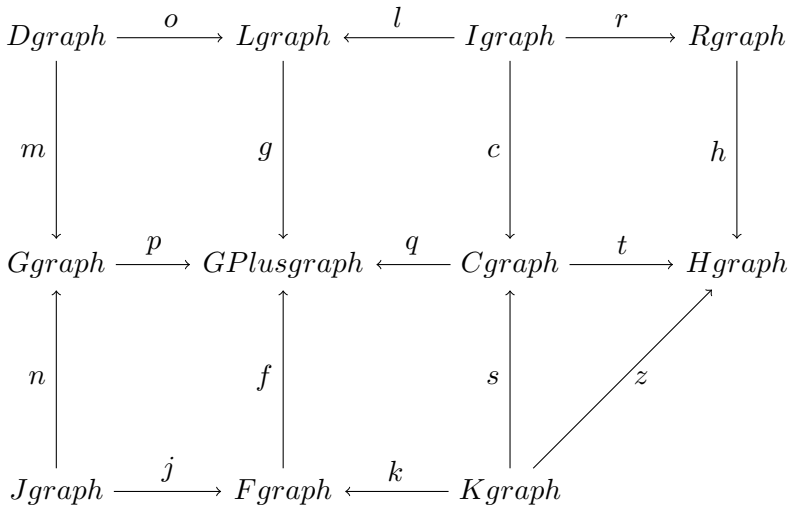

Abbildung A.1.: Borrowed Context Variablen

Um die im Quellcode beschriebenen Variablenbezeichnungen des Isomorphie-Checks besser zu verstehen ist zusätzlich Abbildung A.2 angegeben.

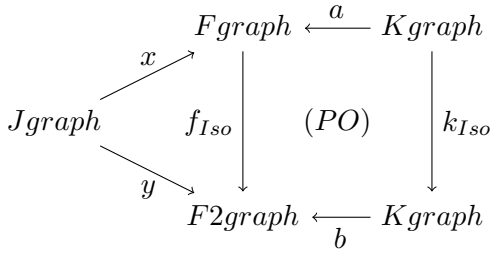

Abbildung A.2.: Isomorphismus Variablen

Für die UpToContext Berechnungen lauten die Variablennamen entsprechend der Abbildung A.3.

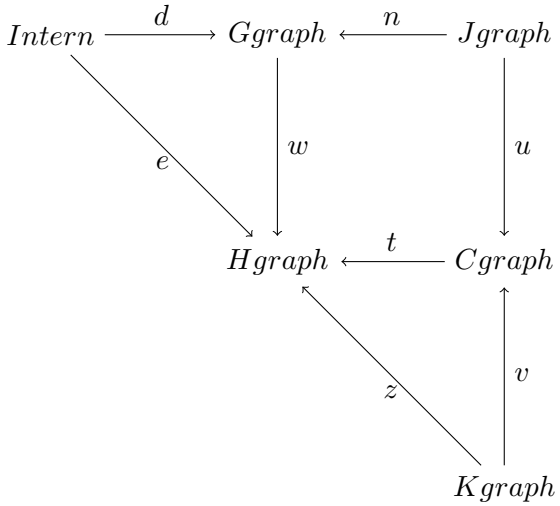

Abbildung A.3.: UpToContext Variablen

Anhang B.

Eidesstattliche Erklärung

Ich versichere an Eides Statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe. Alle Stellen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Ich versichere außerdem, dass ich keine andere als die angegebene Literatur verwendet habe. Diese Versicherung bezieht sich auch auf alle in der Arbeit enthaltenen Zeichnungen, Skizzen, bildlichen Darstellungen und dergleichen.

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

Duisburg, den 4. November 2012

Dennis Nolte

Literaturverzeichnis

[Abd98] Abdulrahim, Mohammad A.: *Parallel algorithms for labeled graph matching.* Golden, CO, USA, Colorado School of Mines, Diss., 1998

[BEK06] Baldan, Paolo ; Ehrig, Hartmut ; König, Barbara: Composition and Decomposition of DPO Transformations with Borrowed Context. In: *Proc. of ICGT '06 (International Conference on Graph Transformation)*, Springer, 2006, S. 153–167. – LNCS 4178

[BK08] Baier, C. ; Katoen, J.P.: *Principles of model checking.* The MIT Press, 2008. – 20–21 S. – ISBN 9780262026499

[CMR+97] Corradini, A. ; Montanari, U. ; Rossi, F. ; Ehrig, H. ; Heckel, R. ; Löwe, M.: Algebraic Approaches to Graph Transformation—Part I: Basic Concepts and Double Pushout Approach. In: Rozenberg, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations.* World Scientific, 1997, Kapitel 3

[EEPT06] Ehrig, H. ; Ehrig, K. ; Prange, U. ; Taentzer, G.: *Fundamentals of Algebraic Graph Transformation.* Springer Verlag, 2006. – ISBN 9783540311874

[EK04] Ehrig, Hartmut ; König, Barbara: Deriving Bisimulation Congruences in the DPO Approach to Graph Rewriting. In: *Proc. of FOSSACS '04*, Springer, 2004, S. 151–166. – LNCS 2987

[FM91] Fernandez, Jean-Claude ; Mounier, Laurent: "On the Fly" Verification of Behavioural Equivalences and Preorders. In: *CAV*, 1991, S. 181–191

[Hac03] Hack, Sebastian: *Graphersetzung für Optimierungen in der Codeerzeugung*, IPD Goos, Diplomarbeit, 12 2003. `http://www.info.uni-karlsruhe.de/papers/da_hack.pdf`

[Hir99] Hirschkoff, Daniel: On the Benefits of Using the Up-To Techniques for Bisimulation Verification. In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems.* London, UK, UK : Springer-Verlag, 1999 (TACAS '99). – ISBN 3–540–65703–7, 285–299

[HK12] Hülsbusch, Mathias ; König, Barbara: Deriving Bisimulation Congruences for Conditional Reactive Systems. In: *Proc. of FOSSACS '12*, Springer, 2012, S. 361–375. – LNCS/ARCoSS 7213

[HKR+10] Hülsbusch, Mathias ; König, Barbara ; Rensink, Arend ; Semenyak, Maria ; Soltenborn, Christian ; Wehrheim, Heike: Showing Full Semantics Preservation in Model Transformation – A Comparison of Techniques. In:

Proc. of iFM '10 (Integrated Formal Methods), Springer, 2010, S. 183–198. – LNCS 6396

[HM01] Hirshfeld, Yoram ; Moller, Faron: Pushdown automata, multiset automata, and Petri nets. In: *Theor. Comput. Sci.* (2001), S. 3–21

[Hül10] Hülsbusch, Mathias: Bisimulation Theory for Graph Transformation Systems. In: *Proc. of the ICGT '10 (Doctoral Symposium of the International Conference on Graph Transformation)*, Springer, 2010, S. 391–393. – LNCS 6372

[Mil99] Milner, Robin: *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999. – ISBN 9780521658690

[Par81] Park, David: Concurrency and automata on infinite sequences. In: Deussen, Peter (Hrsg.): *Theoretical Computer Science* Bd. 104. Springer Berlin / Heidelberg, 1981. – ISBN 978–3–540–10576–3, S. 167–183

[Res12a] Research, Wolfram: *Wolfram Mathematica 8*. `http://http://www.wolfram.com/mathematica/`, 2012

[Res12b] Research, Wolfram: *Wolfram Mathematica 8 Documentation Center*. `http://reference.wolfram.com/mathematica/guide/Mathematica.html`, 2012

[RKE08] Rangel, Guilherme ; König, Barbara ; Ehrig, Hartmut: Deriving Bisimulation Congruences in the Presence of Negative Application Conditions. In: *Proc. of FOSSACS '08*, Springer, 2008, S. 413–427. – LNCS 4962

[Sti95] Stirling, Colin: Lokal Model Checking Games. In: *CONCUR*, 1995, S. 1–11

[Tre08] Tretmans, Jan: Model Based Testing with Labelled Transition Systems. In: *Formal Methods and Testing*, 2008, S. 1–38

[TV05] Tabakov, Deian ; Vardi, Moshe Y.: Experimental Evaluation of Classical Automata Constructions. In: *LPAR* Bd. 3835, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–30553–X, S. 396–411