FAKULTÄT II
DEPARTMENT FÜR INFORMATIK
Abteilung Entwicklung korrekter Systeme

Entwurf und Implementierung von Algorithmen zur Berechnung von Petrinetz-Semantiken für Pi-Kalkül-Prozesse

17. Juni 2009

Bearbeitet von: Tim Strazny
Erstgutachter: Prof. Dr. Ernst-Rüdiger Olderog
Zweitgutachter: Prof. Dr. Eike Best
Betreut von: Dipl.-Inform. Roland Meyer
Anschrift: Tim Strazny, Bloherfelder Straße 109, 26129 Oldenburg
tim.strazny@informatik.uni-oldenburg.de

Zusammenfassung

Häufig enthalten Computerprogramme Fehler. In kritischen Fällen, bei denen die Abwesenheit von Fehlern im Sinne der Spezifikation sichergestellt werden muss, werden Programme mühevoll von Hand bewiesen oder Verfahren der automatischen Verifikation angewendet. Petrinetze, welche die Automatentheorie durch Nebenläufigkeit verallgemeinern, bieten sich zur automatische Verifikation an. Für verteilte und dynamisch rekonfigurierbare, d.h. mobile Systeme bietet sich mit dem π-Kalkül ein formales Modell an, welches mittels neuer Semantiken elegant in P/T-Petrinetze abgebildet werden kann. In der vorliegenden Arbeit werden Algorithmen präsentiert, die bestimmte neue Petrinetz-Semantiken von π-Kalkül-Prozessen berechnen. Mit diesem automatischen Übersetzen wird das Model-Checking von π-Kalkül-Prozessen anwendbar.

Oftentimes, computer programs are malfunctioning. Therefore, a sound functionality in terms of their specifications must be ensured in critical cases. For this purpose, programs are proven either manually, in an exhausting way, or by methods of automatical verification. Petri nets, which generalize the automata theory by concurrency, volunteer for automatic verification. In case of distributed and dynamically reconfigurable, i.e. mobile systems, the pi-calculus represents a formal model that can be mapped to P/T Petri nets with help of new semantics in a convenient way. This diploma thesis presents algorithms for computing certain new Petri net semantics of the π-calculus' processes. Thereby, model checking of pi-calculus processes becomes applicable.

Danksagung

Ich möchte mich ganz herzlich bei meinen Eltern Frauke Butz-Strazny und Klaus Strazny für die nicht enden wollende Unterstützung bedanken.
Verena Dickel möchte ich für ihre entscheidenden Worte und die Unterstützung danken.
Mein Dank gebührt auch Jörn Syrbe, der mich auch nachts mit seiner Anwesenheit und dem Gegenlesen an der Arbeit hielt.
Auch die Geduld, der Antrieb, der Rat, die Hilfestellungen und nicht zu vergessen, das ewige Korrekturlesen, meines Betreuers Roland Meyer sollen hier nicht unerwähnt bleiben und ich möchte mich ganz herzlich dafür bedanken.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **5**

2.1 Der π-Kalkül 5

2.1.1 Bemerkungen zum Sequenzoperator 9

2.2 Petrinetze 10

2.3 Eine strukturelle Petrinetz-Semantik 14

2.4 Eine sequentielle Petrinetz-Semantik 17

2.5 Kombination der Semantiken 20

2.6 Programmcode 21

3 Semantiken **23**

3.1 Berechnung der Semantiken 23

3.1.1 Strukturelle Semantik 24

3.1.1.1 Korrektheit 28

3.1.2 Sequentielle Semantik 29

3.1.2.1 Starke Faltung 34

3.1.2.2 Vergleich 37

3.2 Fragmente 40

3.2.1 Korrektheit und Aufwandsabschätzung 40

4 Restriktionsform **41**

4.1 Berechnung der strengen Restriktionsform 41

4.1.1 Flache Form 48

4.2 Korrektheit und Aufwandsabschätzung 50

4.3 Bemerkungen zur Implementierung 50

5 Reaktionen 51
5.1 Berechnung der Reaktionen . . . 51
5.1.1 Kommunikationspartner . . . 54
5.1.2 Echte Kommunikationen . . . 56
5.2 Korrektheit und Aufwandsabschätzung . . . 58

6 Strukturelle Kongruenz 59
6.1 Notwendige Bedingungen für strukturelle Kongruenz . . . 59
6.1.1 Freie Namen . . . 59
6.1.2 Anzahl von Präfixen, $\mathbf{0}, K\lfloor\tilde{a}\rfloor$ . . . 60
6.1.3 Anzahl gebundener Namen . . . 62
6.1.4 Reihenfolge von Präfixen . . . 64
6.2 Eine Ordnung auf Prozessen . . . 65
6.3 Prüfen von struktureller Kongruenz . . . 67
6.3.1 Substitutionen . . . 71
6.3.2 Präfixe . . . 73
6.3.3 Kommutative Operatoren . . . 75
6.4 Korrektheit und Aufwandabschätzung . . . 77
6.5 Bemerkungen zur Implementierung . . . 77

7 Überdeckungsgraph-Algorithmen 79
7.1 Aufbau eines Überdeckungsgraphen . . . 79
7.1.1 Komplexität . . . 80
7.1.2 Vorgehen . . . 81
7.1.3 Aufbau eines Überdeckungsgraphen . . . 81
7.1.4 Korrektheit . . . 85
7.1.5 Ausgaben . . . 85
7.1.6 Markieren überdeckter Zustände . . . 86
7.1.6.1 Korrektheit . . . 87
7.1.6.2 Auswahlreihenfolge . . . 88
7.1.7 Nebenläufigkeit . . . 88
7.1.8 Aufwandsabschätzung . . . 89
7.1.9 Sonderfall geschlossene Prozesse . . . 89
7.1.10 Bemerkungen zur Implementierung . . . 89

7.2 Feuerbarkeit von Transitionen 90
7.2.1 Rekursive Berechnung 90
7.2.2 Korrektheit 92
7.2.3 Aufwandsabschätzung 94
7.2.4 Bemerkungen zur Implementierung 94
7.3 Deadlocks 95
7.3.1 Eine hinreichende Bedingung für Deadlocks 95
7.3.2 Eine hinreichende Bedingung für Deadlockfreiheit 95
7.3.3 „Don’t know“ 96
7.4 Beschränktheit 97

8 Auffaltung zur Approximation der sequentiellen Semantik 99
8.1 Aufwandsabschätzung 105

9 Implementierung 107
9.1 Paket- und Klassenbeschreibung 107
9.2 Erweiterbarkeit 108
9.3 π-Grammatik 110
9.4 Compilereinstellungen 112
9.5 Programmaufruf 115

10 Zusammenfassung und Ausblick 119

Anhang I
Beispiele III
Abbildungsverzeichnis IX
Listings XI
Literaturverzeichnis XIII

1 Einleitung

Bei der Entwicklung von Computerprogrammen werden häufig Fehler übersehen, die dann zu unerwünschtem Verhalten der Programme führen. Insbesondere verteilte Systeme, die in verschiedenen Berechnungseinheiten Teilprobleme lösen und deren Interaktion eine Lösung des behandelten Problems darstellt, sind meist kompliziert. Jedes Teilproblem mag für sich betrachtet die Spezifikation erfüllen und damit korrekt sein, ihre Interaktion kann jedoch Fehler wie wechselseitige Abhängigkeiten enthalten.

Der π-Kalkül ist ein formales Modell einer besonderen Teilklasse verteilter Systeme, nämlich sogenannter dynamisch rekonfigurierbarer, kurz „mobiler“ Systeme. In dieser Systemklasse liegen nicht nur verschiedene Berechnungseinheiten verteilter Systeme vor, sie entstehen und verschwinden auch während der Ausführung, beziehungsweise ändern ihre Verbindungsstruktur. Neben den Problemen der Interaktion in gewöhnlichen verteilten Systemen bergen daher dynamisch rekonfigurierbare Systeme Fehler wie Kommunikation inkompatibler Berechnungseinheiten, das Ansprechen nicht mehr oder noch nicht vorhandener Partner oder die Überlastung einer Berechnungseinheit durch zu viele Partner.

Um sicherzustellen, dass ein Programm die gewünschte Funktionalität bietet, wurde bereits früh ein Vorgehen zum formalen Beweisen der Korrektheit von Algorithmen im Sinne ihrer Spezifikationen entwickelt.

Formales Beweisen ist schon für relativ kleine Programme mühsam, da unter anderem alle auftretenden Randfälle berücksichtigt werden müssen. Die Automatisierung solcher Beweise kann dabei Abhilfe schaffen.

Neben dem Theorembeweien und dem sogenannten Equational Reasoning bietet es sich an, ein Automatenmodell des Programms zu erstellen, welches die Zustandsübergänge des Programms und damit sein Ablaufverhalten wiederspiegelt. Auf dieses Modell werden Algorithmen angesetzt, die entscheiden, ob das Modell die gewünschten Eigenschaften

besitzt. Ist der Zusammenhang zwischen dem Modell und dem Programm stark genug, so gelten die Eigenschaften auch für das Programm.

Dass Programme häufig sehr viele Zustände annehmen können, die dann im entwickelten Automatenmodell enthalten sind, macht die Entscheidung, ob eine bestimmte Eigenschaft für das Modell gilt sehr aufwändig.

Für verteilte Systeme hat sich das spezielle Automatenmodell der Petrinetze durchgesetzt. Seit ihrer Einführung in den 1960er Jahren entwickelte sich schnell eine breite Theorie, mit welcher der Fokus von der Modellierung verteilter Systeme auf die Verifikation solcher Systeme verschoben wurde. Theoretisch betrachtet sind Petrinetze spezielle Graphen, auf denen Zustände und Zustandsübergänge definiert sind. Diese Graphstruktur kann ausgenutzt werden, um Eigenschaften für das Modell abzuleiten, ohne die Systemabläufe zu betrachten. Das macht automatisierte Verfahren im Allgemeinen für dieses Modell sehr schnell.

Mit ihrer Erforschung und der Weiterentwicklung der Petrinetztheorie entstand eine Vielzahl an Programmen, welche die meisten Petrinetz-bezogenen Aufgaben automatisierten.

Die Dissertation von Roland Meyer beschäftigt sich mit der Frage, wie man oben erwähnte dynamisch rekonfigurierbare Systeme verifiziert. Er schlägt dazu eine Übersetzung der π-Kalkül-Modelle in Petrinetze vor. Die Argumentation dabei ist, dass der π-Kalkül gut geeignet ist, die Systemklasse formal zu modellieren, während Petrinetze gut zur Verifikation gewünschter Eigenschaften geeignet sind.

Meyer stellt einige Grundanforderungen an eine zur Verifikation geeigneten Petrinetz-Semantik für π-Kalkül-Prozesse, nämlich

1. Retrievability, das Zurückgewinnen von Prozessen aus einem Zustand eines entsprechenden Petrinetzes,
2. Analysability, die Entscheidbarkeit möglichst vieler Eigenschaften des Systems,
3. Compositionality, die Berechnung der Semantik eines zusammengesetzten Prozesses, aus den Semantiken von Teilprozessen,
4. Finiteness, die Endlichkeit der erzeugten Petrinetze und
5. Expressiveness, eine möglichst große Klasse von Systemen soll übersetzt werden.

In der Literatur sind bereits viele Ansätze zur Übersetzung von π-Kalkül-Prozessen in Automaten zu finden. Sie verfolgen jedoch entweder nicht das Ziel der Verifikation, oder erfüllen die Grundanforderungen nicht.

- Engelfriet und Gelsema erzeugen unendliche Petrinetze mit ω-Kanten, um eine bestimmte strukturelle Kongruenz zu entscheiden.
- Busi und Gorrieri betrachten Zusammenhänge von Restriktionen, benutzen aber sogenannte Inhibitorkanten in den Petrinetzen
- Amadio und Meyssonnier beschäftigen sich mit der Klassifizierung von Prozessen
- Devillers, Klaudel und Koutny erzeugen, mit Verifikation als Zielsetzung, höhere Petrinetze, auf denen allerdings keine exakten Verfahren zur Verifikation existieren
- Montanari und Pistore übersetzen, ebenfalls mit der Absicht der Verifikation, Prozesse in HD-Automaten und spalten Prozesse dabei entlang der Parallelkomposition

In seiner Dissertation schlägt Meyer eine neue Übersetzung vor, die insbesondere der automatische Verifikation von Prozessen dient und für die bereits die meisten der Grundeigenschaften bewiesen wurden. Meyer verwendet einfache P/T-Petrinetze, um die Verwendung exakter Entscheidungsverfahren zu erlauben. Seine Übersetzungsvorschrift entscheidet sich durch die Zerlegung Prozesse entlang sequentieller Teilprozesse drastisch von allen bestehenden. [Mey07b] Diese Konstruktion erlaubt es, Verbindungsfragen, die in dynamisch rekonfigurierbaren Systemen wichtig sind, anhand der Graphstrukturen zu beantworten. Er zeigt in seiner Dissertation weitere „Analysability-Resultate", die andeuten, dass interessante Fragen für dynamisch rekonfigurierbare Systeme effizient in seiner neuen strukturellen Semantik bewiesen werden können.

Neben der obligatorischen Retrievability und der angesprochenen Analysability fordert Meyer, dass die Semantik für eine ausdrucksstarke Teilklasse von Prozessen endlich sein soll. Für seine strukturelle Semantik zeigt Meyer, dass sie genau für sogenannte strukturell stationäre Prozesse, solche aus endlich vielen Teilstrukturen, endlich wird.

Meyer vergleicht sein Endlichkeitsresultat mit den Semantiken von Engelfriet/Gelsama, beziehungsweise Amadio/Meyssonnier. Er stellt fest, dass diese für andere Teilklassen von Systemen endlich werden. Nach einer Anpassung ihrer Semantiken ist es Meyer möglich, dieses neue Endlichkeitkeitsresultat für die Semantiken von Engelfriet/Gelsa-

ma und Amadio/Meyssonnier zu zeigen. Weiter erkennt Meyer, dass sich seine und die Übersetzungsvorschriften von Engelfriet/Gelsama und Amadio/Meyssonier kombinieren lassen, was zu der derzeit größten Klasse an π-Kalkül-Prozessen führt, die eine endliche Petrinetz-Semantik haben. [Mey07a]

Eine Übersetzungsvorschrift verlangt eine Implementierung. Zum einen kann die Praxistauglichkeit, also die tatsächliche Analysability eines Netzes nur mit Werkzeugen festgestellt werden. In diesem Fall wird vom Werkzeug eine größere Fallstudie in ein Netz übersetzt. Zum anderen sind Semantiken kein Selbstzweck, sondern es sollen Systeme damit verifiziert werden. Man nimmt dem „Benutzer der Semantik“ die Arbeit, sie auszurechnen ab.

Die Definitionen von Meyer sind rein deklarativ, es wird keine prozedurale Funktion angegeben. Diese Arbeit übernimmt diesen Teil und enthält folgende wichtige Ergebnisse:

- Eine elegante Definition des Sequenzoperators
- Die Vereinfachung der Entscheidung struktureller Kongruenz mittels einer Ordnung auf Prozessen
- Eine Vereinfachung eines Überdeckungsgraphen zur Berechnung von Kommunikationspartnern
- Das Auslagern der Berechnung von Kommunikationspartnern in einen eigenen Thread
- Eine Alternative zu Meyers sequentieller Semantik
- Kombination der strukturellen und der sequentiellen Semantik auf eine benutzerfreundlichere Weise als bei Meyer

2 Grundlagen

Im Folgenden werden die wichtigsten Grundlagen dieser Arbeit vorgestellt.

2.1 Der π-Kalkül

Es gibt einige Varianten von Milners grundlegender Definition des π-Kalküls, die die Verwendung erleichtern sollen. In dieser Arbeit wird eine polyadische Variante des π-Kalküls mit sogenannten Wächtern, einer parametrisierten Rekursion an Stelle der Replikation benutzt und zusätzlich das Konzept der Sequenz direkt verwendet (vgl. [Mil99, Mey07b]).

Diese Variante kann in einem syntaktisch einfacheren monadischen π-Kalkül, der außerdem den Sequenzoperator nicht benutzt, ausgedrückt werden und stellt somit lediglich den üblichen „syntactic sugar" dar. Nichtsdestotrotz erleichtern höhere Operatoren und das Versenden und Empfangen von beliebig vielen Namen über einen Kanal, statt nur einem beim monadischen Kalkül, wie auch die Einführung von Wächtern nicht nur die Arbeit des Anwenders, sondern auch des hier entwickelten Programms. Zwar sind mehr Fälle zu behandeln, aber grundsätzlich weniger Arbeit beim Übersetzen zu tun. Auch die erzeugten Netze werden bei Verwendung der höheren Konstrukte kleiner, als würden sie erst in den syntaktisch ärmeren Kalkül überführt.

Intuitiv rührt dieser Sachverhalt daher, dass höhere Konstrukte, ein größeres Maß an Semantik tragen. Beispielsweise lässt sich am Prozess $\nu a.(\nu b.\overline{a}\langle b\rangle.\overline{b}\langle c\rangle.\overline{b}\langle d\rangle.P \mid Q \mid R)$ nicht erkennen, ob der erste Teilprozess eine Reihe von Namen über b senden kann. Schließlich könnte Q nur einen Namen empfangen. Lautet der Prozess aber $\nu a.(\overline{a}\langle c, d\rangle.P \mid Q \mid R)$, so ist klar, dass genau zwei Namen versendet und empfangen werden müssen. Im ersten Fall drei Reaktionsschritte nötig, um die drei Namen zu versenden, während im zweiten Fall nur ein Schritt getan wird.

Erfreulicherweise lässt sich die in [Mey07b] vorgestellte Semantik, die sich auf den monadischen π-Kalkül mit parametrisierter Rekursion bezieht, kanonisch auf den oben beschriebenen Kalkül fortsetzen.

Definition 1 gibt die formale Syntax des hier verwendeten Kalküls an. Für eine (stets endliche) Sequenz von Namen $a_1, \ldots, a_n$ wird auch $\tilde{a}$ geschrieben, allerdings wird diese Sequenz bei Restriktionen $\nu\tilde{a}$ von Zeit zu Zeit auch als Menge aufgefasst. Dies ist später durch die strukturelle Kongruenz gerechtfertigt, die aussagt, dass die Reihenfolge zweier Restriktionen vertauscht werden darf.

Definition 1 (Der π-Kalkül)**.** Die Menge der π-Kalkül-Prozesse $\mathcal{P}$ ist wie folgt definiert:

$$
\begin{array}{rcl}
\pi & ::= & \overline{x}\langle\tilde{y}\rangle \mid x(\tilde{y}) \mid \tau \mid [a = b].\pi \\
P & ::= & \sum_{i \in I} \pi_i.P_i \mid P_1|P_2 \mid P_1;P_2 \mid \nu a.P \mid K\lfloor\tilde{a}\rfloor
\end{array}
$$

Wobei I eine endliche Indexmenge ist und die leere Summe mit $\mathbf{0}$ beschrieben wird.

In Bezug auf Algorithmen, die Prozesse des Kalküls abarbeiten, wird hier häufig von dieser strengen Syntax abgewichen, indem Klammern eingespart werden und für die Kompositionsoperatoren $+, ;, |$ mehr als nur zwei Operanden erlauben.

Der im Folgenden eingeführten Begriff von Prozesskontexten wird unter anderem zur Vereinfachung von Algorithmen verwendet.

Definition 2 (Prozesskontext $\mathcal{C}$)**.** Ein Prozesskontext ist ein Prozess mit einem Platzhalter $[\,]$, für den ein anderer Prozess eingesetzt wird. Das Ergebnis des Einsetzens eines Prozesses P, $\mathcal{C}[P]$, ist ein Prozess ohne Platzhalter $[\,]$. Prozesskontexte werden durch folgende Syntax beschrieben:

$$
\mathcal{C} ::= [\,] \mid \pi.\mathcal{C} + M \mid \nu a.\mathcal{C} \mid \mathcal{C}|P \mid P|\mathcal{C} \mid \mathcal{C};P \mid P;\mathcal{C}
$$

Grundlegend für den π-Kalkül ist das Konzept der Namen. Die Namen eines Prozesses zerfallen in zwei Mengen, die wie folgt definiert sind:

Definition 3 (Gebundene und freie Namen)**.** Ein in einem Prozess enthaltener Name a ist genau dann gebunden, wenn er in einer Restriktion νa auftaucht, oder über einen Kanal empfangen wird $x(a)$. Die Menge der gebundenen Namen eines Prozesses P wird mit $bn(P)$ bezeichnet. Ein im Prozess enthaltener Name ist genau dann frei, wenn er

nicht gebunden ist. Die Menge der freien Namen eines Prozesses P wird mit $fn(P)$ bezeichnet.

Enthält ein Prozess keine freien Namen, so nennen wird er „geschlossen“ genannt. Die Menge der geschlossenen Prozesse wird mit $\mathcal{CP}$ bezeichnet.

Ohne Einschränkung wird davon ausgegangen, dass Prozesse keine doppelt gebundenen Namen haben (siehe Definition 3) und dass die freien Namen eines Teilprozesses nicht in einem anderen Teilprozess gebunden sind. Mit Definition 2 lässt sich dies wie folgt formalisieren: Seien $P \in \mathcal{P}$ und $Q = \nu\tilde{a}.Q'$ oder $Q = x(\tilde{a}).Q'$, es soll dann gelten:

$$\forall \mathcal{C}[Q] = P : \tilde{a} \cap bn(Q') = \varnothing \wedge ((\exists \mathcal{C}[R] = P : \tilde{a} \cap fn(R) \neq \varnothing) \Rightarrow \exists \mathcal{C}[R] = Q')$$

Mit anderen Worten: wird a genau in Q gebunden, so wird es in keinem Teilprozess von Q gebunden und ist a frei in R, so ist R ein Teilprozess von Q

Folgende Definition wird benötigt, um die strukturellen Kongruenz zu erhalten.

Definition 4 (Substitution $\{b/a\}$)**.** Die Substitution $\{b/a\}P$ eines gebundenen Namens a durch b innerhalb eines Prozesses P entspricht einem textuellen Ersetzen von a durch b. Sie wird einerseits im Rahmen der strukturelle-Kongruenz-erhaltenen (vgl. Definition 5) α-Konversion benutzt, wobei b dann nicht in $fn(P) \cup bn(P)$ ist. Andererseits wird sie, ohne das b einzuschränken, bei dem Empfangen von Namen eingesetzt.

Kanonisch wird die Substitution auf Sequenzen von Namen fortgesetzt, falls alle a_i in $\tilde{a} = a_1, \ldots, a_n$ für die Substitution $\{\tilde{b}/\tilde{a}\}$ verschieden sind.

Die strukturelle Kongruenz ist das zentrale Hilfsmittel im π-Kalkül, über den Kongruenzklassen von Prozessen konstruiert und das Verhalten von Prozessen definiert werden können. In der hier verwendeten Variante des Kalküls gibt es, wie erwähnt, keine Replikation sondern parametrisierte Rekursion, deren Aufruf einen Reaktionsschritt fordert. Mit dieser Definition ist die strukturelle Kongruenz entscheidbar [Mey07b]. Dies ist für die Umsetzung der Algorithmen notwendig.

Definition 5 (Strukturelle Kongruenz)**.** Strukturelle Kongruenz, $\equiv$, ist eine reflexive, transitive und symmetrische Kongruenzrelation auf $\mathcal{P}$, die folgende Regeln erfüllt:

- α-Konversion gebundener Namen

$$\nu a.P \equiv \nu b.\{b/a\}P, \text{ wenn } b \notin fn(P) \cup bn(P) \setminus \{a\}$$

- Kommutativität von Auswahl- und Paralleloperator

$$P_1 + \cdots + P_k + P_{k+1} + \cdots + P_n \equiv P_{k+1} + \cdots + P_n + P_1 + \cdots + P_k$$

$$P|Q \equiv Q|P$$

- Assoziativität der Parallel- und Sequenzoperatoren

$$P|(Q|R) \equiv (P|Q)|R \qquad P;(Q;R) \equiv (P;Q);R$$

- **0**-Prozess als neutrales Element bei Sequenz- und Paralleloperator

$$P|\mathbf{0} \equiv P \qquad P;\mathbf{0} \equiv P \qquad \mathbf{0};P \equiv P$$

- Neutralität erfüllter Wächter

$$[a = b].P \equiv P, \text{ falls } a = b$$

- Neutralität wirkungsloser Restriktionen

$$\nu x.\mathbf{0} \equiv \mathbf{0}$$

- Umsortieren von Restriktionen

$$\nu x.\nu y.P \equiv \nu y.\nu x.P$$

- Verändern der Sichtbarkeit einer Restriktion (Scope Extrusion)

$$\nu x.(P|Q) \equiv P|(\nu x.Q), \text{ wenn } x \notin fn(P)$$

$$\nu x.(P;Q) \equiv P;(\nu x.Q), \text{ wenn } x \notin fn(P)$$

$$\nu x.(P;Q) \equiv (\nu x.P);Q, \text{ wenn } x \notin fn(Q)$$

Das Verhalten eines π-Kalkül-Prozesses wird durch die sogenannte Reaktionsrelation festgelegt.

Definition 6 (Reaktionsrelation)**.** Die Reaktionsrelation $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ ist durch folgende Regeln definiert.

$$\text{Tau: } \tau.P + M \rightarrow P$$

$$\text{React: } (\overline{a}\langle x_1, \ldots, x_n\rangle.P + M) \mid (a(y_1, \ldots, y_n).Q + N) \rightarrow P \mid \{\tilde{x}/\tilde{y}\}Q$$

$$\text{Const: } K\lfloor \tilde{a} \rfloor \rightarrow \{\tilde{a}/\tilde{x}\}P, \text{ wenn } K(\tilde{x}) := P$$

$$\text{Seq: } \frac{P|Q \rightarrow P'|Q'}{(P;R) \mid (Q;S) \rightarrow (P';R) \mid (Q';S)}$$

$$\text{Par: } \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q} \qquad \text{Res: } \frac{P \rightarrow P'}{\nu x.P \rightarrow \nu x.P'}$$

$$\text{Struct: } \frac{P \rightarrow P'}{Q \rightarrow Q'}, \text{ wenn } P \equiv Q \text{ und } P' \equiv Q'$$

Nicht unwichtig ist, dass $\{P' \mid P \rightarrow P'\}$ in endlich viele Kongruenzklassen unter der strukturellen Kongruenz zerfällt und es damit nur endlich viele Reaktionen eines Prozesses gibt deren Ergebnisse nicht strukturell kongruent sind.

Definition 7 (Echte Kommunikation)**.** Eine Reaktion $P|Q \rightarrow R$ wird echte Kommunikation genannt, falls

$$\forall P \rightarrow P' \forall Q \rightarrow Q' : P|Q' \not\equiv R \not\equiv P'|Q,$$

also eine Kommunikation zwischen P und Q stattgefunden hat. Beide Prozesse haben dann einen Präfix konsumiert.

2.1.1 Bemerkungen zum Sequenzoperator

Da die Definition des in dieser Arbeit eingeführten Sequenzoperators von der Definition in [Mil99, Bsp. 5.27, S. 49ff] abweicht und entsprechende Beweise noch ausstehen, ist er als experimentell anzusehen. Die Algorithmen und Beweise in dieser Arbeit nehmen an, dass die Definition sinnvoll ist, benötigen den Operator aber nicht.

Das Verhalten des Sequenzoperators ist intuitiv und nah an den gängigen Regeln des π-Kalküls gehalten: In einer Sequenz von Prozessen kann nur der erste Prozess reagieren und der Nullprozess stellt ein neutrales Element des Sequenzoperators dar. In einem Prozess $P;Q$ kann Q also erst dann reagieren, wenn P zu $\mathbf{0}$ übergegangen ist. Der Übergang zu $\mathbf{0}$ entspricht intuitiv der vollständigen Terminierung eines Prozesses. Ist P

eine Parallelkomposition von Prozessen, so kann er nur zu **0** übergehen kann, wenn jeder parallele Prozess seinerseits zu **0** übergegangen ist.

Damit wird das Problem der verteilten Terminierung elegant gelöst, indem geeignete Regeln struktureller Kongruenz an Stelle des traditionellen Vorgehens mittels expliziter Terminierungsaktionen verwendet werden. Diese Regeln stellen kanonische Erweiterungen der strukturellen Kongruenz dar.

Die Komposition ist noch nicht vollständig untersucht, scheint aber durchaus die für einen Sequenzoperator gewünschten Eigenschaften zu erfüllen.

2.2 Petrinetze

Seit Carl Adam Petri 1962 in seiner Dissertation [Pet62] das Konzept der Petrinetze vorstellte entwickelte sich eine breite Theorie um dieses zugängliche und zugleich mathematisch begründete Modell nebenläufiger Systeme.

Definition 8 (P/T-Petrinetz)**.** Ein Petrinetz mit Anfangsmarkierung ist ein 4-Tupel $N = (S, T, F, M_0)$, wobei $S \cap T = \varnothing$ und S die Menge der Stellen, T die Menge der Transitionen, $F : S \times T \cup T \times S \to \mathbb{N}$ eine totale Kantengewichtsfunktion und $M_0 \in \mathbb{N}^S$ die Anfangsmarkierung sind. Elemente in $\mathbb{N}^S$ werden Zustände oder Markierungen von N genannt.

Ist $M(s) = k$ für einen Zustand M, so sagen wir, dass Stelle s mit genau k Marken (auch Token) belegt ist.

Grafisch werden Stellen als Kreise, Transitionen als Rechtecke und Kanten als Pfeile dargestellt.

In Abbildung 2.1 wird ein einfaches Petrinetz dargestellt.

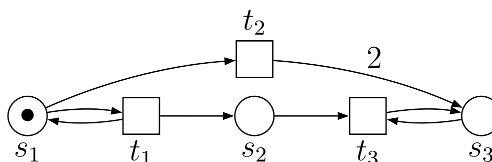

Abbildung 2.1: Ein Petrinetz

Um das Verhalten von Petrinetzen festzulegen, ist der Begriff der Aktivierung notwendig.

Definition 9 (Aktivierung)**.** Sei $N = (S, T, F, M_0)$ ein Petrinetz, $M \in \mathbb{N}^S$, $t \in T$. Zustand M aktiviert t, geschrieben $M\,[t\rangle$, genau dann, wenn $M \geq F(\bullet, t)$.

Wenn ein Zustand eine Transition aktiviert, so kann die Transition den Zustand des Netzes ändern. Definition 10 führt den Begriff des Feuerns ein, bei dem eine Transition Marken auf gewissen Stellen konsumiert und Marken auf Stellen produziert.

Definition 10 (Transitionsrelation)**.** Sei $N = (S, T, F, M_0)$ ein Petrinetz. Relation $[\cdot\rangle \subseteq \mathbb{N}^S \times T \times \mathbb{N}^S$, auch als $\dot{\rightarrow}$ geschrieben, ist definiert durch:

$$M\,[t\rangle\,M' \Leftrightarrow M\,[t\rangle \wedge M'(s) = M(s) - F(s,t) + F(t,s).$$

Für $M\,[t\rangle\,M'$ wird auch „aus M feuert t nach M'“ gesagt.

Ist es unerheblich, mit welchen Transitionen zwei Zustände M, M' in der Relation sind, so wird auch $M \rightarrow M'$ geschrieben.

Nun wird die Transitionsrelation auf Folgen von Transitionen verallgemeinert.

Definition 11 (Feuersequenz)**.** Sei $N = (S, T, F, M_0)$ ein Petrinetz, $M, M', M'' \in \mathbb{N}^S$ und $\sigma \in T^*$, mit $\sigma' \in T^\omega$ unendlich. Aktiviertsein und Feuern wird kanonisch auf Sequenzen von Transitionen ausgeweitet. Dabei gelte $M\,[\epsilon\rangle\,M$, sowie $M\,[t\sigma\rangle\,M'' \Leftrightarrow M\,[t\rangle\,M' \wedge M'\,[\sigma\rangle\,M''$. Es wird gesagt, dass M das σ' aktiviert, wenn jeder endliche Präfix von σ' in M aktiviert ist.

Ist es gleichgültig, mit welcher Transitionsfolge ein Netz von einem Zustand M nach Zustand M' kommt, so wird im Sinne des reflexiven transitiven Abschlusses $M \rightarrow^* M'$ geschrieben.

Über die Transitionsrelation kann nun in Definition 12 festgehalten werden, welche Zustände ein System annehmen kann.

Definition 12 (Erreichbarkeitsmenge)**.** Sei N ein Petrinetz und M ein Zustand. Die Erreichbarkeitsmenge $[M\rangle$ ist definiert als $\{M' \mid \exists \sigma \in T^* : M\,[\sigma\rangle\,M'\}$.

Weitere wichtige Konzepte sind die des Vor- und Nachbereichs, welche die Vorgänger, bzw. Nachfolgermenge eines Elements in Bezug auf eine Relation beschreiben. Sie werden hier hauptsächlich für Transitionen und Stellen betrachtet.

Definition 13 (Vor- und Nachbereich)**.** Zu Stelle $s \in S$ eines Petrinetzes (S, T, F, M_0) werden Vorbereich ${}^\bullet s := \{t \in T \mid F(t, s) \neq 0\}$ und Nachbereich $s^\bullet := \{t \in T \mid F(s, t) \neq 0\}$ definiert. Analog dazu seien Vor- und Nachbereich für Transitionen definiert.

Bei der Betrachtung von Petrinetzen, wie auch bei der Beschreibung von Algorithmen, ist der Begriff der Multimenge nützlich, der hier kurz angerissen wird.

Definition 14 (Multimenge)**.** Eine Multimenge ist eine Funktion von einer Grundmenge in die Menge der ganzen Zahlen: $m : \mathbb{Z}^X$ und lässt sich ebenso als Vektor in $\mathbb{Z}^X$ verstehen. Die Summe und Differenz zweier Multimengen mit derselben Grundmenge ist definiert als

$$(m + n)(x) := m(x) + n(x), \qquad (m - n)(x) := m(x) - n(x)$$

Eine alternative Schreibweise ist die des aus Programmiersprachen bekannten „Bag“s. Dabei wird eine Art Mengenschreibweise benutzt, bei der Elemente mehrfach und mit negativem Vorzeichen vorkommen dürfen.

Vor der Einführung des für bestimmte Erreichbarkeitsprobleme praktischen Begriffs des Überdeckungsgraphen wird definiert, was ein Graph ist.

Definition 15 (Graph)**.** Ein gerichteter, kantenbeschrifteter Graph G ist das Tupel (V, E, L), mit Knotenmenge V, Kantenmenge $E \subseteq V \times L \times V$ und Kantenbeschriftungsalphabet L. Enthält E ein Tupel (a, x, b), so wird für diese mit x beschriftete Kante von Knoten a nach Knoten b auch $a \xrightarrow{x} b$ geschrieben. Mit $\rightarrow^*$ ist die reflexive transitive Hülle von $\rightarrow$ bezeichnet. Wenn es nötig ist, zu verdeutlichen, zu welchem Graphen die Kantenmenge gehört, wird $\rightarrow_G$ oder $\rightarrow_E$ geschrieben.

Wie gesehen werden kann ist Petrinetz (S, T, F, M_0) ein Graph $(S \cup T, F, \mathbb{N})$, so dass die speziell für Petrinetzelemente eingeführten Begriffe der Vor- und Nachbereiche auf allgemeine Graphen erweitert werden können:

Definition 16 (Vor- und Nachbereiche für allgemeine Graphen)**.** Sei (V, E, L) ein Graph und $v \in V$. Dann ist der Vorbereich von v als ${}^\bullet v := \{v' \in V \mid \exists l \in L : (v, l, v') \in E\}$ und der Nachbereich von v als $v^\bullet := \{v' \in V \mid \exists l \in L : (v', l, v) \in E\}$ definiert.

In Überdeckungsgraphen sind die Knoten sogenannte verallgemeinerte Zustände, die mit folgender Definition eingeführt werden.

Definition 17 (Verallgemeinerter Zustand)**.** Sei $N = (S, T, F, M_0)$ ein Petrinetz. Die Menge $\mathbb{N}_\omega$ sei definiert als $\mathbb{N} \cup \{\omega\}$, wobei $\forall a \in \mathbb{N} : \omega + a = \omega - a = \omega$. Es gilt dabei $a \in \mathbb{N} \Rightarrow a < \omega$, so dass das ω für „beliebig groß“ oder „unendlich“ steht. Ein verallgemeinerter Zustand von N ist ein Vektor $M \in \mathbb{N}_\omega^S$. Ordnungen und Operationen werden auf verallgemeinerte Zustände kanonisch fortgesetzt.

Für verallgemeinerte Zustände wird eine weitere Halbordnung $\leq_\omega$ definiert, die wiedergibt, dass ein verallgemeinerter Zustand a spezifizierter ist, als b, d.h. weniger ω-Knoten enthält und sonst gleich ist, [PW02]:

$$a \leq_\omega b \Leftrightarrow \forall 1 \leq i \leq n : (b_i = \omega \vee a_i = b_i).$$

Im Gegensatz zum Vorgehen in [PW02, Def. 3.2.26, S. 66 ff], bei dem ein Überdeckungsgraph direkt über seine Konstruktion definiert wird, wird hier eine Definition über die grundlegenden Eigenschaften eines Überdeckungsgraphen gewählt.

Definition 18 (Überdeckungsgraph)**.** Zu einem Petrinetz $N = (S, T, F, M_0)$ ist der gerichtete, kantenbeschriftete Graph $Cov(N) = (V, E, T)$ ein Überdeckungsgraph, wenn $Cov(N)$ zusammenhängend und $V \subseteq \mathbb{N}_\omega^S$ endlich ist, sowie folgende Bedingungen erfüllt werden:

1. $Cov(N)$ ist lokal determiniert: $\forall M \in V \forall t \in T : |E \cap \{(M, t, x) \mid x \in V\}| \leq 1$
2. Jeder Feuersequenz in N entspricht ein Pfad in $Cov(N)$:
   Seien $\sigma \in T^*, M_s \in [M_0\rangle_N$ und $M \in \mathbb{N}^S$ mit $M_s\, [\sigma\rangle_N\, M$, dann gilt:

   $$\exists M_s', M' \in \mathbb{N}_\omega : M_s \leq_\omega M_s' \wedge M \leq_\omega M' \wedge M_0 \rightarrow^*_{Cov(N)} M_s' \wedge M_s' \xrightarrow{\sigma}_{Cov(N)} M'.$$

3. Jedem Pfad in $Cov(N)$ entspricht eine Feuersequenz in N:
   Seien $\sigma \in T^*, M_s, M \in V$ mit $M_0 \rightarrow^*_{Cov(N)} M_s$ und $M_s \xrightarrow{\sigma}_{Cov(N)} M$, dann gilt:

   $$\exists M_s' \in [M_0\rangle_N, M' \in \mathbb{N}^S : M_s' \leq_\omega M_s \wedge M' \leq_\omega M \wedge M_s'\, [\sigma\rangle_N\, M'.$$

An einem Überdeckungsgraphen kann abgelesen werden, ob eine Transition des zugrundeliegenden Netzes feuerbar, also jemals aktiviert, oder sogar beliebig oft feuerbar ist.

Definition 19 (Beliebig oft feuerbar)**.** Eine Transition t heißt in Zustand M beliebig oft feuerbar, genau dann, wenn

$$\forall k \in \mathbb{N} \exists \sigma_1, \ldots, \sigma_n \in T^* : M\, [\sigma_1 t \sigma_2 t \ldots \sigma_n t\rangle$$

(Dies entspricht dem Begriff der 2-Lebendigkeit von Transitionen in [PW02, Def. 3.3.12, S.86])

Auch lässt sich ablesen, ob Stellen beschränkt sind.

Definition 20 (Beschränktheit)**.** In einem Petrinetz $N = (S, T, F, M_0)$ heißt eine Stelle s k-beschränkt, wenn für alle $\forall M \in [M_0\rangle : M(s) \leq k$. Ein Petrinetz heißt k-beschränkt, wenn alle seine Stellen k-beschränkt sind.

Zustände, in den keine Transition aktiviert ist, werden Deadlocks genannt.

Definition 21 (Deadlock)**.** Ein Zustand M ist Deadlock eines Netzes N, genau dann, wenn $\forall t \in T : \neg(M\,[t\rangle)$ gilt, in M also keine Transition aktiviert ist.

2.3 Eine strukturelle Petrinetz-Semantik

Die in [Mey07b] eingeführte Petrinetz-Semantik wird hier kurz „strukturelle Semantik" genannt. Sie zu berechnen stellt den Hauptbeitrag der vorliegenden Arbeit dar.

Bevor die eigentliche Definition der Semantik vorgestellt werden kann, müssen einige Standardformen von Prozessen eingeführt werden, für die sich bestimmte Eigenschaften herleiten lassen.

Die einfachste hier benutzte Standardform ist die, ohne „überflüssige" Teile, also solche, die unter Erhalt struktureller Kongruenz „verschwinden" dürfen.

Definition 22 (Standardform $\mathcal{P}_\mathbf{0}$)**.** Die Menge $\mathcal{P}_\mathbf{0} \subseteq \mathcal{P}$ enthält ausschließlich Prozesse $P_\mathbf{0}$ mit folgender Eigenschaft:

$$\forall P \in \mathcal{P} : P \equiv P_\mathbf{0} \Rightarrow (|bn(P_\mathbf{0})| \leq |bn(P)| \wedge |\#_\mathbf{0}(P_\mathbf{0})| \leq |\#_\mathbf{0}(P)|)$$

Dabei ist $\#_\mathbf{0}(-)$ die später benötigte Funktion aus Definition 34, welche die Anzahl der enthaltenen **0**-Prozesse zählt. Prozesse aus $\mathcal{P}_\mathbf{0}$ enthalten ausschließlich gebundene Namen, die auch tatsächlich „benutzt" werden und sie enthalten keine „überflüssigen" **0**-Prozesse.

Zentral für die Semantik ist der Begriff der Fragmente, der über die folgende wichtige Standardform definiert ist.

Definition 23 (Restriktionsform $\mathcal{P}_\nu$)**.** Die Menge der Prozesse in Restriktionsform $\mathcal{P}_\nu \subseteq \mathcal{P}$ enthält ausschließlich Prozesse der folgenden Form:

$$P_\nu = \sum_{i \in I} \pi_i.P_i \mid K\lfloor\tilde{a}\rfloor \mid \nu a.(P_{\nu 1}|\ldots|P_{\nu n}) \mid \nu a.(Q_{\nu 1};\ldots;Q_{\nu m}) \mid P_{\nu 1}|P_{\nu 2} \mid P_{\nu 1};P_{\nu 2},$$

mit $a \in fn(Q_{\nu 1}) \cap fn(Q_{\nu m}) \wedge \forall 1 \leq i \leq n : a \in fn(P_{\nu i})$.

Sei $P_\nu = \prod_{i=1}^{n} P_{\nu i}$, wobei der oberste Operator von $P_{\nu i}$ nicht die Parallelkomposition $|$ ist. $Frag(P_\nu) := \{P_{\nu i} \neq \mathbf{0} \mid 1 \leq i \leq n\}$ bezeichnet die Fragmente von P_ν. Die Menge aller Fragmente ist $\mathcal{P}_\mathcal{F} \subseteq \mathcal{P}_\nu$ [Mey07b].

Wie in [Mey07b] festgestellt wird, gibt es zu jedem Prozess einen strukturell kongruenten in Restriktionsform. Die rekursive Funktion ν stellt eine entsprechende Abbildung dar:

Definition 24 ($\nu : \mathcal{P} \to \mathcal{P}_\nu$)**.** Funktion $\nu : \mathcal{P} \to \mathcal{P}_\nu$ ist folgendermaßen rekursiv definiert:

$$\begin{aligned}
\nu(\textstyle\sum_{i\in I} \pi_i.P_i) &:= \textstyle\sum_{i\in I} \pi_i.P_i \\
\nu(K\lfloor\tilde{a}\rfloor) &:= K\lfloor\tilde{a}\rfloor \\
\nu(P_1 \mid P_2) &:= \nu(P_1) \mid \nu(P_2) \\
\nu(P_1;P_2) &:= \nu(P_1);\nu(P_2) \\
\nu(\nu a.(P_1;P_2)) &:= \begin{cases} \nu a.\nu(P_1);P_2, & \text{wenn } a \notin fn(P_2) \\ P_1;\nu a.\nu(P_2), & \text{wenn } a \notin fn(P_1) \\ \nu a.(\nu(P_1);\nu(P_2)), & \text{sonst} \end{cases} \\
\nu(\nu a.P) &:= \nu a.(\textstyle\prod_{j\in I_a} F_j) \mid \textstyle\prod_{i\in I\setminus I_a} F_i,
\end{aligned}$$

wobei $\nu(P) = \prod_{i \in I} F_i$ mit $I = \{1, \ldots, n\}$ und $a \in fn(F_j)$ genau dann, wenn $j \in I_a \subseteq I$, [Mey07b].

In [Mey07b] wird gezeigt, dass $\nu(-)$ einem Prozess P einen strukturell kongruenten Prozess in $\mathcal{P}_\nu$ zuordnet.

Die Standardform der Prozesse in $\mathcal{P}_\nu$ lässt sich weiter zur strengen Restriktionsform verschärfen, welche der „cell normal form“ in [EG04] entspricht. Dort ist sie allerdings ein technisches Hilfsmittel für einen Beweis der Entscheidbarkeit von struktureller Kongruenz für eine andere Variante des Kalküls.

Definition 25 (Strenge Restriktionsform $\mathcal{P}_{s\nu}$)**.** Die Menge der Prozesse in $\mathcal{P}_{s\nu} \subseteq \mathcal{P}_{\nu} \cap \mathcal{P}_{\mathbf{0}}$ enthält ausschließlich Prozesse der folgenden Form:

$$\begin{aligned} P_{s\nu} \;=\; & \textstyle\sum_{i\in I} \pi_i.P_{s\nu i} \mid K\lfloor \tilde{a} \rfloor \mid \nu a.(P_{s\nu 1}|\dots|P_{s\nu n}) \mid \nu a.(Q_{s\nu 1};\dots;Q_{s\nu m}) \\ & \mid P_{s\nu 1}|P_{s\nu 2} \mid P_{s\nu 1};P_{s\nu 2}, \end{aligned}$$

wobei $a \in fn(Q_{s\nu 1}) \cap fn(Q_{s\nu m}) \wedge \forall 1 \leq i \leq n : a \in fn(P_{s\nu i})$.

Es lassen sich genau wie in Definition 23 Fragmente identifizieren. Einziger Unterschied ist, dass alle Teilprozesse eines Prozesses in $\mathcal{P}_{s\nu}$ ebenfalls in $\mathcal{P}_{s\nu}$ liegen.

Um die Anzahl der Vorkommen von Fragmenten in einem Prozess zu zählen, wurde eine Funktion $dec(-)$ entwickelt.

Definition 26 ($dec : \mathcal{P}_{\nu} \to \mathbb{N}^{\mathcal{P}_{\mathcal{F}}/\equiv}$)**.** Sei $P_{\nu} = \prod_{i=1}^{n} F_i$. Dem Prozess P_{ν} ordnen wir eine Multimenge $dec(P_{\nu}) : \mathcal{P}_{\mathcal{F}}/_{\equiv} \to \mathbb{N}$ zu, indem wir $(dec(P_{\nu}))([F]) := |J_F|$ wählen, wobei $J_F \subseteq \{1,\dots,n\}$ mit $F \equiv F_j \Leftrightarrow j \in J_F$ ist [Mey07b].

Damit gibt $(dec(P_{\nu}))([F])$ wieder, wie viele zu F strukturell kongruente Fragmente in P_{ν} enthalten sind.

Mit diesen Funktionen kann die in [Mey07b] vorgestellte Semantik definiert werden:

Definition 27 (Strukturelle Semantik $\mathcal{N} : \mathcal{P} \to \mathbf{PN}$)**.** Jedem Prozess $P \in \mathcal{P}$ ordnet die strukturelle Semantik ein Petrinetz $\mathcal{N}[\![P]\!] := (S, T, F, M_0)$ zu mit

$$\begin{aligned} S \;&:=\; Frag(Reach(P))/_{\equiv} \\ T \;&:=\; \{([F],[Q]) \in S \times \mathcal{P}/_{\equiv} \mid F \to Q\} \\ &\quad\; \cup \{([F_1],[F_2],[Q]) \in S \times S \times \mathcal{P}/_{\equiv} \mid F_1|F_2 \to Q \text{ echte Kommunikation}\} \end{aligned}$$

Seien s das Fragment $[F]$, t die Transition $([F'],[Q])$ und t' die Transition $([F_1],[F_2],[Q])$, dann sind die Kantengewichtsfunktion und die Anfangsmarkierung wie folgt definiert, [Mey07b]:

$$\begin{aligned} F(s,t) \;&:=\; (dec(F'))([F]) \\ F(s,t') \;&:=\; (dec(F_1))([F]) + (dec(F_2))([F]) \\ F(t,s) \;&:=\; F(t',s) := (dec(\nu(Q)))([F]) \\ M_0 \;&:=\; dec(\nu(P)) \end{aligned}$$

Die Stellenmenge der Semantik eines Prozesses ist die Menge der Kongruenzklassen der vom Prozess aus erreichbaren Prozesse. Transitionen gibt es für Reaktionen einzelner Prozesse und für Kommunikationen über freie Namen zwischen zwei Prozessen.

In [Mey07b] wird der Begriff der strukturellen Stationarität eingeführt und besprochen. Dort wird in Lemma 10 festgestellt, dass genau die Prozesse eine endliche strukturelle Semantik haben, die strukturell stationär sind.

Definition 28 (Strukturelle Stationarität)**.** Ein Prozess $P \in \mathcal{P}$ ist genau dann strukturell stationär, wenn es eine endliche Menge an Fragmenten gibt, die zu jedem von P aus erreichbaren Prozess P' einen strukturell kongruenten enthält, [Mey07b]:

$$\exists \{F_1, \ldots, F_n\} \forall Q \in \mathit{Reach}(P) \forall F \in \mathit{Frag}(\nu(Q)) \exists 1 \leq i \leq n : F \equiv F_i.$$

Eine erreichbare Markierung des Petrinetzes $\mathcal{N}[\![P]\!]$ lässt sich mittels der $\mathit{retrieve}(-)$-Funktion ein von P aus erreichbarer Prozess zurückgewinnen.

Definition 29 ($\mathit{retrieve} : \mathbb{N}^S \to \mathcal{P}/_{\equiv}$)**.** Sei $\mathcal{N}[\![P]\!] = (S, T, F, M_0)$ die strukturelle Semantik des Prozesses $P \in \mathcal{P}$. Die Funktion $\mathit{retrieve} : \mathbb{N}^S \to \mathcal{P}/_{\equiv}$ ordnet jeder Markierung $M \in \mathit{Reach}(\mathcal{N}[\![P]\!])$ eine Prozessklasse $[Q] \in \mathcal{P}/_{\equiv}$ wie folgt zu, [Mey07b]:

$$\mathit{retrieve}(M) := \left[\prod_{[F] \in S} \prod_{i=1}^{M([F])} F \right],$$

wobei $\prod_{i=1}^{0} F := \mathbf{0}$, [Mey07b].

Theorem 1 in [Mey07b] sagt aus, dass die strukturelle Semantik das Verhalten des Prozesses genau wiedergibt. Mit diesem Theorem gilt:

Die Transitionssysteme sind isomorph und $\mathit{retrieve}(\mathit{dec}(\nu([Q]))) = [Q]$.

2.4 Eine sequentielle Petrinetz-Semantik

Die in [Mey07a] eingeführte Concurrency-Petrinetz-Semantik wird hier abkürzend „sequentielle Semantik“ genannt. Beweise von wichtigen Eigenschaften, wie der Möglichkeit der Zurückgewinnung vom zu einem Petrinetzzustand gehörenden Prozess, stehen noch aus.

Definition 30 (Indexfunktion $Ind : \mathcal{P} \to \mathcal{P}_{\mathcal{I}}$)**.** Indexfunktion Ind ordnet einem Prozess aus $\mathcal{P}$ einen indizierten Prozess aus $\mathcal{P}_{\mathcal{I}} \subseteq \mathcal{P}$, [Mey07a]. Sie ist rekursiv definiert als

$$\begin{aligned}
Ind(K\lfloor\tilde{a}\rfloor) &:= K\lfloor\tilde{a}\rfloor \\
Ind(\textstyle\sum_{i\in I} \pi_i.P_i) &:= \textstyle\sum_{i\in I} \pi_i.P_i \\
Ind(P|Q) &:= Ind(P)|Ind(Q) \\
Ind(\nu a.P) &:= \nu(a.T).Ind(\{(a,T)/a\}P) \\
Ind(\nu(a,T).P) &:= \nu(a.H).Ind(\{(a,H)/(a,T)\}P) \\
Ind(\nu(a,H).P) &:= \nu(a.H).Ind(P).
\end{aligned}$$

Da die Indexfunktion Ind lediglich unter Restriktion gebundene Namen umbenennt, ohne die Eindeutigkeit der Namen zu beeinflussen, gilt offensichtlich $Ind(P) \equiv_\alpha P$, d.h. die Indexfunktion führt lediglich eine α-Konversion auf P durch.

Mit der Indexfunktion lässt sich der Begriff der indizierenden Läufe einführen, über welche die Stellenmenge der Semantik zu berechnen ist.

Definition 31 (Indizierende Läufe)**.** Ein indizierender Lauf zu einem Prozess $P_0 \in \mathcal{P}$ ist ein Lauf der Form

$$r := P_0 \equiv_\alpha Ind(P_0) \to P_1 \equiv_\alpha Ind(P_1) \to \ldots,$$

wobei

- die Namen, die von Ind geändert wurden, also restringierte Namen, die nicht von einem Präfix verdeckt werden, nicht durch α-Konversion umbenannt werden und
- restringierte Namen mit Präfix in P_i, aber ohne Präfix in $P_{i+1}, i \in \mathbb{N}$, werden so umbenannt, dass sie, unabhängig vom Interleavingverhalten, eindeutig sind und
- eindeutige Namen werden nicht mit der $\nu a.\mathbf{0} \equiv \mathbf{0}$-Regel entfernt.

Die Menge der von P_0 aus per indizierendem Lauf erreichbaren Prozesse ist $Reach_{\mathcal{I}}(P_0) := \bigcup_{r \text{ ind. Lauf}} Reach_{\mathcal{I}}(r)$.

Da die Reaktionsrelation die (Struct)-Regel enthält, gilt offensichtlich, [Mey07a],

$$Reach(P)/_{\equiv} = Reach_{\mathcal{I}}(P)/_{\equiv}.$$

Es kann also ohne Einschränkung die Menge der via indizierendem Lauf erreichbaren Prozesse betrachtet werden.

Um die Stellenmenge für die Semantik berechnen zu können, wird die Funktion SR eingeführt.

Definition 32 ($SR : \mathcal{P}_{\mathcal{I}} \to \mathbb{N}^{\mathcal{P}/\equiv \cup \mathcal{N}}$)**.** Die rekursive Funktion SR ordnet einem indizierten Prozess eine Multimenge von Kongruenzklassen seiner sequentiellen Teilprozesse und markierten Namen unter Restriktion zu, [Mey07a]:

$$\begin{aligned} SR(K\lfloor\tilde{a}\rfloor) &:= \{[K\lfloor\tilde{a}\rfloor]\} \\ SR(\textstyle\sum_{i\in I} \pi_i.P_i) &:= \{[\textstyle\sum_{i\in I} \pi_i.P_i]\} \\ SR(P|Q) &:= SR(P) \cup SR(Q) \\ SR(\nu(a,T).P) &:= \{a\} \cup_{\mathcal{M}} SR(\{a/(a,T)\}P) \\ SR(\nu(a,H).P) &:= SR(\{a/(a,H)\}P) \end{aligned}$$

Mit diesen Funktionen wird nun die sequentielle Semantik definiert, die in [Mey07a] auch Concurrency Semantics genannt wird.

Definition 33 (Sequentielle Semantik $\mathcal{N}_{\mathcal{C}} : \mathcal{P} \to \mathbf{PN}$)**.** Die sequentielle Semantik ist eine Funktion $\mathcal{N}_{\mathcal{C}} : \mathcal{P} \to \mathbf{PN}$, die ein Petrinetz $\mathcal{N}_{\mathcal{C}}[\![P_0]\!] = (S, T, F, M_0) \in \mathbf{PN}$ zu jedem $P_0 \in \mathcal{P}$ wie folgt zuordnet, [Mey07a]:

$$S := SR(Ind(Reach_{\mathcal{I}}(P_0)))$$

$$M_0(a) := 1 \Leftrightarrow \forall a' \in S : \neg succ(a', a) \qquad M_0([P]) := (SR(Ind(P_0)))([P])$$

Sei $in(Q, ([Q_1], \ldots, [Q_n]), (a_1, \ldots, a_m), (a'_1, \ldots, a'_{m'}))$ definiert als Bedingung

$$\begin{aligned} & SR(Ind(Q)) = \{a_1, \ldots, a_m\} \cup_{\mathcal{M}} \{[Q_1], \ldots, [Q_n]\} \\ \wedge \quad & \forall a' \in \{a'_1, \ldots, a'_{m'} \exists a \in \{a_1, \ldots, a_m\} : succ(a, a') \\ \wedge \quad & \forall b \in S \setminus \{a'_1, \ldots, a'_{m'} \forall a \in \{a_1, \ldots, a_m\} : \neg succ(a, b) \end{aligned}$$

Die Menge $T \subseteq \mathbb{N}^S \times \mathbb{N}^S$ der Transitionen enthält folgende Elemente:

$$\begin{aligned} & (\{[P], a_1, \ldots, a_m\}, \{[Q_1], \ldots, [Q_n], a'_1, \ldots, a'_{m'}\}) \in T \\ \Leftrightarrow \quad & P \to Q \wedge Q \equiv_\alpha Ind(Q) \\ & \wedge\, in(Q, ([Q_1], \ldots, [Q_n]), (a_1, \ldots, a_m), (a'_1, \ldots, a'_{m'})) \end{aligned}$$

$$\begin{aligned} & (\{[P_1], [P_2], a_1, \ldots, a_m\}, \{[Q_1], \ldots, [Q_n], a'_1, \ldots, a'_{m'}\}) \in T \\ \Leftrightarrow \quad & P_1|P_2 \to Q \text{ echte Kommunikation } \wedge Q \equiv_\alpha Ind(Q) \\ & \wedge\, in(Q, ([Q_1], \ldots, [Q_n]), (a_1, \ldots, a_m), (a'_1, \ldots, a'_{m'})). \end{aligned}$$

Kantengewichtsfunktion F wird über die Transitionen definiert:

$$\forall t = (X, Y) \in T \forall s \in S : F(s,t) = (X)(s) \wedge F(t,s) = (Y)(s).$$

Das *succ*-Prädikat kann grob dadurch beschrieben werden, dass $succ(b, a)$ genau dann gilt, wenn b vor a erzeugt wurde. Auf das *succ*-Prädikat wird an dieser Stelle nicht näher eingegangen.

Wie bei der strukturellen Semantik wird eine Charakterisierung für die Endlichkeit der Semantik angegeben. Theorem 2 in [Mey07a] besagt, dass die sequentielle Semantik $\mathcal{N}_{\mathcal{C}}[\![P]\!]$ genau dann endlich ist, wenn P restriktions-stationär ist. Das bedeutet, dass es eine endliche Menge an Namen gibt, die die Menge aller in erreichbaren Prozessen restringierten Namen, welche nicht unter einem Präfix sind, enthält. Formal:

$$\bigcup_{Q \in Reach_{\mathcal{I}}(P)} rn_{NoPrefix}(Q) \text{ ist endlich}$$

2.5 Kombination der Semantiken

Da beide Semantiken jeweils unterschiedliche Anforderungen an Prozesse stellen, damit die konstruierten Petrinetze endlich sind, stellt sich die Frage, ob sich die Semantiken kombinieren lassen. Eine Kombination der Semantiken könnte dann möglicherweise Prozesse in endliche Petrinetze abbilden, die von keiner der beiden Semantiken in endliche Petrinetze übersetzt werden.

Das in dieser Arbeit vorgestellte Verfahren nimmt eine Semantik als „treibende" Semantik und die jeweils andere als Hilfssemantik an. Dabei wird nach der treibenden Semantik konstruiert, bis sicher ist, dass das Netz unendlich würde. Dieser Problemfall wird mit der Hilfssemantik umgesetzt. Anschließlich wird mit der treibenden Semantik fortgefahren.

Dieses Vorgehen unterscheidet sich von der in [Mey07a] definierten Semantik. Dort werden verschiedene Restriktionsoperatoren vorgeschlagen und die auftretenden Namen

- nach der sequentiellen Semantik behandelt, falls die erste Restriktion verwendet wurde und
- nach der strukturellen Semantik behandelt, falls die zweite Restriktion verwendet benutzt wurde.

Angetrieben wurden die Ideen zur Kombination der Semantiken von den Beobachtungen, dass beide Semantiken für verschiedene Prozesse unendlich, bzw. endlich werden und diese Unterschiede auf eine Art gegensätzlich sind, sich aber kombinieren lassen:

- Die strukturelle Semantik ist endlich, wenn nur endlich viele Fragmente in den erreichbaren Prozessen vorkommen. D.h., sie wird unendlich, wenn unendlich viele Fragmente in den erreichbaren Prozessen vorkommen. [Mey07b]
- Die sequentielle Semantik ist endlich, wenn nur endlich viele restringierte Namen in den erreichbaren Prozessen vorkommen. Werden unendlich viele durch Restriktionen gebundene Namen erzeugt, so wird die Semantik unendlich. [Mey07a]

Bei der strukturellen Semantik werden Restriktionen und damit die Kommunikationsstrukturen erhalten, während bei der sequentiellen Semantik die Restriktionen zu freien Namen aufgelöst werden.

Konkret ließe sich bei der Übersetzung problematischer Prozesse der Umgang mit Namen zwischen den Semantiken umschalten.

Strukturell/Sequentiell Identifiziert man im Laufe des Erzeugen des Netzes mit der strukturellen Semantik ein Fragment mit einer Restriktion als ersten Teil, welches unendlich in die Breite wachsen kann, so markiert man die Restriktion und löst sie mit der sequentiellen Semantik zu einem freien Namen auf.

Sequentiell/Strukturell Wird im Laufe der Berechnung der sequentiellen Semantik eine Stelle erzeugt, deren zugehörender Prozess eine Restriktion enthält, die prinzipiell unbeschränkt oft zu einem freien Namen aufgelöst wird, so markiert man sie. Sie wird dann im Sinne der strukturellen Semantik behandelt und nicht aufgelöst.

Das hier präsentierte Verfahren bietet [Mey07a] gegenüber den Vorteil, dass keine Kenntnis über die Verwendung der Namen zur Wahl der geeigneten Restriktion notwendig ist. Ferner ist in [Mey07a] keine treibende Semantik zu finden, die Übersetzungsvorschrift ist statisch. Vergleichend lässt sich feststellen, dass das Entdecken einer Unendlichkeit und deren Behandlung durch die Hilfssemantik einer anderen Restriktion in [Mey07a] entspricht.

Das Problem, zu wissen, wann einer der beschriebenen Fälle von Unendlichkeit bei der Berechnung der Semantiken auftritt, ist semi-entscheidbar. Es können aber hinreichende Bedingungen dafür angeführt werden.

2.6 Programmcode

Da im Rahmen dieser Arbeit einige Algorithmen entwickelt und besprochen werden, soll die verwendete Syntax erläutert werden. Es wird sich einer Pseudo-Programmiersprache bedient, die eine Mischung von gängigen Programmiersprachen und mathematischer Notation darstellt. Als Ausdrücke sind sowohl übliche mathematische und programmiertechnische Ausdrücke wie auch eine Mischung aus beiden gestattet.

let Oft taucht das `let`-Schlüsselwort mit einer nachfolgenden Gleichung $X = Y$ auf. Dies ist als „Ohne Einschränkungen/großen Umstand kann X als Y aufgefasst werden“ zu verstehen. Es kann zum Beispiel eine Zerlegung eines Wortes $w = \alpha \cdot v \cdot \beta$ gemeint sein, die einfach umzusetzen ist, oder bereits vorliegt. Auch Mengen und Funktionen werden gelegentlich mit `let` definiert. Mit `let` wird der Fokus auf dem für den Algorithmus Wesentlichen gehalten oder ein bestimmtes Objekt definiert.

assert Das `assert`-Schlüsselwort stellt eine Vorbedingung für den Algorithmus dar. Ist diese Bedingung erfüllt, muss die durch das `ensure`-Schlüsselwort beschriebene Bedingung für die korrekte Arbeitsweise erfüllt werden.

ensure Die Nachbedingung wird mit dem `ensure`-Schlüsselwort beschrieben und steht meist direkt vor einem `return`. Es kann aber auch an anderen Stellen in einem Programm stehen, um dort auf das Gelten eines bestimmten Sachverhalts hinzuweisen.

in, out, inout Diese Schlüsselwörter geben an, ob ein Parameter in dem Unterprogramm nur gelesen (`in`), nur geschrieben (`out`), oder gelesen und geschrieben (`inout`) werden darf.

Kommentare Wenn bei der ersten Zuweisung einer Variablen nicht klar ist, welchen Wertebereich sie besitzt, so wird dies abkürzend in einem Kommentar vermerkt.

Die übrigen Schlüsselwörter und Anweisungen haben die jeweils übliche Bedeutung.

3 Semantiken

Um die Berechnung der Semantiken für einen Prozess automatisieren zu können, sind die deklarativen Definitionen prozedural umzusetzen.

Die entstehenden Algorithmen müssen die Semantiken korrekt berechnen, d.h. die Ausgabe das Algorithmus muss dem Ergebnis der Semantikfunktion entsprechen.

Für die strukturelle Semantik wird diesen Anforderungen entsprochen. Die Ergebnisse werden in diesem und den folgenden Kapitel der Arbeit vorgestellt. In Bezug auf die sequentielle Semantik wird hier ein abgewandeltes Petrinetz berechnet, das aber in seinem Ablaufverhalten dem von der Semantik definierten entspricht. Der Grund für diese geänderte Berechnung ist die geringere Größe des resultierenden Netzes, wie auch die geringere Komplexität der Berechnung.

3.1 Berechnung der Semantiken

Die nun vorgestellten Algorithmen sind Rahmenprogramme zur Berechnung der Semantiken, die eine Vielzahl an Hilfsalgorithmen einsetzen. Aus diesen Rahmenprogrammen kann ein Verständnis für die Motivation und Anforderungen an die Hilfsalgorithmen abgeleitet werden. Die Algorithmen, die meist umfangreicher und komplexer als die Rahmenprogramme sind, werden in den nachfolgenden Kapiteln entwickelt.

Wie bei Graphalgorithmen oft geschehen, wird bei allen Semantiken eine Art Rand von noch zu behandelnden Stellen und Stellenpaaren aufgebaut. Transitionen werden, analog zu den formalen Definitionen der Semantiken, auf zwei unterschiedliche Weisen erzeugt: Kommunikationen über gebundene Namen und τ-Schritte innerhalb eines Fragments werden getrennt von „echten“ Kommunikationen zweier Fragmente über freie Namen berechnet.

Diese beiden Formen von Reaktionen werden in den Berechnungen separiert. Reaktionen einer Stelle resultieren in Transitionen, die genau die Stelle im Vorbereich haben. Können zwei Stellen $[P], [Q]$ echt kommunizieren, weil sie gleichzeitig markiert werden können, so würde die Berechnung aller Reaktionen von $P|Q$ auch die Reaktionen von P alleine und von Q alleine ergeben – natürlich in Parallelkomposition mit dem jeweils anderen, unveränderten Prozess. Für das Erzeugen von Transitionen ist es aber wichtig zu wissen, ob eine echte Kommunikation zwischen zwei Stellen stattgefunden hat. Da die Reaktionen von P meist nicht zusammen mit den echten Kommunikationen, von $P|Q$, d.h.

$$P \to P' \wedge Q \to Q' \wedge P|Q \to X \Rightarrow X \not\equiv P'|Q \wedge X \not\equiv P|Q'$$

zu berechnen sind, bietet es sich an, von $P|Q$ nicht alle, sondern nur die echten Kommunikationsreaktionen zu berechnen. Ein Aussortieren von bereits behandelten Reaktionen von P entfällt dann.

Ebenfalls den Semantik-Berechnungen gemeinsam ist die Verwendung eines Überdeckungsgraphen, um festzustellen, welche Fragmente gleichzeitig erreicht werden und damit über freie Namen kommunizieren könnten.

3.1.1 Strukturelle Semantik

Um die strukturelle Semantik eines Prozesses zu berechnen, werden vom Systemprozess aus erreichbare Prozesse sukzessive errechnet und ihre Kongruenzklassen als Stellen zum Petrinetz hinzugefügt. Entsprechend der Semantik werden Transitionen und Kanten eingefügt.

Umgesetzt ist dies durch eine Randmenge an Kongruenzklassen, die dem Netz hinzugefügt werden, und deren Bearbeitung noch aussteht. Zu Beginn der Berechnung wird der Systemprozess in seine Fragmente zerlegt, zu denen die Kongruenzklassen die Stellen des initialen Netzes bilden. Die Anfangsmarkierung wird dabei direkt bei der Zerlegung des Prozesses in Fragmente generiert. Die Stellen des initialen Netzes werden in die Randmenge übernommen und ebenso die Paare verschiedener Stellen und auch Paare (s, s), wenn s unter der Anfangsmarkierung mindestens zwei Marken trägt, da diese miteinander kommunizieren könnten.

In der Hauptschleife wird jeweils ein Element aus der Randmenge herausgenommen und

bearbeitet. Zu dem Element werden die Reaktionen berechnet, wobei wie beschrieben die beiden Fälle „alle Reaktionen eines Prozesses“ und „echte Kommunikationen zwischen zwei Prozessen“ unterschieden werden. Jede so berechnete Reaktion wird dann in eine entsprechende Transition übersetzt und sichergestellt, dass es für die Fragmente des reagierten Prozesses Stellen im Netz gibt. Werden dabei neue Stellen erzeugt, so werden sie in die Randmenge eingefügt. Enthält das Netz bisher keine Transition mit genau den errechneten Ein- und Ausgangskanten, so wird die neue Transition dem Netz hinzugefügt. Da das Netz nun prinzipiell mehr Zustände annehmen kann, wird berechnet, welche Stellenpaare mögliche (neue) Kommunikationspartner darstellen. Diese werden ebenfalls in die Randmenge übernommen.

Der Algorithmus terminiert, wenn die Randmenge geleert wird, also keine neuen Transitionen mehr hinzugefügt werden können. Insbesondere hält die Berechnung genau dann nicht, wenn der eingegebene Prozess nicht strukturell stationär ist und damit immer neue Fragmente erreicht werden, die zu keiner bereits erzeugten Kongruenzklasse gehören, so dass für sie eine neue Stelle eingeführt wird, die wiederum reagieren kann.

Folgender Algorithmus berechnet die strukturelle Semantik wie beschrieben.

Listing 3.1: Berechnung der strukturellen Semantik

```
1  function structSem(in P)
2      assert true
3      // Eingabeprozess in strenge Restriktionsform überführen
4      P := restrictedForm(P) // Listing 4.1
5      // Initiales Netz aufbauen
6      N := (S = fragments(P)/≡, T = ∅, F = 0, M_0 = dec(P))
7      // Initialen Überdeckungsgraph aufbauen
8      Cov := (V = {M_0}, E = ∅, T = ∅)
9
10     /* checkCommunications liefert eine Menge von Stellenpaaren,
11      * die prinzipiell in dem gegebenen Zustand kommunizieren
12      * können. Die Menge C wird die Stellenpaare enthalten,
13      * die bereits in die Randmenge aufgenommen worden sind. */
14     C := checkCommunications(M_0) // Listing 7.1
15
```

```
16      /* Menge G stellt die Randmenge von zu betrachtenden
17       * Prozessen dar, die möglicherweise reagieren können und
18       * somit zu einer neuen Transition führen könnten. */
19      G := fragments(P) ∪ C // ⊆ P_sν ∪ P_sν × P_sν , Listing 3.3
20
21      // Hilfsmenge, die die behandelten Elemente aus G enthält
22      G' := ∅ // ⊆ P_sν ∪ P_sν × P_sν
23
24      // Solange noch Elemente in der Randmenge sind
25      while G ≠ ∅ do
26          wähle x ∈ G
27          G := G \ {x}
28          G' := G' ∪ {x}
29
30          // B stellt den Vorbereich der Transition dar
31          let B : ℕ^P Multimenge
32          if x ∈ P_sν then
33              B := {x}
34              // Reaktionen des Prozesses berechnen
35              R := reactions(x) // Listing 5.1
36          else // x ∈ P_sν × P_sν
37              let x = (x_1, x_2)
38              B := {x_1, x_2}
39              // Echte Kommunikationen berechnen
40              R := comReactions(x_1, x_2) // Listing 5.3
41          end if
42
43          // Für jede errechnete Reaktion...
44          for each Q ∈ R do
45              let N = (S, T, F, M_0)
46              // Transition t_neu erschaffen
47              T' := T ∪ {t_neu}
48              F' := F
49              // Vorbereich aufbauen: x, x_1, x_2 oder nur x_1 = x_2
```

```
50          for each s ∈ supp(B) do
51              F'(s, t_neu) := B(s)
52          end for
53          // Nachbereich aufbauen
54          Q := restrictedForm(Q) // Listing 4.1
55          S' := fragments(Q) // Listing 3.3
56          for each s' ∈ S' do
57              F'(t_neu, s') := 0
58          end for
59          for each s' ∈ S' do
60              if ∃s ∈ S : s ≡ s' then
61                  ersetze s' in S' durch s
62                  s' := s
63              end if
64              F'(t_neu, s') := F'(t_neu, s') + 1
65          end for
66          ensure •t_neu = supp(B) ∧ t_neu• = Frag(Q)/≡
67                  ∧ Kanten im Sinne der Semantik
68
69          /* Gibt es bereits eine Transition mit den gleichen
70           * Ein- und Ausgangskanten, so wird die neue
71           * Transition nicht ins Netz eingefügt. */
72          if ∀t ∈ T∃s ∈ S ∪ S' : (F(s,t) ≠ F(s,t_neu) ∨ F(t,s) ≠ F(t_neu,s)) then
73              // neue Stellen in die Randmenge aufnehmen
74              G := G ∪ S'
75
76              // Menge neuer möglicher Kommunikationspartner
77              C' := ∅
78              /* Überdeckungsgraphen aktualisieren und (neue)
79               * Stellenpaare berechnen, zwischen denen eine
80               * Kommunikation möglich ist, Listing 7.1 */
81              Cov := addTransition(Cov, N, F', t_neu, C')
82              // Neue Stellenpaare in Randmenge übernehmen
83              G := (G ∪ C') \ C
```

```
84                  // Neue Stellenpaare als behandelt markieren
85                  C := C ∪ C′
86                  // Erweitertes Netz übernehmen
87                  N := (S′ ∪ S, T′, F′, M₀)
88              end if
89              let N = (S″, T″, F″, M₀)
90              ensure ∃t ∈ T″ : F″|t_neu entspricht F″|t
91              ensure t_neu• ⊆ S′ ∧ S′ \ S ⊆ G
92              ensure S″ ∪ {(s, s′) ∈ S″ × S″ | ∃M ∈ [M₀⟩ : (M(s) ≥ 1 ≤ M(s′)
93                                      ∧(s = s′ ⇒ M(s) ≥ 2))} ⊆ G ∪ G′
94          end for // Weitere Reaktionen
95      end while // Randmenge nicht leer
96
97      ensure N = 𝒩⟦P⟧
98      return N
99  end function
```

3.1.1.1 Korrektheit

Es wird von einer FIFO-Reihenfolge bei der Auswahl des Elements der Randmenge ausgegangen. Dies garantiert, das ein Element, dass zur Randmenge hinzugefügt wurde, nach endlicher Zeit zur Bearbeitung herausgenommen wird.

Der Beweis der Korrektheit ist wegen der unterschiedlichen Fälle und dem Erreichbarkeitsbegriff komplex. Er kann induktiv über die Menge der Stellen des Netzes geführt werden und wird hier kurz angerissen.

Seien P ein Prozess und $N = (S, T, F, M_0)$ das vom Algorithmus berechnete Netz. Offensichtlich ist $Frag(\nu(P))/_{\equiv} \subseteq S$. Zu Beginn des Algorithmus gilt $S \subseteq G$ und stets gilt die Aussage

$$S \cup \{(s, s') \in S \times S \mid \exists M \in [M_0\rangle : (M(s) \geq 1 \leq M(s') \wedge (s = s' \Rightarrow M(s) \geq 2))\} \subseteq G \cup G'.$$

Dies sagt aus, dass jede Stelle und jedes Paar, das kommunizieren könnte, im Rand oder in der Menge der behandelten Elemente ist. Am Algorithmus kann abgelesen werden,

dass ein Element genau dann als behandelt markiert werden kann, wenn es im Rand war und dann behandelt wurde.

Da der Algorithmus zu einem Element aus G die entsprechenden Reaktionen berechnet und zugehörige neue Transitionen und Stellen erzeugt und dann wiederum die neuen Reaktions-/Kommunikationskandidaten zu G hinzufügt, wird die gewünschte Semantik vollständig und korrekt berechnet.

3.1.2 Sequentielle Semantik

Da die Berechnung der sequentiellen Semantik sehr aufwändig ist und die Entwicklung der folgenden Algorithmen noch vor der Formalisierung der Semantik stattfand, wird der Begriff der sequentiellen Semantik weniger streng verwendet.

Der Algorithmus `seqSem` berechnet eine „starke Faltung“ der sequentiellen Semantik eines Prozesses. Unter starker Faltung wird hier ein Netz verstanden, welches Transitionen, die Restriktionen zu freien Namen auflösen, nicht eindeutig unterscheidet, sondern in einer Transition zusammenfasst. Wie in Unterabschnitt 3.1.2.1 gezeigt wird, können dadurch Zustände erreicht werden, die im Netz der sequentiellen Semantik nicht erreichbar sind. Der Vorteil dieser starken Faltung ist, dass sie sich ähnlich der strukturellen Semantik berechnen lässt. Lediglich die Restriktionen ohne Präfix werden zu freien Namen, die noch nicht im Netz vorkommen, aufgelöst. Nach Berechnung der starken Faltung wird für die Transitionen, bei denen Restriktionen aufgelöst werden, eine obere Schranke der Anzahl der Feuerschritte, die sie höchstens vom Startzustand aus machen kann. Existiert für jede Namen-erzeugende Transition eine solche Schranke in $\mathbb{N}$, so ist die Semantik endlich und die starke Faltung kann zu einer Approximation der sequentiellen Semantik aufgefaltet werden. Dabei werden die Teile des Netzes vervielfältigt, die einen freien Namen beinhalten, der von einer Transition erzeugt wurde, die in der starken Faltung häufiger als ein Mal feuern kann. Die so vervielfältigten, Namen-erzeugenden Transitionen bekommen jeweils eine neue, anfänglich mit einer Marke belegte Stelle in ihren Vorbereich, so dass sie nur ein einziges Mal feuern können. So wird gewährleistet, dass jeder freie Name nur ein Mal erzeugt wird und eindeutig ist. In Unterabschnitt 3.1.2.1 wird näher auf die starke Faltung eingegangen.

Listing 3.2: Approximation der sequentiellen Semantik

```
1  function seqSem(in P)
2      assert true
3      // Eingabeprozess in strenge Restriktionsform überführen
4      P := restrictedForm(P) // Listing 4.1
5      // Initiales Netz aufbauen
6      N := (S =fragments(P)/≡, T = ∅, F = 0, M_0 = dec(P))
7      // Initialen Überdeckungsgraph aufbauen
8      Cov := (V = {M_0}, E = ∅, T = ∅)
9
10     /* checkCommunications liefert eine Menge von Stellenpaaren,
11      * prinzipiell in dem gegebenen Zustand kommunizieren
12      * können. Die Menge C wird die untersuchten Stellenpaare
13      * enthalten. */
14     C := checkCommunications(M_0) // Listing 7.1
15
16     /* Menge G stellt die Randmenge von zu betrachtenden
17      * Prozessen dar, die möglicherweise reagieren können und
18      * somit eine neue Transition eingefügt werden könnte. */
19     G := fragments(P) ∪ C // ⊆ P_sν ∪ P_sν × P_sν, Listing 3.3
20
21     // Hilfsmenge, die die behandelten Elemente aus G enthält
22     G' := ∅ // ⊆ P_sν ∪ P_sν × P_sν
23
24     /* T_ν bildet Transitionen auf die Menge der von
25      * ihnen erzeugten Namen ab. */
26     let T_ν : T → ℙ(𝒩) // initial ∅ für alle t ∈ T
27
28     // Solange noch Elemente in der Randmenge sind
29     while G ≠ ∅ do
30         wähle x ∈ G
31         G := G \ {x}
32         G' := G' ∪ {x}
```

```
33
34      // B stellt den Vorbereich der Transition dar
35      let B : 𝒫 → ℕ Multimenge
36      if x ∈ 𝒫_sν then
37          B := {x}
38          // Reaktionen des Prozesses berechnen
39          R := reactions(x) // Listing 5.1
40      else // x ∈ 𝒫_sν × 𝒫_sν
41          let x = (x_1, x_2)
42          B := {x_1, x_2}
43          // Echte Kommunikationen berechnen
44          R := comReactions(x_1, x_2)// Listing 5.3
45      end if
46
47      // Für jede errechnete Reaktion...
48      for each Q ∈ R do
49          // Neue Transition erschaffen
50          T' := T ∪ {t_neu}
51          F' := F
52          // Vorbereich aufbauen: x, x_1, x_2 oder nur x_1 = x_2
53          for each s ∈ supp(B) do
54              F'(s, t_neu) := B(s)
55          end for
56
57          Q := restrictedForm(Q) // Listing 4.1
58          U := fragments(Q) // Listing 3.3
59
60          /* Die berechneten Fragmente können mit Restrik-
61           * tionen beginnen, die mit dieser Semantik
62           * aufgelöst werden. Die Menge der tatsächlichen
63           * Nachbereichsstellen wird S' sein. */
64          S' := ∅
65
66
```

```
67          while U ≠ ∅ do
68              wähle H ∈ U
69              U := U \ {H}
70              if H = νa.H' then
71                  // Die Restriktion muss aufgelöst werden
72                  // Wähle dazu einen neuen Namen
73                  wähle a' ∈ 𝒩 \ ⋃_{s∈S∪S'}(bn(s) ∪ fn(s))
74                  // Entsprechend umbenennen
75                  H' := {a'/a}H'
76                  // Der Prozess kann in Fragmente zerfallen
77                  U := U ∪ fragments(H') // Listing 3.3
78                  /* Vermerke, dass diese Transition eine
79                   * Restriktion zu einem neuen freien Namen
80                   * a' aufgelöst hat. */
81                  T_ν(t_neu) := T_ν(t_neu) ∪ {a'}
82              else
83                  // keine Restriktion
84                  S' := S' ∪ {H}
85              end if
86          end while
87
88          // Nachbereich aufbauen
89          for each s ∈ S' do
90              F'(t_neu, s) := 0
91          end for
92          for each s ∈ S' do
93              if ∃s' ∈ S : s ≡ s' then
94                  ersetze s in S' durch s'
95                  s := s'
96              end if
97              F'(t_neu, s) := F'(t_neu, s) + 1
98          end for
99
100         /* Gibt es bereits eine Transition mit den gleichen
```

```
101              * Ein− und Ausgangskanten, so wird die neue
102              * Transition nicht ins Netz eingefügt. */
103             if ∀t ∈ T∃s ∈ S ∪ S' : (F(s,t) ≠ F(s,t_neu) ∨ F(t,s) ≠ F(t_neu,s)) then
104                 // neue Stellen in die Randmenge aufnehmen
105                 G := G ∪ S'
106
107                 /* Überdeckungsgraphen aktualisieren und (neue)
108                  * Stellenpaare berechnen, zwischen denen eine
109                  * Kommunikation prinzipiell möglich ist. */
110                 S' := S ∪ S'
111                 // Menge neuer möglicher Kommunikationspartner
112                 C' := ∅
113                 /* Überdeckungsgraphen erweitern und neue
114                  * Kommunikationspaare berechnen, Listing 7.1 */
115                 Cov := addTransition(Cov, N, F', t_neu, C')
116                 // Neue Stellenpaare in Randmenge übernehmen
117                 G := (G ∪ C') \ C
118                 // Neue Stellenpaare als behandelt markieren
119                 C := C ∪ C'
120                 // Erweitertes Netz übernehmen
121                 N := (S', T', F', M_0)
122             end if
123             ensure ∃t ∈ T'' : F''|_{t_neu} entspricht F''|_t
124             ensure t_neu• ⊆ S' ∧ S' \ S ⊆ G
125             ensure S'' ∪ {(s,s') ∈ S'' × S'' | ∃M ∈ [M_0⟩ : (M(s) ≥ 1 ≤ M(s')
126                                      ∧(s = s' ⇒ M(s) ≥ 2))} ⊆ G ∪ G'
127         end for // Weitere Reaktionen
128     end while // Randmenge nicht leer
129
130     firingBounds := firingBound(Cov, M_0, supp(T_ν)) // Listing 7.3
131     if ∃t ∈ supp(T_ν) : firingBounds(t) = ω then
132         ensure Die Semantik des Prozesses könnte ein unendliches Netz sein.
133         return nil
134     end if
```

```
135
136     todo := supp(T_ν)
137     while todo ≠ ∅ do
138         wähle t ∈ todo
139         todo := todo \ {t}
140         if firingBounds(t) > 1 then
141             N := unfold(N, t, T_ν, firingBounds, todo) // Listing 8.1
142         end if
143         firingBounds(t) := 0 // Als behandelt markieren.
144     end while
145     // Sichere Stellen im Vorbereich der Transitionen erzeugen
146     for each t ∈ supp(firingBounds) do
147         if firingBounds(t) > 1 then
148             S := S ∪ {s_t}
149             F(s_t, t) := 1
150             M_0 := M_0 + 1 · s_t
151         end if
152     end for
153
154     return N
155 end function
```

3.1.2.1 Starke Faltung

In diesem Unterabschnitt wird das neue Konzept der starken Faltung anhand eines Beispiels visualisiert. Ein anderes Beispiel verdeutlicht einen Problemfall für die starke Faltung: Da mehrere Stellen und Transitionen in der starken Faltung aufeinander fallen terminiert der Algorithmus mit der Aussage, dass das Netz unendlich werden könnte, obwohl tatsächlich nur endlich viele Restriktionen aufgelöst werden können.

Abbildung 3.1 veranschaulicht im linken oberen Teil die starke Faltung des Netzes zum darüber stehenden Prozess S. In diesem Netz könnte die Namen-erzeugende Transition t_1, wie auch Transition t_3, höchstens zweimal feuern. Der Netzteil, indem der von t_1 erzeugte, freie Name a vorkommt, wird aufgefaltet, wodurch das Netz rechts oben entsteht.

Dabei wird im rechten Netzteil das Vorkommen von a durch a' ersetzt und die Kopie der Transition t_3 erzeugt ein b'.

$$
\begin{aligned}
S &:= (\overline{x}.\mathbf{0} \mid \overline{x}.\mathbf{0} \mid \overline{y}.\mathbf{0} \mid \overline{y}.\mathbf{0} \mid x.\nu a.P\lfloor a\rfloor \mid x.\nu a.P\lfloor a\rfloor) \\
P(a) &:= y.(\nu b.Q\lfloor a,b\rfloor \mid P\lfloor a\rfloor)
\end{aligned}
$$

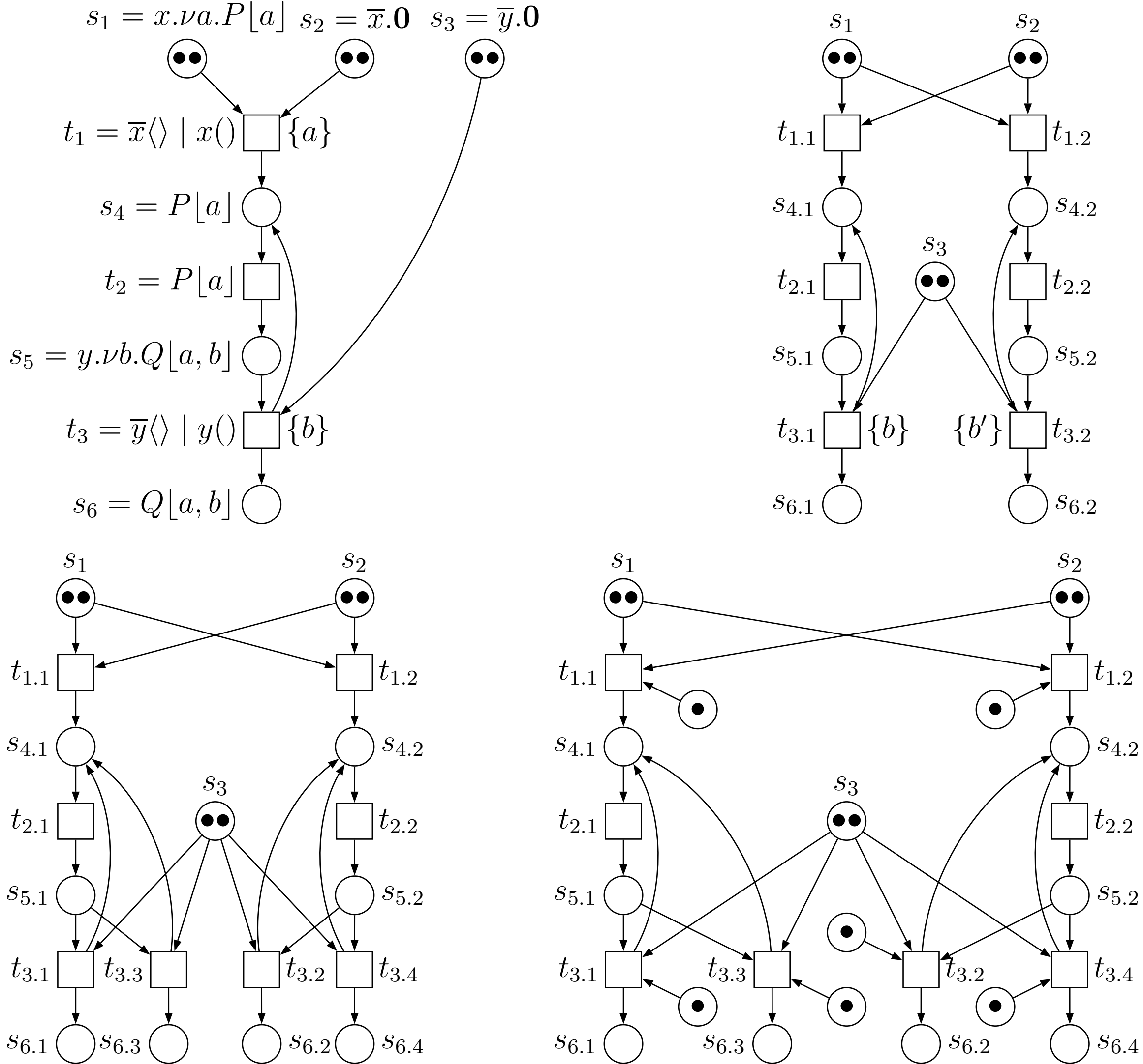

Abbildung 3.1: Beispiel einer Auffaltung

Im nächsten Schritt (links unten) werden die Transitionen $t_{3.1}$ und $t_{3.2}$, sowie die Netzteile, die b, bzw. b' benutzen, entsprechend der Anzahl möglicher Feuerschritte von t_3 vervielfältigt.

Als letzte Änderung werden die Namen erzeugenden Transitionen und ihre Kopien mit

sicheren Stellen verbunden, so dass jede Namen erzeugende Transition höchstens einmal feuern kann.

Es ist sicherzustellen, dass ein und derselbe Namen dabei nicht mehrmals über mehrere Kopien einer Transition erzeugt wird. Von einer Transition können nur dann mehr Kopien als maximal mögliche Feuerschritte erzeugt werden, wenn in einem Netzteil mit einem aufgelösten Namen, wie in Abbildung 3.1, vervielfacht wird, der eine weitere Namen erzeugende Transition enthält.

Kann Transition t höchstens k-mal feuern und im aufgefalteten Netz sind $n > k$ Kopien von t (einschließlich) enthalten, so können höchstens k Kopien feuern, da die Stellen in den Vorbereichen der Kopien höchstens weniger Marken enthalten können, als die Stellen im Vorbereich von t im stark gefalteten Netz. Ein formaler Beweis steht für diesen komplexen Sachverhalt noch aus.

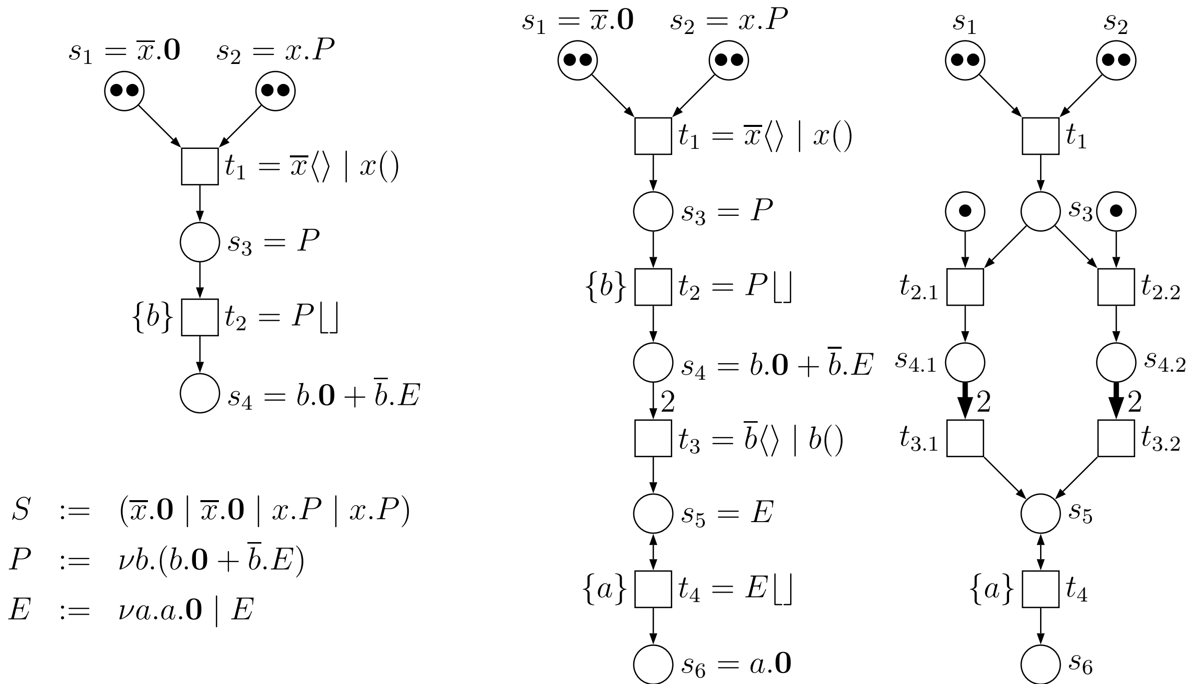

Abbildung 3.2: Ein Problemfall für die starke Faltung

Abbildung 3.2 stellt einen Teil des Verlaufs der Berechnung der sequentiellen Semantik für einen problematischen Prozess S dar. Das Netz auf der linken Seite entspricht dem stark gefalteten Netz. Transition t_2 erzeugt den Namen b und am Überdeckungsgraphen wird abgelesen, dass sie zweimal feuern kann, also zwei verschiedene b erzeugt werden müssten. Dies wird in der starken Faltung nicht korrekt dargestellt: hier handelt es sich

um ein einzelnes b, so dass Stelle s_4 mit zwei Marken belegt wird und mit sich selbst kommunizieren kann. Tatsächlich dürfte diese Kommunikation nicht stattfinden.

Der weitere Aufbau des stark gefalteten Netzes ist in der Mitte der Abbildung zu erkennen. Durch die falsche Kommunikation wird ein Prozess E erreichbar, der unbeschränkt viele Namen a erzeugt, was zu einem unendlichen Netz führt.

Würde man das Netz, abgesehen von Transition t_4, auffalten, so erhielte man das Netz auf der rechten Seite der Abbildung. Dort ist erkennbar, dass Transition t_4 nicht feuern kann. Weder $s_{4.1}$, noch $s_{4.2}$ können mit mehr als einer Marke belegt werden und deswegen können auch $t_{3.1}$ und $t_{3.2}$ nicht feuern.

Die Ausgabe des Algorithmus, dass der behandelte Prozess unbeschränkt viele Namen erzeugt, ist offensichtlich falsch. Eine Abhilfe ist, erst die Netzteile aufzufalten, die zu Namen erzeugenden Transitionen gehören, welche beschränkt viele Namen erzeugen und dann zu diesem, zum Teil aufgefalteten Netz, den Überdeckungsgraphen neu zu berechnen. Mit diesem neuen Überdeckungsgraphen könnte dann genauso weiter verfahren werden, bis alle Namen erzeugenden Transitionen beliebig oft feuerbar sind, oder bis sie alle höchstens einmal feuerbar sind.

Nichtsdestotrotz ist die vorgestellte Approximation sicher, d.h. falls der Algorithmus feststellt, die Semantik ist endlich, so gilt dies definitiv. Eine Aussage, die Semantik sei unendlich, kann wie beschrieben irrtümlich sein.

3.1.2.2 Vergleich

In Abbildung 3.3 wird die sequentielle Semantik und die aus Abbildung 3.1 bekannte Auffaltung des Prozesses S angegeben. Die beiden Netze sind nahezu identisch. Allerdings ist in dem durch Auffaltung entstandenen Netz ein Zustand erreichbar, der $s_{6.1}$ und $s_{6.2}$ markiert, was im Netz der sequentiellen Semantik nicht der Fall ist. Dass in der sequentiellen Semantik die Stellen $s_{6.1} = Q\lfloor a_1, b_1 \rfloor$ und $s_{6.2} = Q\lfloor a_2, b_1 \rfloor$ den gleichen freien Namen b_1 benutzen und in der Auffaltung jeweils neue Namen erzeugt werden, rechtfertigt das Verhalten des durch Auffaltung erzeugten Netzes.

Während man an Abbildung 3.3 sehen konnte, dass fast gleichgroße Netze bei der sequentiellen Semantik und der Auffaltung zu einem Prozess berechnet werden, gibt es Prozesse, für welche die Auffaltung erheblich kleiner ist.

$$
\begin{aligned}
S &:= (\overline{x}.\mathbf{0} \mid \overline{x}.\mathbf{0} \mid \overline{y}.\mathbf{0} \mid \overline{y}.\mathbf{0} \mid x.\nu a.P\lfloor a\rfloor \mid x.\nu a.P\lfloor a\rfloor) \\
P(a) &:= y.(\nu b.Q\lfloor a,b\rfloor \mid P\lfloor a\rfloor)
\end{aligned}
$$

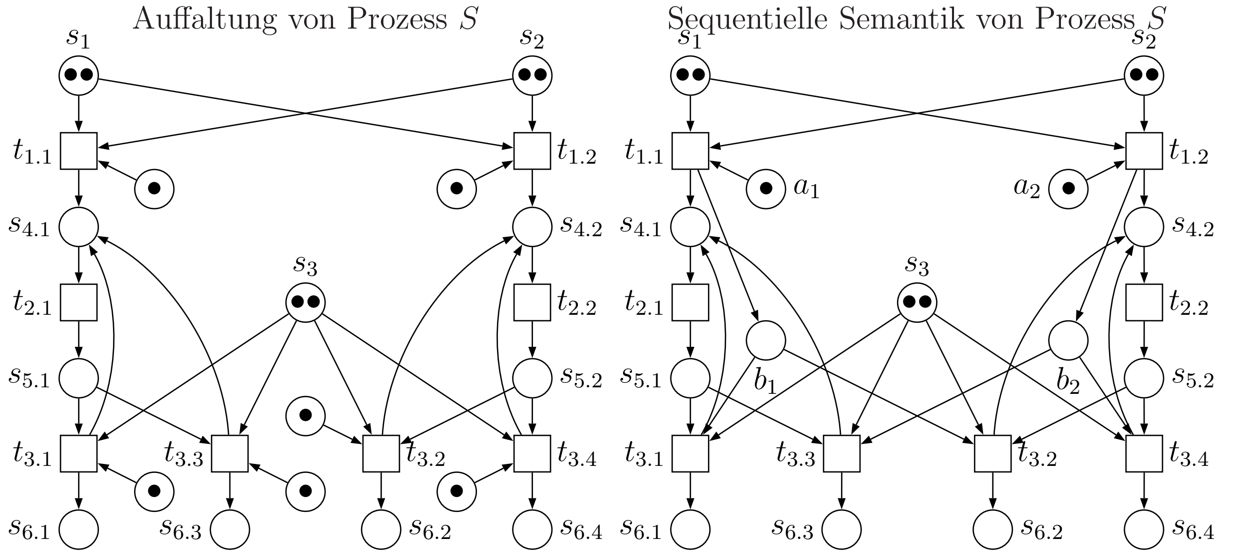

Abbildung 3.3: Vergleich 1: Auffaltung und sequentielle Semantik

Ein Beispielprozess ist P mit

$$
\begin{aligned}
P &:= \overline{a}\langle b\rangle.\nu c.\overline{a}\langle c\rangle.\mathbf{0}|R\lfloor a\rfloor \\
R(x) &:= x(y).\nu d.(R\lfloor x\rfloor|x(z).\overline{d}\langle z\rangle.\mathbf{0}).
\end{aligned}
$$

Seine Auffaltung in Abbildung 3.4 ist deutlich kleiner und enthält einfachere Verknüpfungen als die sequentielle Semantik des Prozesses. Dennoch sind ihre Transitionssysteme isomorph. Die grauen Stellen und Kanten sorgen in der sequentiellen Semantik dafür, dass Namen-erzeugende Transitionen die richtigen Namen in einer richtigen Reihenfolge erzeugen.

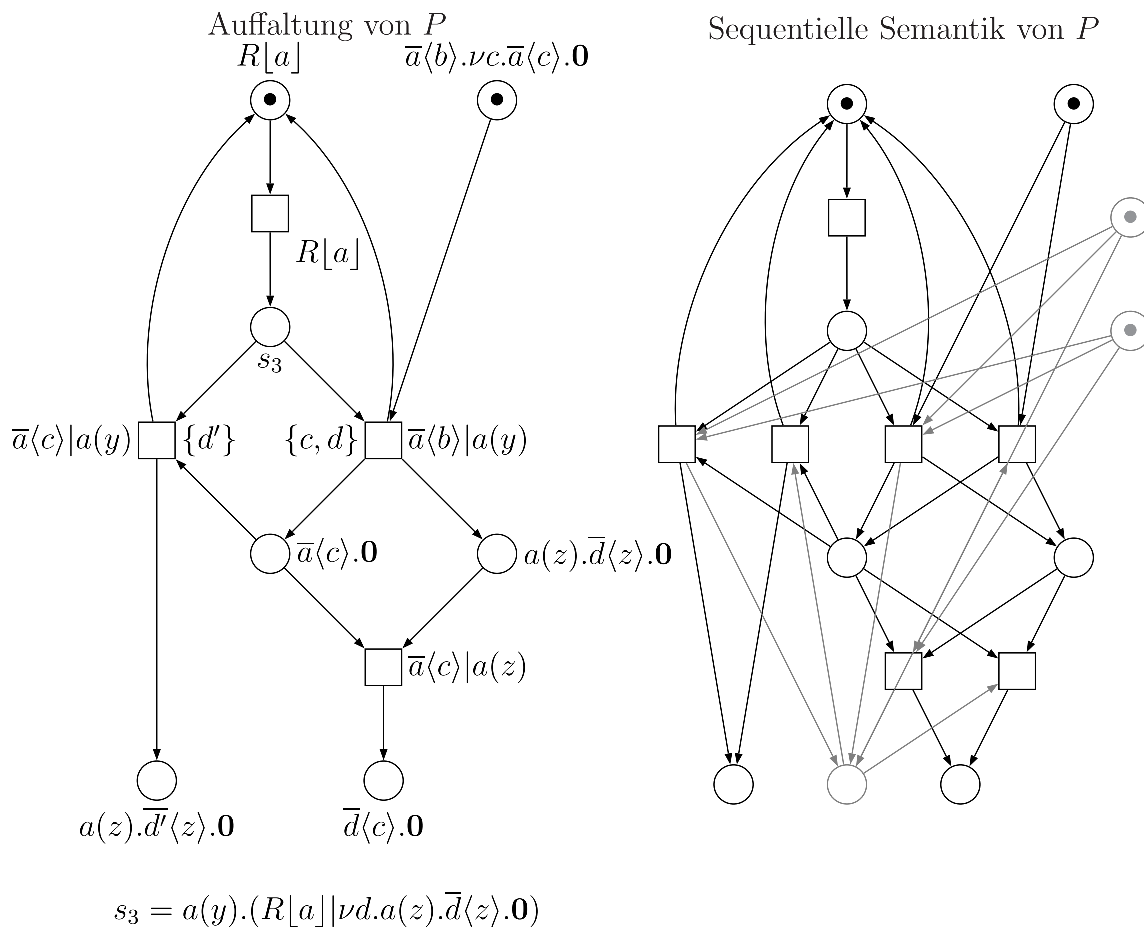

Abbildung 3.4: Vergleich 2: Auffaltung und sequentielle Semantik

3.2 Fragmente

Wurde eine Reaktion $F \rightarrow Q$ oder $F_1|F_2 \rightarrow Q$ berechnet, so ist Prozess Q in seine Fragmente zu zerlegen. Diese bilden die Stellen in den Nachbereichen der Transitionen $([F], [Q])$, bzw. $([F_1], [F_2], [Q])$. Der in Listing 3.3 beschriebene Algorithmus berechnet die Fragmente eines Prozesses in Restriktionsform. Dazu werden zwei Fälle unterschieden: ist der eingegebene Prozess keine Parallelkomposition, so ist der Prozess selbst das einzige Fragment – andernfalls wird die Berechnung der Fragmente auf die parallelen Teilprozesse fortgesetzt. Deren Fragmentmengen werden vereinigt.

Listing 3.3: Berechnung von Fragmenten

```
1  function fragments(in P)
2      assert P ∈ 𝒫_ν
3      if P = ∏_{i∈I≠∅} P'_i then
4          // Bei einer Parallelkomposition ist die Vereinigung
5          // der Fragmente der Teilprozesse zu bilden
6          ensure Frag(P) = ⋃_{i∈I} Frag(P'_i)
7          return ⋃_{i∈I} fragments(P'_i)
8      else
9          // In jedem anderen Fall ist P das einzige Fragment
10         ensure Frag(P) = {P}
11         return {P}
12     end if
13 end function
```

3.2.1 Korrektheit und Aufwandsabschätzung

Dass der Algorithmus gerade die in Definition 23 beschriebenen Fragmente berechnet, gilt, da die Fragmente dort als $P_{\nu i} \neq \mathbf{0}$ in $P_\nu = \prod_{i=1}^{n} P_{\nu i}$ definiert sind, wobei der oberste Operator in $P_{\nu i}$ nicht die Parallelkomposition ist. Hat ein Prozess die Parallelkomposition als obersten Operator, so sind die Fragmente der parallelen Teilprozesse ebenfalls Fragmente des Prozesses.

Offensichtlich kann jeder Teilprozess vom Algorithmus höchstens einmal besucht werden, so dass er in linearer Zeit das Ergebnis zurückliefert.

4 Restriktionsform

Wie in Abschnitt 3.1 beschrieben ist, wird die Berechnung der strengen Restriktionsform für die Semantiken benötigt. Eine rekursive Funktion zur Herstellung der Restriktionsform ist in mathematischer Schreibweise bereits in Definition 24 gegeben. Die strenge Restriktionsform, die in diesem Kapitel aufgebaut wird, zeichnet sich dadurch aus, dass sie möglichst flach ist. D.h. zu einem Prozess $\nu a.\nu b.(P\lfloor a\rfloor | \nu c.(Q\lfloor a,b\rfloor | P\lfloor c\rfloor))$ wird die flache, strenge Restriktionsform $\nu c.P\lfloor c\rfloor | \nu a.(P\lfloor a\rfloor | \nu b.Q\lfloor a,b\rfloor)$ berechnet. Zu diesem Prozess berechnet die Funktion aus Definition 24 $\nu a.(P\lfloor a\rfloor | \nu b.(Q\lfloor a,b\rfloor | \nu c.P\lfloor c\rfloor))$ als Restriktionsform. Für dieses Beispiel entspricht dies der strengen Restriktionsform, da keine Präfixe vorkommen. Die flache Form hat aus Sicht der verwendeten Datenstrukturen den Vorteil, dass beim Traversieren des Syntaxbaumes mehr Iterationen über die Kinder als Rekursionen in die Tiefe einzusetzen sind. Insbesondere bei der häufigen Berechnung der Fragmente eines Prozesses können diese direkt ohne Tiefendurchlauf erkannt werden.

4.1 Berechnung der strengen Restriktionsform

Um die flache, strenge Restriktionsform eines Prozesses P zu errechnen, wird zunächst die Funktion `flatten` auf P angewendet. Diese maximiert die Sichtbarkeit restringierter Namen und nutzt die Assoziativität von ; und | aus, um möglichst breite Prozesskompositionen zu erhalten. Ist der Prozess von der Form $\nu\tilde{a}.\mathbf{0}$, so wird direkt $\mathbf{0}$ zurückgeliefert. Anschließend werden die restringierten Namen entfernt, die nicht vom Prozess benutzt werden, die also über die Regel $\nu a.\mathbf{0} \equiv \mathbf{0}$ und die Scope Extrusion entfernt werden können.

In der folgenden Fallunterscheidung werden die Fälle, der Referenz und der Auswahlkomposition trivial behandelt: Referenzen sind in flacher, strenger Restriktionsform und bei Auswahlkompositionen wird die Funktion rekursiv aufgerufen. Komplexer ist der

Umgang mit einer Parallel- und Sequenzkomposition.

Bei der Behandlung der Parallelkomposition werden zunächst die Restriktionen zwischengespeichert und alle Teilprozesse in flache, strenge Restriktionsform gebracht und **0**-Prozesse entfernt. Die Menge *procs* dieser Prozesse wird dann schrittweise verkleinert: Zu jedem (zwischengespeicherten) restringierten Namen wird die Menge von Prozessen berechnet, die diesen Namen benutzen. Von diesen Mengen wird die kleinste (nichtleere) parallel komponiert und mit einer Restriktion des zugehörigen Namens versehen. Der so entstandene Prozess in flacher, strenger Restriktionsform ersetzt seine Teilprozesse in der Menge *procs*. Dies wird wiederholt, bis keine (zwischengespeicherten) Namen mehr zu restringieren sind. Die verbleibenden Prozesse der *procs*-Menge werden zu einer flachen, strengen Restriktionsform parallel komponiert und zurückgeliefert. Sie stellen die Fragmente des eingegebenen Prozesses dar.

Analog zur Behandlung der Parallelkomposition wird die Sequenzkomposition behandelt, wobei *procs* statt einer Menge ein Wort über $\mathcal{P}$ ist, da dieser Operator nicht kommutativ ist.

Listing 4.1: Überführen in Restriktionsform

```
1  function restrictedForm(in P)
2      assert P = flatten(P)
3
4      if P = νã.0 then
5          Q := 0
6          ensure Q ≡ P ∧ Q ∈ 𝒫_sν
7          return Q
8      else if P = νã.P' then
9          // Ungenutzte Namen entfernen
10         Q := P'
11         for each a_j do
12             if a_j ∈ fn(P') then
13                 Q := νa_j.Q
14             end if
15         end for
16         ensure Q ≡ P ∧ Q = νã.Q' ∧ ∀a ∈ ã : a ∈ fn(Q')
```

```
17      end if
18
19      if $Q = \nu\tilde{a}.K\lfloor\tilde{b}\rfloor$ then
20          // Referenzen sind in flacher, strenger Restriktionsform
21          ensure $Q \equiv P \wedge Q \in \mathcal{P}_{s\nu}$
22          return Q
23
24      else if $Q = \nu\tilde{a}.(\sum_{i\in I\neq\varnothing}\pi_i.P_i)$ then
25          // Bei der Auswahlkomposition rekursiv fortfahren
26          Q := $\nu\tilde{a}.(\sum_{i\in I\neq\varnothing}\pi_i$.restrictedForm($P_i$))
27          ensure $Q \equiv P \wedge Q \in \mathcal{P}_{s\nu}$
28          return Q
29
30      else if $Q = \nu\tilde{a}.(\prod_{i=1}^{n} P_i)$ then
31          // Alle Teilprozesse in flacher, strenger Restriktionsform
32          // in der procs-Menge zusammenfassen
33          procs := {restrictedForm($P_1$),..., restrictedForm($P_n$)} \ {0}
34          // m ordnet jedem Namen die Menge der Prozesse,
35          // die ihn benutzen, zu. Der Träger dieser Abbildung
36          // ist die Menge an Namen, die noch behandelt
37          // werden müssen.
38          let $m : \mathcal{N} \to \mathbb{P}(\mathcal{P})$ // initial ∅ für alle $x \in \mathcal{N}$
39          // Initiale Zuordnung bestimmen
40          for each $a \in \tilde{a}$ do
41              for each $P' \in procs$ do
42                  if $a \in fn(P')$ then
43                      m(a) := $m(a) \cup \{P'\}$
44                  end if
45              end for
46          end for
47          ensure $\forall a \in \tilde{a} \forall P \in procs : (P \in m(a) \Leftrightarrow a \in fn(P))$
48
49          // y ist der Name mit der kleinsten Menge an
50          // Prozessen, die ihn benutzen
```

```
51          y := ϵ
52          for each x ∈ supp(m) do
53              if y = ϵ ∨ |m(x)| < |m(y)| then
54                  y := x
55              end if
56          end for
57          ensure (supp(m) ≠ ∅ ⇔ y ≠ ϵ) ∧ ∀x ∈ supp(m) : |m(x)| ≥ |m(y)|
58
59          // Wiederholen, solange es eine nichtleere,
60          // Menge an Prozessen gibt, die einen
61          // gemeinsamen Namen nutzen.
62          while y ≠ ϵ do
63              let 'suppm = |supp(m)|
64
65              // Prozesse der minimalen Menge
66              // parallel komponieren
67              Q' := νy.∏_{P'∈m(y)} P'
68              ensure Q ∈ 𝒫_{sν}
69
70              // Die benutzten Prozesse aus der Menge
71              // entfernen und die neue Komposition einfügen
72              procs := (procs \ m(y)) ∪ {Q'}
73
74              // Den nächsten Namen mit minimaler Prozess−
75              // menge berechnen und gleichzeitig die
76              // Prozessmengen korrigieren.
77              y' := ϵ
78              for each x ∈ supp(m) \ {y} do
79                  if m(x) ∩ m(y) ≠ ∅ then
80                      // Die in Q' komponierten Prozesse
81                      // entfernen und dafür Q' einfügen
82                      m(x) := (m(x) \ m(y)) ∪ {Q'}
83                  end if
84
```

```
85                  if y' = ϵ ∨ |m(x)| < |m(y')| then
86                      y' := x
87                  end if
88              end for
89              // Den Namen als behandelt markieren
90              m(y) := ∅
91              y := y'
92
93              ensure (supp(m) ≠ ∅ ⇔ y ≠ ϵ) ∧ ∀x ∈ supp(m) : |m(x)| ≥ |m(y)|
94                      ∧|supp(m)| <' suppm
95          end while
96          ensure procs ⊆ P_sν
97
98          // Sobald alle Namen behandelt wurden, können die
99          // errechneten Fragmente parallel komponiert werden.
100         Q := ∏_{P'∈procs} P'
101         ensure Q ≡ P ∧ Q ∈ P_sν
102         return Q
103
104     else if Q = νã.(P_1; ...; P_n) then
105         // Ein Wort über P
106         procs := ϵ // ∈ P*
107         // Das initiale Wort aufbauen, das alle Teilprozesse
108         // in flacher, strenger Restriktionsform enthält.
109         for i = 1...n do
110             P' := restrictedForm(P_i)
111             if P' ≢ 0 then
112                 procs := procs · P'
113             end if
114         end for
115         ensure ∀α · P · β = procs : P ∈ P_sν \ {0})
116         // Zu jedem Namen liefert m das minimale Wort,
117         // bei dem der erste und der letzte Prozess den
118         // Namen benutzen. Sein Träger enthält die Namen
```

```
119        // die noch zu behandeln sind.
120        let m : 𝒩 → 𝒫* // initial ε für alle x ∈ 𝒩
121        for each a ∈ ã do
122            let procs = S_1 · ··· · S_k
123            if ∃S ∈ {S_1, ..., S_k} : a ∈ fn(S) then
124                k_s := min{1 ≤ m ≤ k | a ∈ fn(S_m)}
125                k_e := max{k_s ≤ m ≤ k | a ∈ fn(S_m)}
126                for m = k_s ... k_e do
127                    m(a) := m(a) · S_m
128                end for
129            end if
130        end for
131        ensure ∀a ∈ ã∀α · P · β · Q · γ = procs :
132                (P · β · Q = m(a) ⇔ a ∈ fn(P) ∩ fn(Q) ∧ a ∉ fn(α) ∪ fn(γ))
133
134        // y ist der Name, dem m das kleinste Wort
135        // ungleich ε zuordnet.
136        y := ε
137        for each x ∈ supp(m) do
138            if y = ε ∨ |m(x)| < |m(y)| then
139                y := x
140            end if
141        end for
142        ensure (supp(m) ≠ ∅ ⇔ y ≠ ε) ∧ ∀x ∈ supp(m) : |m(x)| ≥ |m(y)|
143
144        // Solange noch Namen zu behandeln sind...
145        while y ≠ ε do
146            let 'suppm = |supp(m)|
147            // Komponiere die Prozesse, die den Namen
148            // mit minimalem Wort benutzen, in Sequenz.
149            let m(y) = S_1 · ··· · S_k
150            Q' := νy.(S_1; ...; S_k)
151            ensure Q' ∈ 𝒫_sν
152            // Das Wort zum behandelten Namen aus procs
```

```
153                 // entfernen und durch die Komposition ersetzen
154                 let procs = α · m(y) · β
155                 procs := α · Q′ · β
156
157                 // Das neue Minimum suchen und gleichzeitig
158                 // die Abbildung m korrigieren.
159                 y′ := ϵ
160                 for each x ∈ supp(m) \ {y} do
161                     if ∃α′, β′ ∈ 𝒫* : m(x) = α′ · m(y) · β′ then
162                         m(x) := α′ · Q′ · β′
163                     end if
164                     if y′ = ϵ ∨ |m(x)| < |m(y′)| then
165                         y′ := x
166                     end if
167                 end for
168                 // Namen y als behandelt markieren.
169                 m(y) := ϵ
170                 y := y′
171                 ensure (supp(m) ≠ ∅ ⇔ y ≠ ϵ) ∧ ∀x ∈ supp(m) : |m(x)| ≥ |m(y)|
172                        ∧|supp(m)| <′ suppm
173             end while
174             let procs = S_1 · ··· · S_k
175             ensure ∀1 ≤ i ≤ k : S_i ∈ 𝒫_sν
176
177             // Die verbleibenden Prozesse ohne Restriktion in
178             // Sequenz komponieren.
179             Q := S_1; ...; S_k
180             ensure Q ≡ P ∧ Q ∈ 𝒫_sν
181             return Q
182         end if
183 end function
```

4.1.1 Flache Form

Funktion `flatten` nutzt die Assoziativität von ; und |, wie auch die Scope-Extrusion-Regel aus, um einen Prozess in eine strukturell kongruente Form zu überführen, die nur maximal breite Sequenz- und Parallelkompositionen enthält.

Listing 4.2: Berechnung der Restriktionsform: Flache Form

```
1  function flatten(in νã.P)
2      assert true
3      if P = K⌊b̃⌋ then
4          // Referenzen sind bereits flach.
5          return P
6      end if
7
8      let P = (P_1 ∗ ··· ∗ P_n) ∧ ∗ ∈ {;, +, |}
9      if ∗ = + then
10         // Die Auswahlkomposition ist nicht assoziativ.
11         return νã.(P_1 ∗ ··· ∗ P_n)
12     end if
13
14     ensure ∗ ∈ {;, |} // Assoziative Operatoren
15     // res̃ stellt die Menge der Restriktionen dar
16     res̃ := ã
17     // Das Wort der Teilprozesse, die über die Assoziativität
18     // geliftet werden können.
19     lift := P_1 · ··· · P_n
20     // Das Wort, das die Prozesse enthält, die nicht weiter
21     // geliftet werden können.
22     lifted := ε
23     while lift ≠ ε do
24         // Ersten Prozess betrachten
25         let lift = P'_1 · α
26         lift := α
27         // Was ist P'_1 für ein Prozess?
```

```
28          if  P'_1 = νb̃.(P''_1 *_2 ··· *_2 P''_m) ∧ *_2 ∈ {; , |} ∧ *_2 ≠ *  then
29              // P'_1 ist Komposition eines assoziativen Operators,
30              // der von dem gerade behandelten verschieden ist.
31              // P'_1 kann flach gemacht werden.
32              P' := flatten(P'_1)
33              // Die extrahierten Restriktionen übernehmen...
34              let  P' = νc̃.P''
35              res~ := res~ ∪ b̃ ∪ c̃
36              // ... und das flache P'' ohne Restriktionen
37              // als behandelt betrachten.
38              lifted := lifted · P''
39          else if  P'_1 = νb̃.(P''_1 * ··· * P''_m)  then
40              // Wegen der Assoziativität von * werden die
41              // Restriktionen extrahiert und die Teilprozesse
42              // geliftet und als zu behandeln markiert.
43              res~ := res~ ∪ b̃
44              lift := P''_1 · ··· · P''_m · lift
45          else if  P'_1 = νb̃.P''  then
46              // In jedem anderen Fall kann P'_1 nicht
47              // geliftet werden und lediglich die Restriktionen
48              // werden extrahiert
49              res~ := res~ ∪ b̃
50              lifted := lifted · P''
51          end if
52      end while
53      let  lifted = P'_1 · ··· · P'_k
54      // Die extrahierten Restriktionen werden vor die Komposition
55      // der gelifteten Prozesse geschrieben.
56      P' := νres~.(P'_1 * ··· * P'_k)
57      ensure  νã.(P_1 * ··· * P_k) ≡ P'
58              ∧∀α · P · β = lifted : P ≢ νx̃.(Q_1 * ··· * Q_m)
59              ∧(νã.(P_1 * ··· * P_n) ∈ 𝒫_sν) ⇒ (∀α · P · β = lifted : P ∈ 𝒫_sν)
60      return P'
61  end function
```

4.2 Korrektheit und Aufwandsabschätzung

Die Anwendung der `flatten`-Funktion entspricht dem wiederholten Anwenden der Regel der Scope Extrusion und der Assoziativität. Kommutativität von bestimmten Operatoren wurde nicht ausgenutzt, so dass die Funktion auch für den Sequenzoperator eingesetzt wird. Die `flatten`-Funktion betrachtet jeden Teilprozess höchstens einmal und läuft deshalb in zur Prozessgröße linearer Zeit.

Das Herstellen der flachen, strengen Restriktionsform betrachtet jeden Prozess ebenfalls nur einmal, enthält jedoch mehrfach geschachtelte Schleifen für die Bearbeitung des Parallel- und des Sequenzoperators. Für jede Restriktion wird die `while` $y \neq \epsilon$ `do`-Schleife einmal ausgeführt. Es ist damit von er zur Prozessgröße und Anzahl der enthaltenen Restriktionen linearer Laufzeit auszugehen. Eine formale Betrachtung des Aufwandes und ein Beweis der Korrektheit stehen aus.

4.3 Bemerkungen zur Implementierung

Da die Berechnung der strengen Restriktionsform häufig stattfindet, wird jede strenge Restriktionsform in einem assoziativen Speicher mit dem eingegebenen Prozess verknüpft. Wiederholtes Ausrechnen der strengen Restriktionsform, auch von Teilprozessen, wird dann unnötig. Dieser assoziative Speicher hat die besondere Eigenschaft, dass er Assoziationen löscht, sobald der Prozess nicht mehr benötigt wird.

5 Reaktionen

Zur Berechnung der Semantiken (Abschnitt 3.1) werden in der Hauptschleife die Reaktionen eines Prozesses aus dem Rand errechnet und die Fragmente jeder Reaktion der Stellenmenge hinzugefügt. Dazu werden drei Funktionen vorgestellt, die mögliche Kommunikationspartner innerhalb eines Prozesses finden, die Reaktionen eines Prozesses berechnen und echte Kommunikationen zwischen zwei Prozessen berechnen.

5.1 Berechnung der Reaktionen

Um die Reaktionen eines Prozesses zu finden wird Funktion `reactions` eingeführt, die einen Prozess aus $\mathcal{P}_\mathbf{0}$ entgegennimmt und einen Wahrheitswert, der angibt, ob Kommunikationen innerhalb des Prozesses zu berechnen sind. Kommunikationen sind nur außerhalb von Präfixen beim ersten Paralleloperator zu berechnen, da mit der `communications`-Funktion auch geschachtelte Parallelkompositionen untersucht werden.

Bei der Berechnung wird der Prozess, gemäß der Reaktionsregel (Res), ohne Restriktionen betrachtet. Diese werden abschließend, vor dem Rückliefern der Ergebnisse, den Reaktionen vorangestellt.

Zu jeder Reaktionsregel, mit Ausnahme von (Struct) wird ein Fall unterschieden, für den die Menge der Reaktionen berechnet wird. Für eine Auswahlkomposition ist die Menge der Reaktionen gerade die Vereinigung der Reaktionen der einzelnen Teilprozesse. Bei der Sequenz ergibt sich die Reaktionsmenge aus den Reaktionen des ersten Prozesses, bei denen jeweils der Rest der Sequenz angehängt wird. Komplizierter ist es, die Reaktionen einer Parallelkomposition zu berechnen.

Eine Parallelkomposition kann einerseits die Reaktionen der parallelen Teilprozesse ausführen, wobei jeweils nur ein Teilprozess einen τ-Reaktionsschritt macht oder eine Re-

ferenz auflöst, und andererseits die Kommunikationen zwischen jeweils zwei Prozessen mit passenden Präfixen (sofern diese noch zu berechnen sind). Zur Berechnung der internen Kommunikationen (wenn *comReactions* wahr ist) wird der Prozess in eine flache Form gebracht (siehe Unterabschnitt 4.1.1) und die enthaltenen Kommunikationspartner mittels der `communications`-Funktion berechnet. Über diese Kommunikationspartner werden dann die Reaktionen ausgerechnet.

Listing 5.1: Berechnung von Reaktionen eines Prozesses

```
1  function reactions(in P)
2      assert P ∈ 𝒫_0
3      R := reactions(P, true)
4      ensure ∀Q ∈ R : (P → Q) ∧ ∀P → P′∃Q ∈ R : (Q ≡ P′)
5      return R
6  end function
7
8  function reactions(in νx̃.P, in comReactions)
9      assert P ∈ 𝒫_0
10
11     if P = 0 ∨ P = ā⟨b̃⟩.P′ ∨ P = a(b̃).P′ ∨ (P = [a = b].P′ ∧ a ≠ b) then
12         R := ∅
13         ensure (∀Q ∈ R : (P → Q)) ∧ (∀νP → P′∃Q ∈ R : (Q ≡ P′))
14
15     else if P = [a = a].P′ then
16         R := reactions(P′, comReactions)
17         ensure (∀Q ∈ R : (P → Q)) ∧ (∀νP → P′∃Q ∈ R : (Q ≡ P′))
18
19     else if P = K⌊ã⌋ ∧ K(ỹ) := P′ then
20         R := {{ã/ỹ}P′}
21         ensure (∀Q ∈ R : (P → Q)) ∧ (∀νP → P′∃Q ∈ R : (Q ≡ P′))
22
23     else if P = τ.P′ then
24         R := {P′}
25         ensure (∀Q ∈ R : (P → Q)) ∧ (∀νP → P′∃Q ∈ R : (Q ≡ P′))
26
```

```
27      else if P = ∑_{i∈I≠∅} P'_i then
28          R := ∅
29          for each i ∈ I do
30              R := R ∪ reactions(P_i, false)
31          end for
32          ensure (∀Q ∈ R : (P → Q)) ∧ (∀νP → P'∃Q ∈ R : (Q ≡ P'))
33
34      else if P = P'_1; P'_2; ...; P'_n ∧ P'_1 then
35          for each Q ∈ reactions(P'_1, comReactions) do
36              R := R ∪ { Q; P'_2; ...; P'_n }
37          end for
38          ensure (∀Q ∈ R : (P → Q)) ∧ (∀νP → P'∃Q ∈ R : (Q ≡ P'))
39
40      else if P = ∏_{i∈I≠∅} P'_i then
41          R := ∅
42          for each i ∈ I do
43              for each Q ∈ reactions(P'_i, false) do
44                  // Reaktionen einzelner Teilprozesse
45                  R := R ∪ { (Q | ∏_{j∈I\{i}} P'_j) }
46              end for
47          end for
48          ensure (∀Q ∈ R : (P → Q)) ∧ (∀P -τ,ref-> P'∃Q ∈ R : (Q ≡ P'))
49
50          if comReactions then
51              // Kommunikationen
52              P' := flatten(P)
53              C := communications(P'')
54              for each ((S, S', x̃), (T, T', ỹ)) ∈ C do
55                  let P' = C_1[S]
56                  U := C_1[S'] // S durch S' ersetzen
57                  let U = C_2[T]
58                  // T durch {x̃/ỹ}T' ersetzen
59                  R := R ∪ { C_2[{x̃/ỹ}T'] }
```

```
60                 end for
61             end if
62             ensure (∀Q ∈ R : (P → Q)) ∧ (∀νP → P′∃Q ∈ R : (Q ≡ P′))
63         end if
64
65         R′ := R
66         R := ∅
67         for each P′ ∈ R′ do
68             R := R ∪ {νx̃.P′}
69         end for
70
71         ensure (∀Q ∈ R : (νx̃.P → Q)) ∧ (∀νx̃.P → P′∃Q ∈ R : (Q ≡ P′))
72         return R
73 end function
```

5.1.1 Kommunikationspartner

Bei der Berechnung von Kommunikationspartnern werden zwei Abbildungen in, out benutzt, um zu speichern, welche Prozesse auf welchen Kanälen wieviele (und welche) Namen versenden oder empfangen: $in, out : \mathcal{N} \times \mathbb{N} \to \mathbb{P}(\mathcal{P} \times \mathcal{P} \times \mathcal{N}^*)$ ist $(T, T', \tilde{x}) \in in(a, k)$, so bedeutet dies, dass Prozess T auf Kanal a genau k Namen $\tilde{x}$ empfängt und dabei zu T' übergeht. Für out gilt dies analog. Mit einer Funktion werden alle Teilprozesse bis zu Präfixen durchlaufen und die Abbildungen entsprechend erweitert. Abschließend werden alle Paare von Prozessen gebildet, die kommunizieren können. Es wird sichergestellt, dass zwei Teilprozesse derselben Auswahlkomposition nicht als Kommunikationspartner ausgegeben werden.

Listing 5.2: Berechnung von Reaktionen: Kommunikationswege

```
1 function communications(in P)
2     assert P ∈ 𝒫₀ ∧ P = flatten(P)
3     let in : 𝒩 × ℕ → ℙ(𝒫 × 𝒫 × 𝒩*) // initial ∅ für alle 𝒩 × ℕ
4     let out : 𝒩 × ℕ → ℙ(𝒫 × 𝒫 × 𝒩*) // initial ∅ für alle 𝒩 × ℕ
5     communications(P, in, out)
6     C := ⋃_{(a,k)∈supp(out)∩supp(in)} out(a,k) × in(a,k)
```

```
7        for each  A = ((S, S', x̃), (T, T', ỹ)) ∈ C  do
8            if S = T then
9                C := C \ {A}  // a̅⟨b⟩.0 + a(c).0 → 0  vermeiden
10           end if
11       end for
12       ensure ∃P' ∈ 𝒫 : P kann mittels einer Kommunikation nach P' übergehen,
13              wobei Subprozess S die k Namen x̃ über Kanal a sendet und dabei nach
14              S' übergeht und Subprozess T die gesendeten Namen als ỹ empfängt und
15              nach {x̃/ỹ}T' übergeht ⇔ ((S, S', x̃), (T, T', ỹ)) ∈ C
16       return C
17   end function
18
19   procedure communications(in P, inout in_g, inout out_g)
20       assert P ∈ 𝒫_0 ∧ P = flatten(P)
21
22       let in : 𝒩 × ℕ → ℙ(𝒫 × 𝒫 × 𝒩*)  // initial ∅ für alle 𝒩 × ℕ
23       let out : 𝒩 × ℕ → ℙ(𝒫 × 𝒫 × 𝒩*)  // initial ∅ für alle 𝒩 × ℕ
24
25       if P = 0 ∨ P = τ.P' ∨ (P = [a = b].P' ∧ a ≠ b) then
26           ensure P kann nicht kommunizieren
27       else if P = [a = a].P'
28           communications(P', in, out)
29       else if P = P'_1; P'_2; ...; P'_n then
30           communications(P'_1, in, out)
31       else if P = ∏_{i∈I≠∅} P'_i then
32           for each i ∈ I do
33               communications(P'_i, in, out)
34           end for
35       else if P = ∑_{i∈I≠∅} P'_i then
36           // Hier wird jeweils die ganze Komposition ersetzt
37           for each i ∈ I do
38               if P'_i = a(b_1, ..., b_k).P'
39                   in(a, k) := in(a, k) ∪ (P, P', b̃)
40               else if P'_i = a̅⟨b_1, ..., b_k⟩.P'
```

```
41                     out(a,k) := out(a,k) ∪ (P, P', b̃)
42                end if
43           end for
44      end if
45
46      ensure (∃P', P_1, P'_1 ∈ 𝒫 : P|P_1 → P'|P'_1, wobei P über Kanal a genau k Namen
47                b̃ an Prozess P_1 sendete und dabei selbst nach P' überging
48                ⇔ (P, P', b̃) ∈ out(a,k))
49             ∧(∃P', P_1, P'_1 ∈ 𝒫 : P|P_1 → P'|P'_1, wobei P über Kanal a genau k Namen
50                x̃ als ỹ von Prozess P_1 empfing und dabei selbst nach {x̃/ỹ}P' überging
51                ⇔ (P, P', ỹ) ∈ in(a,k))
52
53      for each (a,k) ∈ supp(out) do
54           out_g(a,k) := out_g ∪ out(a,k)
55      end for
56      for each (a,k) ∈ supp(in) do
57           in_g(a,k) := in_g ∪ in(a,k)
58      end for
59  end function
```

5.1.2 Echte Kommunikationen

Echte Kommunikationen zwischen zwei Prozessen verwenden immer Namen, die in beiden Prozessen frei sind, als Kanäle. Um echte Kommunikationen zu finden werden *in*- und *out*-Abbildungen für jeden Prozess mittels der besprochenen `communications`-Methode berechnet. Die Kommunikationspartner ergeben sich dann, analog zur internen Kommunikation, als Vereinigung von kartesischen Produkten der *in*-Abbildung des einen und der *out*-Abbildung des anderen Prozesses. Es werden dabei nur die in beiden Prozessen freien Namen als Kanäle erlaubt. Wie die Reaktionen aussehen wird analog zu der Behandlung interner Kommunikationen in Parallelkompositionen über Prozesskontexte beschrieben. Die Restriktionen beider Prozesse werden vereinigt und eine neue Parallelkomposition der reagierten Prozesse wird erzeugt.

Listing 5.3: Berechnung von Kommunikationen zweier Prozesse

1 **function** comReactions(**in** P_1, **in** P_2)
2 **assert** $P_1, P_2 \in \mathcal{P}_\mathbf{0}$
3
4 **let** $in_1 : \mathcal{N} \times \mathbb{N} \rightarrow \mathbb{P}(\mathcal{P} \times \mathcal{P} \times \mathcal{N}^*)$ *// initial* $\varnothing$ *für alle* $\mathcal{N} \times \mathbb{N}$
5 **let** $out_1 : \mathcal{N} \times \mathbb{N} \rightarrow \mathbb{P}(\mathcal{P} \times \mathcal{P} \times \mathcal{N}^*)$ *// initial* $\varnothing$ *für alle* $\mathcal{N} \times \mathbb{N}$
6
7 **let** $in_2 : \mathcal{N} \times \mathbb{N} \rightarrow \mathbb{P}(\mathcal{P} \times \mathcal{P} \times \mathcal{N}^*)$ *// initial* $\varnothing$ *für alle* $\mathcal{N} \times \mathbb{N}$
8 **let** $out_2 : \mathcal{N} \times \mathbb{N} \rightarrow \mathbb{P}(\mathcal{P} \times \mathcal{P} \times \mathcal{N}^*)$ *// initial* $\varnothing$ *für alle* $\mathcal{N} \times \mathbb{N}$
9
10 P_1' := flatten(P_1)
11 P_2' := flatten(P_2)
12
13 communications(P_1', in_1, out_1)
14 communications(P_2', in_2, out_2)
15 $C := \bigcup_{(a,k) \in supp(out_1) \cap supp(in_2) \cap fn(P_1) \cap fn(P_2)} out_1(a,k) \times in_2(a,k)$
16 $\cup \bigcup_{(a,k) \in supp(out_2) \cap supp(in_1) \cap fn(P_1) \cap fn(P_2)} out_2(a,k) \times in_1(a,k)$
17
18 $R := \varnothing$
19 **for each** $((S, S', \tilde{x}), (T, T', \tilde{y})) \in C$ **do**
20 **let** $P_1' = \nu\tilde{b}_1.\mathcal{C}_1[S]$
21 **let** $P_2' = \nu\tilde{b}_2.\mathcal{C}_2[T]$
22 $R := R \cup \{\ \nu\tilde{b}_1.\nu\tilde{b}_2.(\mathcal{C}_1[S'] \mid \mathcal{C}_2[\{\tilde{x}/\tilde{y}\}T'])\ \}$
23 **end for**
24
25 **ensure** $\forall Q \in R : (P_1|P_2 \rightarrow Q$
26 $\wedge((P_1|P_2 \rightarrow P_1'|P_2) \Rightarrow Q \not\equiv P_1'|P_2)$
27 $\wedge((P_1|P_2 \rightarrow P_1|P_2') \Rightarrow Q \not\equiv P_1|P_2'))$
28 $\wedge\forall P_1|P_2 \rightarrow Q \forall P_1 \rightarrow P_1' \forall P_2 \rightarrow P_2' : ((Q \not\equiv P_1'|P_2 \wedge Q \not\equiv P_1|P_2') \Rightarrow Q \in R)$
29 **return** R
30 **end function**

5.2 Korrektheit und Aufwandsabschätzung

Für die Korrektheit kann argumentiert werden, indem belegt wird, dass es zu jeder Regel der Reaktionsrelation einen entsprechenden Fall im Algorithmus gibt, der die Regel korrekt wiederspiegelt. Dabei erweist sich die Regel (Struct) nicht als Problem, da ohne Einschränkungen gefordert wird, dass der eingegebene Prozess in $\mathcal{P}_{\mathbf{0}}$ liegt und es einen Fall im Programm gibt, der erfüllte Wächter korrekt behandelt. Ein formaler Beweis steht noch aus.

Der Algorithmus findet alle ausführbaren Reaktionen in, zur Prozessgröße, linearer Zeit. Der Prozess wird höchstens dreimal durchlaufen: einmal für alle einfachen Fälle und bei einer Parallelkomposition ein weiteres mal mit der `flatten`-Funktion und noch einmal mit der `communications`-Funktion.

6 Strukturelle Kongruenz

Zur Entscheidung, ob zwei Prozesse strukturell kongruent sind, bietet es sich intuitiv an, eine Standardform für diese Prozesse zu wählen. Dennoch verkomplizieren Regeln wie die α-Konversion und die Kommutativität einiger Operatoren das Entscheidungsverfahren.

Durch das Einführen von einfach zu prüfenden notwendigen Bedingungen für strukturelle Kongruenz und einer strengen Standardform für Prozesse wird das Entscheiden vereinfacht.

6.1 Notwendige Bedingungen für strukturelle Kongruenz

Ziel ist es, einige notwendige Bedingungen für die strukturelle Kongruenz von Prozessen zu finden, bei denen einerseits ohne großen Aufwand festzustellen ist, ob sie erfüllt werden und die andererseits möglichst scharf sind, d.h. in ihrer Gesamtheit oder alleine möglichst nah an der strukturellen Kongruenz sind.

6.1.1 Freie Namen

Zwei strukturell kongruente Prozesse besitzen die gleichen freien Namen.

Im Rahmen der strukturellen Kongruenz gibt es keine Regel mit der ein Einführen, Entfernen und Umbenennen von freien Namen erreicht werden kann. Wie bereits in [Mil99] erwähnt, gilt:

$$P \equiv Q \Rightarrow fn(P) = fn(Q).$$

6.1.2 Anzahl von Präfixen, $\mathbf{0}, K\lfloor\tilde{a}\rfloor$

Zwei strukturell kongruente Prozesse enthalten die gleichen Anzahlen an in-, out-, τ-Präfixen, nicht erfüllten Wächtern und Referenzen. Sind diese Prozesse in $\mathcal{P}_{\mathbf{0}}$, so enthalten sie darüber hinaus auch gleichviele $\mathbf{0}$-Teilprozesse.

Die erste Aussage folgt, wie die über freie Namen, daraus, dass die Regeln der strukturellen Kongruenz, außer erfüllten Wächtern, keine Präfixe und keine Referenzen einführen oder entfernen. Da die Prozesse in $\mathcal{P}_{\mathbf{0}}$ nur $\mathbf{0}$-Teilprozesse mit Präfixen haben, die keine erfüllten Wächter sind, müssen auch diese Anzahlen bei strukturell kongruenten Prozessen gleich sein.

Die folgende Definition beschreibt die Zählweisen genau.

Definition 34 **($\#_{in}(-), \#_{out}(-), \#_{\tau}(-), \#_{[\neq]}(-), \#_{\mathbf{0}}(-), \#_{ref}(-)$).** Sei Prozess P in $\mathcal{P}$. Funktionen $\#_{\star}(P) \in \mathbb{N}, \star \in \{in, out, \tau, [\neq]\}$ sind wie folgt definiert, $* \in \{+, ;, |\}$.

$$\begin{aligned}
\#_{\star}(\mathbf{0}) &:= 0 \\
\#_{\star}(K\lfloor\tilde{b}\rfloor) &:= 0 \\
\#_{\star}(\nu a.P) &:= \#_{\star}(P) \\
\#_{\star}(\pi.P) &:= \#_{\star}(\pi) + \#_{\star}(P) \\
\#_{\star}(P_1 * P_2) &:= \#_{\star}(P_1) + \#_{\star}(P_2)
\end{aligned}$$

Für Präfixe gilt jeweils das Folgende, wobei für nicht aufgeführte Fälle der Funktionswert 0 geliefert wird.

$$\begin{aligned}
\#_{in}(a(\tilde{b})) &:= 1 \\
\#_{out}(\overline{a}\langle\tilde{b}\rangle) &:= 1 \\
\#_{\tau}(\tau) &:= 1 \\
\#_{[\neq]}([a = b]) &:= \begin{cases} 1, & a \neq b \\ 0, & a = b \end{cases}
\end{aligned}$$

Davon nur gering abweichend seien $\#_{\mathbf{0}}(-)$ und $\#_{ref}(-)$ definiert, welche die Anzahl der $\mathbf{0}$-Prozesse, beziehungsweise die Anzahl der Referenzen $K\lfloor\tilde{a}\rfloor$ wiedergeben. Für sie ist $\#_{\mathbf{0},ref}(\pi) := 0$, $\#_{ref}(K\lfloor\tilde{b}\rfloor) := 1$ und $\#_{\mathbf{0}}(\mathbf{0}) = 1$.

Seien $P, Q \in \mathcal{P}$ und $P_{\mathbf{0}}, Q_{\mathbf{0}} \in \mathcal{P}_{\mathbf{0}}$ mit $P \equiv P_{\mathbf{0}} \wedge Q \equiv Q_{\mathbf{0}}$. Es gilt:

$$\begin{aligned} P \equiv Q \quad \Rightarrow \quad & \#_{in}(P) = \#_{in}(Q) \wedge \#_{out}(P) = \#_{out}(Q) \wedge \#_{\tau}(P) = \#_{\tau}(Q) \\ & \wedge \#_{[\neq]}(P) = \#_{[\neq]}(Q) \wedge \#_{ref}(P) = \#_{ref}(Q) \wedge \#_{\mathbf{0}}(P_{\mathbf{0}}) = \#_{\mathbf{0}}(Q_{\mathbf{0}}). \end{aligned}$$

Beweis. Sei $\star \in \{in, out, \tau, ref, [\neq]\}$. Per Induktion über die Ableitung struktureller Kongruenz wird $P \equiv Q \Rightarrow \#_{\star}(P) = \#_{\star}(Q)$ gezeigt.

Induktionsanfang: Seien $P, Q, R \in \mathcal{P}$ und M, N Summen von Prozessen. Für jede Regel der strukturellen Kongruenz wird gezeigt, dass die Funktionswerte bei Anwendung der Regel erhalten bleiben. Mit der Definition der Funktion $\#_{\star}(-)$ gelten die Gleichungen:

$$\begin{aligned}
\#_{\star}(\nu a.P) &= \#_{\star}(P) = \#_{\star}(\nu b.\{b/a\}P) \\
\#_{\star}(N + M) &= \#_{\star}(N) + \#_{\star}(M) = \#_{\star}(M) + \#_{\star}(N) = \#_{\star}(M + N) \\
\#_{\star}(P|Q) &= \#_{\star}(P) + \#_{\star}(Q) = \#_{\star}(Q) + \#_{\star}(P) = \#_{\star}(Q|P) \\
\#_{\star}(P|(Q|R)) &= \#_{\star}(P) + \#_{\star}(Q|R) = \#_{\star}(P) + \#_{\star}(Q) + \#_{\star}(R) \\
&= \#_{\star}(P|Q) + \#_{\star}(R) = \#_{\star}((P|Q)|R) \\
\#_{\star}(P;(Q;R)) &= \#_{\star}(P) + \#_{\star}(Q;R) = \#_{\star}(P) + \#_{\star}(Q) + \#_{\star}(R) \\
&= \#_{\star}(P;Q) + \#_{\star}(R) = \#_{\star}((P;Q);R) \\
\#_{\star}(P|\mathbf{0}) &= \#_{\star}(P) + \#_{\star}(\mathbf{0}) = \#_{\star}(P) \\
\#_{\star}(P;\mathbf{0}) &= \#_{\star}(P) + \#_{\star}(\mathbf{0}) = \#_{\star}(P) \\
\#_{\star}(\mathbf{0};P) &= \#_{\star}(\mathbf{0}) + \#_{\star}(P) = \#_{\star}(P) \\
\#_{\star}([a = b].P) &= \#_{\star}(P), \text{ wenn } a = b \\
\#_{\star}(\nu a.\mathbf{0}) &= \#_{\star}(\mathbf{0}) \\
\#_{\star}(\nu a.\nu b.P) &= \#_{\star}(\nu b.P) = \#_{\star}(P) = \#_{\star}(\nu a.P) = \#_{\star}(\nu b.\nu a.P) \\
\#_{\star}(\nu a.(P|Q)) &= \#_{\star}(P|Q) = \#_{\star}(P) + \#_{\star}(Q) = \#_{\star}(P) + \#_{\star}(\nu a.Q) \\
&= \#_{\star}(P|\nu a.Q) \\
\#_{\star}(\nu a.(P;Q)) &= \#_{\star}(P;Q) = \#_{\star}(P) + \#_{\star}(Q) = \#_{\star}(P) + \#_{\star}(\nu a.Q) \\
&= \#_{\star}(P;\nu a.Q) \\
\#_{\star}(\nu a.(P;Q)) &= \#_{\star}(P;Q) = \#_{\star}(P) + \#_{\star}(Q) = \#_{\star}(\nu a.P) + \#_{\star}(Q) \\
&= \#_{\star}(\nu a.P;Q)
\end{aligned}$$

Induktionsschluss: Seien $P, Q, R \in \mathcal{P}, M$ eine Summe von Prozessen und gelte bereits $P \equiv Q \wedge \#_\star(P) = \#_\star(Q)$. Dann gilt:

$$\begin{aligned}
\#_\star(P|R) &= \#_\star(P) + \#_\star(R) = \#_\star(Q) + \#_\star(R) = \#_\star(Q|R) \\
\#_\star(P;R) &= \#_\star(P) + \#_\star(R) = \#_\star(Q) + \#_\star(R) = \#_\star(Q;R) \\
\#_\star(\nu a.P) &= \#_\star(P) = \#_\star(Q) = \#_\star(\nu a.Q) \\
\#_\star(\pi.P + M) &= \#_\star(\pi.P) + \#_\star(M) = \#_\star(\pi) + \#_\star(P) + \#_\star(M) \\
&= \#_\star(\pi) + \#_\star(Q) + \#_\star(M) = \#_\star(\pi.Q) + \#_\star(M) = \#_\star(\pi.Q + M)
\end{aligned}$$

Mit dem Prinzip der strukturellen Induktion ist gezeigt, dass die Funktionswerte von $\#_\star(-)$ invariant unter struktureller Kongruenz sind. □

Dass $\#_{ref}(-)$ und $\#_{\mathbf{0}}(-)$ ebenfalls die gewünschte Eigenschaft haben, kann analog bewiesen werden. Zu beachten ist lediglich, dass $P|\mathbf{0}, P;\mathbf{0}, \mathbf{0};P, \nu a.\mathbf{0} \notin \mathcal{P}_\mathbf{0}$ und die entsprechenden Regeln nicht betrachtet werden müssen.

6.1.3 Anzahl gebundener Namen

Zwei strukturell kongruente Prozesse in $\mathcal{P}_\mathbf{0}$ benutzen die gleiche Anzahl an gebundenen Namen.

Dies kann leicht eingesehen werden, da $\nu x.\mathbf{0} \equiv \mathbf{0}$ die einzige Regel im Rahmen der strukturellen Kongruenz ist, die eine Einführung, beziehungsweise eine Entfernung, einer Restriktion erlaubt. Der betroffene gebundene Name wird aber nicht im Prozess verwendet, da er nicht in der Menge der freien Namen des Prozesses auf der rechten Seite vorkommt. Die Mengen der gebundenen Namen müssen nicht gleich sein, da eine α-Konversion zwischen ihnen erlaubt ist.

Seien $P_\mathbf{0}, Q_\mathbf{0} \in \mathcal{P}_\mathbf{0}$. Es gilt:

$$P_\mathbf{0} \equiv Q_\mathbf{0} \Rightarrow |bn(P_\mathbf{0})| = |bn(Q_\mathbf{0})|.$$

Beweis. Per Induktion über die Ableitung struktureller Kongruenz wird $P \equiv Q \Rightarrow |(P)| = |bn(Q)|$ gezeigt.

Induktionsanfang: Seien $P, Q, R \in \mathcal{P}_\mathbf{0}$ und M, N Summen von Prozessen aus $\mathcal{P}_\mathbf{0}$ (wie gefordert sind gebundene Namen eindeutig). Für jede Regel der strukturellen Kongruenz wird gezeigt, dass der Funktionswert bei Anwendung der Regel erhalten

bleibt. Zu beachten ist, dass $P|\mathbf{0}, P;\mathbf{0}, \mathbf{0};P, \nu a.\mathbf{0} \notin \mathcal{P}_{\mathbf{0}}$ und die entsprechenden Regeln nicht betrachtet werden müssen. Mit der Definition von $bn(-)$ gelten die Gleichungen:

$$\begin{aligned}
|bn(\nu a.P)| &= 1 + |bn(P)| = |bn(\nu b.\{b/a\}P)|, \text{ wenn } b \notin fn(P) \cup bn(P) \setminus \{a\} \\
|bn(N+M)| &= |bn(N)| + |bn(M)| = |bn(M)| + |bn(N)| = |bn(M+N)| \\
|bn(P|Q)| &= |bn(P)| + |bn(Q)| = |bn(Q)| + |bn(P)| = |bn(Q|P)| \\
|bn(P|(Q|R))| &= |bn(P)| + |bn(Q|R)| = |bn(P)| + |bn(Q)| + |bn(R)| \\
&= |bn(P|Q)| + |bn(R)| = |bn((P|Q)|R)| \\
|bn(P;(Q;R))| &= |bn(P)| + |bn(Q;R)| = |bn(P)| + |bn(Q)| + |bn(R)| \\
&= |bn(P;Q)| + |bn(R)| = |bn((P;Q);R)| \\
|bn([a=b].P)| &= |bn(P)| \\
|bn(\nu a.\nu b.P)| &= 1 + |bn(\nu b.P)| = 2 + |bn(P)| = 1 + |bn(\nu a.P)| = |bn(\nu b.\nu a.P)| \\
|bn(\nu a.(P|Q))| &= 1 + |bn(P|Q)| = 1 + |bn(P)| + |bn(Q)| = |bn(P)| + |bn(\nu a.Q)| \\
&= |bn(P|\nu a.Q)| \\
|bn(\nu a.(P;Q))| &= 1 + |bn(P;Q)| = 1 + |bn(P)| + |bn(Q)| = |bn(P)| + |bn(\nu a.Q)| \\
&= |bn(P;\nu a.Q)| \\
|bn(\nu a.(P;Q))| &= 1 + |bn(P;Q)| = 1 + |bn(P)| + |bn(Q)| = |bn(\nu a.P)| + |bn(Q)| \\
&= |bn(\nu a.P;Q)|
\end{aligned}$$

Induktionsschluss: Seien $P, Q, R \in \mathcal{P}_{\mathbf{0}}, M$ eine Summe von Prozessen aus $\mathcal{P}_{\mathbf{0}}$ und gelte $P \equiv Q \wedge |bn(P)| = |bn(Q)|$. Dann gilt:

$$\begin{aligned}
|bn(P|R)| &= |bn(P)| + |bn(R)| = |bn(Q)| + |bn(R)| = |bn(Q|R)| \\
|bn(P;R)| &= |bn(P)| + |bn(R)| = |bn(Q)| + |bn(R)| = |bn(Q;R)| \\
|bn(\nu a.P) &= 1 + |bn(P) = 1 + |bn(Q) = |bn(\nu a.Q) \\
|bn(\pi.P+M)| &= |bn(\pi.P)| + |bn(M)| = |bn(\pi)| + |bn(P)| + |bn(M)| \\
&= |bn(\pi)| + |bn(Q)| + |bn(M)| = |bn(\pi.Q)| + |bn(M)| \\
&= |bn(\pi.Q+M)|,
\end{aligned}$$

wobei hier $bn(\pi) = \begin{cases} \{b_1, \ldots, b_k\}, & \text{wenn } \pi = a(b_1, \ldots, b_k) \\ \varnothing, & \text{sonst} \end{cases}$ sei.

Mit dem Prinzip der strukturellen Induktion ist gezeigt, dass die Anzahl gebundener

Namen von Prozessen in $\mathcal{P}_\mathbf{0}$ invariant unter struktureller Kongruenz ist. □

6.1.4 Reihenfolge von Präfixen

Zwei strukturell kongruente Prozesse besitzen die gleiche Reihenfolge von Präfixen und jeweils die gleiche Anzahl an Parametern.

Dass unter der strukturellen Kongruenz keine Präfixe eingeführt, entfernt, geändert oder versetzt werden können, erlaubt diesen Schluss. Die Funktion `pref(-)` bildet einen sequentiellen Prozess auf seine Präfixfolge ab.

Definition 35 ($pref(-)$)**.** Jedem Prozess $P \in \mathcal{P}_\mathbf{0}$ ordne $pref(P)$ ein Wort aus $(\{\mathtt{g}, \mathtt{t}\} \cup (\{\mathtt{i}, \mathtt{o}\} \cdot \mathbb{N}))^*$ so zu, dass $pref(P)$ Aufschluss über die Reihenfolge der Präfixe von P gibt. Formal sei $pref(-)$ mit $* \in \{+, ;, |\}$ folgendermaßen definiert:

$$
\begin{aligned}
pref(\mathbf{0}) &:= \epsilon \\
pref(K\lfloor\tilde{b}\rfloor) &:= \epsilon \\
pref(P_1 * P_2) &:= \epsilon \\
pref(\nu\tilde{a}.P) &:= pref(P) \\
pref(\pi.P) &:= pref(\pi) \cdot pref(P) \\
pref(\tau) &:= \tau \\
pref(a(b_1, \ldots, b_k)) &:= \mathtt{i} \cdot k \\
pref(\overline{a}\langle b_1, \ldots, b_k\rangle) &:= \mathtt{o} \cdot k \\
pref([a = b]) &:= \begin{cases} \mathtt{g}, & a \neq b \\ \epsilon, & a = b \end{cases}
\end{aligned}
$$

Damit gilt für zwei Prozesse $P_\mathbf{0}, Q_\mathbf{0} \in \mathcal{P}_\mathbf{0}$:

$$P_\mathbf{0} \equiv Q_\mathbf{0} \Rightarrow pref(P_\mathbf{0}) = pref(Q_\mathbf{0}).$$

Beweis. Per Induktion über die Ableitung struktureller Kongruenz wird $P \equiv Q \Rightarrow pref(P) = pref(Q)$ gezeigt.

Induktionsanfang: Seien $P, Q, R \in \mathcal{P}_\mathbf{0}$ und M, N Summen von Prozessen aus $\mathcal{P}_\mathbf{0}$. Für jede Regel der strukturellen Kongruenz wird gezeigt, dass der Funktionswert bei Anwendung der Regel erhalten bleibt. Zu beachten ist, dass $P|\mathbf{0}, P;\mathbf{0}, \mathbf{0};P, \nu a.\mathbf{0} \notin \mathcal{P}_\mathbf{0}$

und die entsprechenden Regeln nicht betrachtet werden müssen. Mit der Definitionen von $pref(-)$ gelten die Gleichungen:

$$\begin{aligned}
pref(\nu a.P) &= pref(P) = pref(\nu b.\{b/a\}P) \\
pref(N+M) &= \epsilon = pref(M+N) \\
pref(P|Q) &= \epsilon = pref(Q|P) \\
pref(P|(Q|R)) &= \epsilon = pref((P|Q)|R) \\
pref(P;(Q;R)) &= \epsilon = pref((P;Q);R) \\
pref([a=b].P) &= pref(P), \text{ wenn } a=b \\
pref(\nu a.\nu b.P) &= pref(\nu b.P) = pref(P) = pref(\nu a.P) = pref(\nu b.\nu a.P) \\
pref(\nu a.(P|Q)) &= pref(P|Q) = \epsilon = pref(P|\nu a.Q) \\
pref(\nu a.(P;Q)) &= pref(P;Q) = \epsilon = pref(P;\nu a.Q) \\
pref(\nu a.(P;Q)) &= pref(P;Q) = \epsilon = pref(\nu a.P;Q)
\end{aligned}$$

Induktionsschluss: Seien $P, Q, R \in \mathcal{P}_\mathbf{0}, M$ eine Summe von Prozessen aus $\mathcal{P}_\mathbf{0}$ und gelte $P \equiv Q \wedge pref(P) = pref(Q)$. Dann gilt:

$$\begin{aligned}
pref(P|R) &= \epsilon = pref(Q|R) \\
pref(P;R) &= \epsilon = pref(Q;R) \\
pref(\nu a.P) &= pref(P) = pref(Q) = pref(\nu a.Q) \\
pref(\pi.P+M) &= \epsilon = pref(\pi.Q+M)
\end{aligned}$$

Mit dem Prinzip der strukturellen Induktion ist gezeigt, dass der Funktionswert von $pref(-)$ für Prozesse in $\mathcal{P}_\mathbf{0}$ invariant unter struktureller Kongruenz ist. □

Bemerkungen zur Implementierung Für einen Prozess $P_\mathbf{0} \in \mathcal{P}_\mathbf{0}$ können die Werte der Funktionen in einem einzelnen Tiefendurchlauf des Syntaxbaums berechnet werden. Die Werte werden dann dem Syntaxbaum als Attribute hinzugefügt, was ein erneutes Berechnen überflüssig macht.

6.2 Eine Ordnung auf Prozessen

Mit diesen notwendigen Bedingungen für strukturelle Kongruenz lässt sich der Aufwand zur Beantwortung der Frage, ob zwei Parallel- oder Summenkompositionen von

Prozessen strukturell kongruent sind, vereinfachen. Dazu wird unter Benutzung der notwendigen Bedingungen eine Ordnung auf Prozessen aufgebaut, die einen Schluss erlaubt, welche Paare von Prozesse nicht auf strukturelle Kongruenz geprüft werden müssen.

Um ein schnelles Identifizieren von nicht strukturell kongruenten Prozessen zu gewährleisten, sollte die Ordnung möglichst feingranular sein.

Definition 36. Ordnung $\leq_{\mathcal{P}}$ Die Ordnungsrelation $\leq_{\mathcal{P}} \subseteq \mathcal{P}_\mathbf{0} \times \mathcal{P}_\mathbf{0}$ sei durch folgende Rangfolge beschrieben.

1. $\leq$ auf $|bn(-)|$
2. $\leq$ auf $|fn(-)|$
3. $\leq$ auf $\#_{in}(-)$
4. $\leq$ auf $\#_{out}(-)$
5. $\leq$ auf $\#_{\tau}(-)$
6. $\leq$ auf $\#_{[\neq]}(-)$
7. $\leq$ auf $\#_{\mathbf{0}}(-)$
8. $\leq$ auf $\#_{ref}(-)$
9. $\leq_{lex}$ auf $pref(-)$
10. $\leq_{lex}$ auf $\prod sort(fn(-), \leq_{lex})$

Hierbei ist $\leq_{lex}$ die lexikographische Ordnung und $\Pi sort(fn(-), \leq_{lex})$ das lexikographische Ordnen und Aneinanderreihen der freien Namen.

Analog seien $=_{\mathcal{P}}, \neq_{\mathcal{P}}, <_{\mathcal{P}}, >_{\mathcal{P}}, \geq_{\mathcal{P}}$ definiert.

Die Reihenfolge der Punkte 1-8 ist willkürlich, da es sich aus praktischer Sicht lediglich um Ganzzahlvergleiche handelt. Hingegen stellen die Punkte 9 und 10 Zeichenkettenvergleiche dar, die aufwändiger sind. Wie in Abschnitt 6.1 beschrieben, fällt der Aufwand des Sortierens für Punkt 10 nur ein Mal an.

Bei der Implementierung kann eine Kurzschlussauswertung angewendet werden, so dass die Berechnung beendet wird, sobald ihr Ergebnis feststeht: Der jeweils nächste Punkt muss nur betrachtet werden, wenn der aktuelle eine Gleichheit ergibt, sonst bestimmt der aktuelle Auswertungspunkt das Ergebnis.

Seien $P_\mathbf{0}, Q_\mathbf{0} \in \mathcal{P}_\mathbf{0}$ mit $P_\mathbf{0} \equiv Q_\mathbf{0}$. Es gilt:

$$P_\mathbf{0} \equiv Q_\mathbf{0} \Rightarrow P_\mathbf{0} =_\mathcal{P} Q_\mathbf{0}$$

Der Korrektheit folgt direkt aus der Korrektheit der einzelnen notwendigen Bedingungen. Offensichtlich ist $\leq_\mathcal{P}$, wie auch die analog definierten Symbole, transitiv und es gilt

$$P \leq_\mathcal{P} Q \wedge Q \leq_\mathcal{P} R \Rightarrow P \leq_\mathcal{P} R.$$

6.3 Prüfen von struktureller Kongruenz

Als zentral bei der Berechnung, ob zwei Prozesse strukturell kongruent sind, erweisen sich zwei Substitutionen, die im Sinne der α-Konversion angeben, welche gebundenen Namen des der Prozesse jeweils umzubenennen sind.

Bei der Prüfung struktureller Kongruenz wird eine Tiefendurchlauf durch die Syntaxbäume der eingegebenen Prozesse durchgeführt, wobei stets die Mengen der gebundenen Namen der eingegebenen Prozesse bereitgehalten werden müssen, da folgender Schluss offensichtlich falsch ist: $P \not\equiv Q \Rightarrow \mathcal{C}_1[P] \not\equiv \mathcal{C}_2[Q]$ Dieser Schluss wird beispielsweise von $P = K\lfloor a \rfloor, Q = K\lfloor b \rfloor$ und $\mathcal{C}_1[P] = \nu a.P, \mathcal{C}_2[Q] = \nu b.Q$ widerlegt, da zwar $P \not\equiv Q$, aber die in P, Q freien Namen a, b durch die Kontexte gebunden werden und so ineinander umbenannt werden können.

Es muss also beim Tiefendurchlauf „erinnert“ werden, dass a und B in den Prozessen, deren strukturelle Kongruenz zu prüfen ist, gebundene Namen sind und damit ineinander umbenannt werden dürfen. Ein dazu alternatives, aber weniger effizientes, Vorgehen wäre es, die richtigen Umbenennungen von gebundenen Namen zu erraten, bzw. alle möglichen Umbenennungen auszuprobieren.

Um struktureller Kongruenz zu entscheiden werden, nach dem Prüfen der zusammengefassten notwendigen Bedingungen mittels $=_\mathcal{P}$, die Präfixe der Prozesse verglichen. Können die Präfixe mittels α-Konversionen ineinander überführt werden, so gibt dies bereits einen ersten Aufschluss über die verwendenden Substitutionen. Wenn nicht, so sind die Prozesse nicht strukturell kongruent.

Auch bei Referenzen lässt sich zügig feststellen ob die Prozesse strukturell kongruent sind und mit welchen Umbenennungen.

Bei Prozesskompositionen mittels Sequenz oder Parallelität kommt bereits eine Hilfskonstruktion zum Einsatz: Die Funktion `flatten` wird angewendet, so dass keine direkten (ohne Präfixe) Schachtelungen von der gleichen Operatoren vorkommen und die Komposition maximal breit ist. Dadurch muss nicht auf die Assoziativität der Operatoren eingegangen werden. Bei der Auswahlkomposition kann dies vernachlässigt werden, da sie nicht assoziativ ist (denn Prozesse in Auswahl tragen Präfixe) und offensichtlich stets $M + \pi.(N + O) \not\equiv (M + \pi.N) + O$ ist.

Liegen zwei maximal breite Sequenzen von Prozessen $P_1; \ldots; P_n$ und $Q_1; \ldots; Q_m$ vor, so sind sie genau dann strukturell kongruent, wenn $n = m$ ist und $P_i \equiv Q_i, 1 \leq i \leq n$ gilt, wobei die vorzunehmenden Substitutionen zu beachten sind.

Für die kommutativen Operatoren $+$ und $|$ sind zwei maximal breite Kompositionen $P_1 * \cdots * P_n$ und $Q_1 * \cdots * Q_m$ mit $* \in \{+, |\}$ genau dann strukturell kongruent, wenn $n = m$ und es eine Permutation ρ von $\{1, \ldots, n\}$ gibt, für die $P_i \equiv Q_{\rho(i)}, 1 \leq i \leq n$ gilt, wobei die vorzunehmenden Substitutionen zu beachten sind.

Zwei Prozesse, die identisch sind, sind natürlich strukturell kongruent. Dieser Fall muss eigentlich nur noch für **0**-Prozesse geprüft werden. Die anderen Fälle, dass ein Prozess eine Summen-, Parallel- oder Sequenzkomposition oder eine Referenz ist, wurden bereits beschrieben.

Um die Prüfung abzuschließen, muss versichert werden, dass bei den berechneten Substitutionen nicht zwei gebundene Namen durch ein und denselben ersetzt wird. Dies würde die Invarianz unter struktureller Kongruenz von $|bn(-)|$ für Prozesse aus $\mathcal{P}_\mathbf{0}$ verletzen.

Die folgende Funktion nimmt zwei Prozesse aus $\mathcal{P}_{s\nu}$ entgegen und liefert *true* genau dann, wenn die Prozesse strukturell kongruent sind. Dazu werden Prozesse, sowie ihre Mengen gebundener Namen und zwei leere Substitutionen an die rekursive Hauptfunktion zur Entscheidung struktureller Kongruenz weitergegeben. Der Ausdruck $P \equiv Q$ ist im Kontext des Algorithmus als „$P \equiv Q$ unter der zusätzlichen Annahme, dass bn_P und bn_Q die gebundenen Namen darstellen“ zu verstehen.

Listing 6.1: Strukturelle Kongruenz

```
1 function structurallyCongruent(in P, in Q)
2       assert P,Q ∈ 𝒫_sν
3       result := structurallyCongruent(P, Q, bn(P), bn(Q), ∅, ∅)
4       ensure result ⇔ P ≡ Q
```

```
5       return result
6   end function
7
8   function structurallyCongruent(in P, in Q, in bn_P, in bn_Q,
9                                  inout σ_P, inout σ_Q)
10      assert P,Q ∈ 𝒫_sν
11      let P = ν b̃_1.π_1. ··· .ν b̃_n.π_n.P' ∧ P' ≠ ν b̃_{n+1}.π_{n+1}.P''
12      let Q = ν b̃'_1.π'_1. ··· .ν b̃'_m.π'_m.Q' ∧ Q' ≠ ν b̃'_{m+1}.π'_{m+1}.Q''
13
14      if P ≠_𝒫 Q then
15          ensure P ≢ Q
16          return false
17      end if
18
19      if ¬ checkPrefixes(P, Q, bn_P, bn_Q, σ_P, σ_Q) then
20          ensure P ≢ Q
21          return false
22      end if
23
24      if P' = Q' then
25          ensure P ≡ Q
26          return true
27      end if P' = K⌊x_1, ..., x_n⌋ ∧ Q' = K⌊y_1, ..., y_n⌋ then
28          for i = 1...n do
29              if ¬ subst(x_i, y_i, bn_P, bn_Q, σ_P, σ_Q) then
30                  ensure P ≢ Q
31                  return false
32              end if
33          end for
34      else if P' = P_1; ...; P_n ∧ Q' = Q_1; ...; Q_m then
35          P_flat := flatten(P')
36          Q_flat := flatten(Q')
37          let P_flat = ν ã_P.(P''_1 | ... | P''_k)
38          let Q_flat = ν ã_Q.(Q''_1 | ... | Q''_j)
```

```
39          if k ≠ j then
40              ensure P ≢ Q
41              return false
42          end if
43          for i = 1...n do
44              if ¬ structurallyCongruent(P_i, Q_i, bn_P, bn_Q,
45                                                     σ_P, σ_Q) then
46                  ensure P ≢ Q
47                  return false
48              end if
49          end for
50      else if P' = P_1 + ··· + P_n ∧ Q' = Q_1 + ··· + Q_n then
51          if ¬ checkCommutative((P_1,...,P_n), sort({Q_1,...,Q_n}, ≤_P),
52                                bn_P, bn_Q, σ_P, σ_Q) then
53              ensure P ≢ Q
54              return false
55          end if
56      else if P' = P_1|...|P_n ∧ Q' = Q_1|...|Q_m then
57          P_flat := flatten(P')
58          Q_flat := flatten(Q')
59          let P_flat = νã_P.(P''_1|...|P''_k)
60          let Q_flat = νã_Q.(Q''_1|...|Q''_j)
61          if k ≠ j then
62              ensure P ≢ Q
63              return false
64          end if
65          if ¬ checkCommutative((P''_1,...,P''_k), sort({Q''_1,...,Q''_k}, ≤_P),
66                                bn_P, bn_Q, σ_P, σ_Q) then
67              ensure P ≢ Q
68              return false
69          end if
70      else
71          ensure P ≢ Q
72          return false
```

```
73      end if
74
75      if ∃a, b, c ∈ 𝒩 : (a ≠ b ∧ c ∈ bn_P ∪ bn_Q ∧ (({c/a} ∈ σ_P ∧ {c/b} ∈ σ_P)
76                                   ∨({c/a} ∈ σ_Q ∧ {c/b} ∈ σ_Q))) then
77          ensure P ≢ Q
78          return false
79      end if
80
81      ensure P ≡ Q
82      return true
83  end function
```

6.3.1 Substitutionen

Die Funktion `subst` entspricht dem Prüfen und ggf. Erweitern der Substitutionen: Werden zwei Namen a und b im oben genannten Programmlauf gefunden, die gleich sein sollten, werden sie der `subst`-Funktion übergeben. Diese liefert genau dann $false$, wenn a und b verschiedene freie Namen sind oder nicht ineinander umbenannt werden dürfen. Die Substitutionen bleiben dann unverändert. Sind a und b der gleiche freie Name, so ist nichts zu tun und $true$ wird zurückgeliefert. Handelt es sich bei a und b jeweils um gebundene Namen, so wird versucht, sie ineinander umzubenennen. Erst wird versucht, im ersten Prozess das a umzubenennen und falls es erfolglos ist, wird versucht, das b im zweien Prozess umzubenennen. Ist einer der Versuche erfolgreich so wird die jeweilige Umbenennung in die entsprechende Substitution aufgenommen

Ein solcher Umbenennungsversuch wird mit der zweiten `subst`-Funktion bewerkstelligt. Ihr werden zwei gebundene Namen a, b und die bereits gemerkten Substitutionen σ_P für den Prozess, aus dem a stammt, und σ_Q für den Prozess, aus dem b stammt, übergeben. Gibt es in σ_Q bereits eine Ersetzung von b durch x, so soll auch a statt durch b durch x ersetzt werden. Ist allerdings bereits eine Ersetzung von a in σ_P enthalten und der einzusetzende Name von dem gewünschten b oder x verschieden, so wird der Umbenennungsversuch als fehlgeschlagen gemeldet. Andernfalls wird eine entsprechende Substitution b oder x statt a in σ_P aufgenommen.

Listing 6.2: Strukturelle Kongruenz: Substitutionen

```
1  function subst(in a, in b, in bn_P, in bn_Q, inout σ_P, inout σ_Q)
2      assert
3      if a ∉ bn_P ∧ b ∉ bn_Q then // a,b freie Namen
4          // dürfen nicht in Substitutionen eingetragen werden
5          // Gleichheit ist hergestellt, wenn a = b
6          return a = b
7      else if a ∈ bn_P ∧ b ∈ bn_Q then // a,b gebunden
8          if subst(a, b, σ_P, σ_Q) then
9              return true
10         else
11             return subst(b, a, σ_Q, σ_P)
12         end if
13     else // a frei und b gebunden, oder umgekehrt
14         // keine Umbenennung erlaubt
15         return false
16     end if
17 end function
18
19 function subst(in a, in b, inout σ_1, in σ_2)
20     if ∃x ∈ N : ({x/b} ∈ σ_2) then // b wird durch x ersetzt
21         // wenn, dann a ebenfalls durch x ersetzen
22         b := x
23     end if
24     if ∃x ∈ N \ {b} : {x/a} ∈ σ_1 then
25         // a wird bereits durch einen anderen Namen ersetzt
26         // Ersetzungsversuch gescheitert
27         return false
28     else // neue Ersetzung eintragen
29         σ_1 := σ_1 ∪ {{b/a}}
30         return true
31     end if
32 end function
```

6.3.2 Präfixe

Die `checkPrefixes`-Funktion durchläuft die Präfixe, die keine erfüllten Wächter sind, zweier Prozesse P, Q synchron. Dabei wird versucht, die im Präfixpaar π_P, π_Q vorkommenden Namen jeweils ineinander umzubenennen. Handelt es sich beispielsweise bei den Präfixen um Sendeaktionen, so müssen die Kanalnamen und die versendeten Namen ineinander umbenannt werden. Bei nicht erfüllten Wächtern sind wegen $[a = b] = [b = a]$ gegebenenfalls zwei verschiedene Umbenennungen auszuprobieren.

Listing 6.3: Strukturelle Kongruenz: Präfixe

```
1  function checkPrefixes(in P, in Q, in bn_P, in bn_Q,
2                        inout σ_P, inout σ_Q)
3      assert true
4      let P = π_P1. ... .π_Pn.P' ∧ Q = π_Q1. ... .π_Qm.Q' ∧ P' ≠ π'_P.P'' ∧ Q' ≠ π'_Q.Q''
5      j := 1
6      k := 1
7      while j ≤ n ∧ π_Pj = [a = a] do j := j + 1 end while
8      while k ≤ m ∧ π_Qk = [a = a] do k := k + 1 end while
9      while j ≤ n ∧ k ≤ m do
10         π_P := π_Pj
11         π_Q := π_Qk
12         if (π_P = a(b_1, ..., b_n) ∧ π_Q = a'(b'_1, ..., b'_n))
13             ∨(π_P = ā⟨b_1, ..., b_n⟩ ∧ π_Q = ā'⟨b'_1, ..., b'_n⟩) then
14             if ¬ subst(a, a', bn_P, bn_Q, σ_P, σ_Q) then
15                 ensure P ≢ Q mit σ_P und σ_Q
16                 return false
17             end if
18             for i = 1...n do
19                 if ¬ subst(b_i, b'_i, bn_P, bn_Q, σ_P, σ_Q) then
20                     ensure P ≢ Q mit σ_P und σ_Q
21                     return false
22                 end if
23             end for
24             return true
```

```
25            else if πP = [a = b] ∧ πQ = [a' = b'] then
26                σ'P := σP
27                σ'Q := σQ
28                if ¬ subst(a, a', bnP, bnQ, σ'P, σ'Q)
29                    ∨ ¬ subst(b, b', bnP, bnQ, σ'P, σ'Q) then
30                    if ¬ subst(a, b', bnP, bnQ, σP, σQ)
31                        ∨ ¬ subst(b, a', bnP, bnQ, σP, σQ) then
32                        ensure P ≢ Q mit σP und σQ
33                        return false
34                    end if
35                else
36                    σP := σ'P
37                    σQ := σ'Q
38                end if
39            else if πP = τ = πQ then
40                return true
41            else
42                ensure P ≢ Q
43                return false
44            end if
45            j := j + 1
46            k := k + 1
47            while j ≤ n ∧ πPj = [a = a] do j := j + 1 end while
48            while k ≤ m ∧ πQk = [a = a] do k := k + 1 end while
49        end while
50
51        if j ≤ n ∨ k ≤ m then
52            ensure P ≢ Q
53            return false // ungleich viele echte Präfixe
54        end if
55    end function
```

6.3.3 Kommutative Operatoren

Noch zu klären ist das Finden einer eingangs geschriebenen Permutation, wenn zwei Prozesse mit kommutativen Operatoren vorliegen,. Die rekursive Funktion `checkCommutative` durchläuft alle, unter Beachtung der beschriebenen Ordnung $\leq_{\mathcal{P}}$ auf Prozessen, sinnvollen Permutationen, bis eine passende gefunden wird oder sichergestellt ist, dass es keine passende Permutation gibt. Eingegeben werden zwei gleichlange Listen von Prozessen $(P_1, \ldots, P_n)$ und $(Q_1, \ldots, Q_n)$, bei denen die Q-Liste im Sinne von $\leq_{\mathcal{P}}$ geordnet ist. Sind die Listen leer, so ist nicht zu tun. Ansonsten wird ein beliebiges Element P_k der P-Liste herausgenommen und für jedes Element Q_i der Q-Liste geprüft, ob es strukturell kongruent zu P_k ist. Es werden Kopien der Substitutionen bei der Prüfung verwendet, da sich, auch wenn $Q_i \equiv P_k$ gilt, im weiteren Programmlauf herausstellen kann, dass die berechneten Substitutionen unbrauchbar sind. Ist $Q_i \equiv P_k$, so wird nun rekursiv fortgefahren, eine entsprechende Permutation für $(P_1, \ldots, P_{k-1}, P_{k+1}, \ldots, P_n)$ und $(Q_1, \ldots, Q_{i-1}, Q_{i+1}, \ldots, Q_n)$ mit den Kopien der Substitutionen zu finden. Kann keine Permutation gefunden werden, so sind die Kopien der Substitutionen ohne Wert. Liefert der rekursive Aufruf hingegen ein $true$ für die Existenz einer entsprechenden Permutation, so gibt es eine Permutation ρ von $\{1, \ldots, n\}$ mit $\rho(k) = i$, für die $\forall 1 \leq j \leq n : P_{\rho(j)} \equiv Q_j$ gilt und es kann $true$ als Zeichen der Existenz geliefert werden. Die Kopien ersetzen dabei die eingegebenen Substitutionen. Endet die Schleife über die Q_i ohne $true$ liefern zu können, so wurden bereits alle Permutationen untersucht. Es wird $false$ zurückgeliefert, da es keine Permutation mit der gewünschten Eigenschaft gibt.

Ist die Q-Liste im Sinne von $\leq_{\mathcal{P}}$ geordnet, so kann die aufsteigende Iteration über die Elemente der Q-Liste frühzeitig beendet werden, falls $P_k <_{\mathcal{P}} Q_i$ gilt, da dann wegen der Transitivität von $\leq_{\mathcal{P}}$ ebenfalls für alle $Q_j, j \geq i$ die Eigenschaft $P_k <_{\mathcal{P}} Q_j$ gilt. In diesem Fall gibt es also in der Q-Liste ab Q_i keinen zu P_k strukturell kongruenten Prozess und die Schleife kann mit dem Zurückliefern von $false$ beendet werden, da es keine passende Permutation gibt.

Listing 6.4 stellt den beschriebenen Algorithmus dar.

Listing 6.4: Strukturelle Kongruenz: Kommutativität

```
1 function checkCommutative(in (P_1,...,P_n), in (Q_1,...,Q_n),
2                           in bn_P, in bn_Q, inout σ_P, inout σ_Q)
```

```
 3      assert ∀1 ≤ i < j ≤ n : Q_i ≤_P Q_j
 4      if n = 0 then
 5          return true
 6      end if
 7
 8      wähle k ∈ {1, ..., n}
 9      P' := (P_1, ..., P_{k−1}, P_{k+1}, ..., P_n)
10
11      for i = 1...n do
12          ensure ∀ Permutation ρ von {1, ..., n} : (ρ(k) < i
13                                      ⇒ ∃1 ≤ j ≤ n : P_ρ(j) ≢ Q_j)
14          if P_k <_P Q_i then
15              ensure ∀i ≤ j ≤ n : Q_j ≢ P_k
16              ensure ∀ Permutation ρ von {1, ..., n}∃1 ≤ j ≤ n : P_ρ(j) ≢ Q_j
17              return false
18          end if
19          σ'_P := σ_P
20          σ'_Q := σ_Q
21          if structurallyCongruent(P_k, Q_i, bn_P, bn_Q, σ'_P, σ'_Q)
22              ∧ checkCommutative(P', (Q_1, ..., Q_{i−1}, Q_{i+1}, ..., Q_m),
23                          bn_P, bn_Q, σ'_P, σ'_Q) then
24              σ_P := σ'_P
25              σ_Q := σ'_Q
26              ensure ∃ Permutation ρ von {1, ..., n}∀1 ≤ j ≤ n : P_ρ(j) ≡ Q_j
27              return true
28          end if
29          ensure ∀ Permutation ρ von {1, ..., n} : (ρ(k) ≤ i
30                                      ⇒ ∃1 ≤ j ≤ n : P_ρ(j) ≢ Q_j)
31      end for
32
33      ensure ∀ Permutation ρ von {1, ..., n}∃1 ≤ j ≤ n : P_ρ(j) ≢ Q_j
34      return false
35  end function
```

6.4 Korrektheit und Aufwandabschätzung

Die Korrektheit des vorgestellten Algorithmus kann über die Ableitung struktureller Kongruenz argumentiert werden. Ein formaler Beweis steht noch aus.

Abgesehen von der `checkCommutative`-Funktion wird die strukturelle Kongruenz von dem vorgestellten Algorithmus in zur Prozessgröße linearer Zeit entschieden. Einzig die `checkCommutative`-Funktion, welche die Existenz einer bestimmten Permutation entscheidet, setzt Backtracking ein. Es werden höchstens $n!$ verschiedene Permutationen untersucht, wenn die eingegebenen Prozesslisten n Einträge haben. Damit liegt die Komplexität des Algorithmus in $\mathcal{O}(n!)$. Die eingeführte Ordnung auf Prozessen hilft, diesen worst case in den meisten Fällen zu vermeiden, indem in der `checkCommutative`-Funktion möglichst früh Permutationen ausgeschlossen und nicht weiter untersucht werden.

6.5 Bemerkungen zur Implementierung

Da bei der Berechnung der Semantiken und besonders in der `checkCommutative`-Funktion wiederholt gleiche Paare von Prozessen auf strukturelle Kongruenz geprüft werden müssen, werden die Ergebnisse im Sinne dynamischer Programmierung zwischengespeichert. Zu jedem Paar von (Teil-) Prozessen wird somit höchstens einmal berechnet, ob sie strukturell kongruent sind. Die ungünstige Komplexität $\mathcal{O}(n!)$ des Algorithmus wird zwar nicht verbessert, kommt im Allgemeinen aber nicht im ganzen Ausmaß zum Vorschein. Die assoziativen Speicher, in denen die Ergebnisse der Berechnungen gespeichert werden, haben die Eigenschaft, Assoziationen zu vergessen, sobald die zugehörigen Prozesse nicht mehr benötigt werden.

7 Überdeckungsgraph-Algorithmen

Bei der Berechnung der vorgestellten Semantiken werden gewisse Erreichbarkeitsfragen gestellt. Zu ihrer Beantwortung werden Überdeckungsgraphen eingesetzt.

7.1 Aufbau eines Überdeckungsgraphen

Die strukturelle Semantik $\mathcal{N}[\![P]\!]$ eines Prozesses P enthält, wie in Definition 27 beschrieben, die Stellenmenge $S := Frag(Reach(P))/_{\equiv}$. Da nur von P aus erreichbare Prozesse für die Berechnung der Stellen herangezogen werden, ist zu prüfen, welche Prozesse erreicht werden können. Die erreichbaren Zustände zu bestimmen ist wegen der Isomorphie der Transitionssysteme von P und $\mathcal{N}[\![P]\!]$ (vgl. Theorem 1 in [Mey07b]) in beiden Domänen gleich schwierig zu beantworten (modulo dem Aufwand der $retrieve(-)$-Funktion, siehe Definition 29) und kann so in der Welt der Petrinetze beantwortet werden.

Für Reaktionen eines einzelnen Fragments ist die Erreichbarkeit einfach festzustellen: Existiert eine Stelle $[F] \in S$, so gilt: $\forall Q \in \mathcal{P} : (F \rightarrow Q \Rightarrow Frag(Q) \subseteq_{\equiv} S)$. Es sind also lediglich die Reaktionen von F zu berechnen.

Eine schwierigere Frage ergibt sich bei der Betrachtung von fragmentübergreifenden Kommunikationen: Können zwei Stellen $[F_1], [F_2] \in S$ kommunizieren? Dann würde nämlich folgendes gelten: $\forall Q \in \mathcal{P} : (F_1 | F_2 \rightarrow Q \Rightarrow Frag(Q) \subseteq_{\equiv} S)$

Die $retrieve(-)$-Funktion sagt, (1) dass zwei Fragmente kommunizieren können, wenn sie entweder jemals gleichzeitig mit mindestens einer Marke belegt sind, oder, fals es sich um ein einziges Fragment handelt, (2) dass das Fragment „mit sich selbst“ kommunizieren kann, wenn es „jemals“ mit mindestens zwei Marken belegt ist. Dieser Sachverhalt verdeutlicht, dass nicht nur die Struktur des bisher erzeugten Netzes, sondern auch die Markierung betrachtet werden müssen.

7.1.1 Komplexität

Da die Semantik spezielle Netze erzeugt, bei denen Transitionen entweder eine oder zwei Stellen im Vorbereich haben, liegt die Vermutung nahe, dass es ein effizientes Verfahren gibt, mit dem festgestellt werden kann, ob zwei Stellen gleichzeitig markiert werden können, bzw. ob eine Stelle mit mindestens zwei Marken belegt werden kann. Dieses Problem kann auch mit der Frage beantwortet werden, ob es zu einer bestimmten Markierung eine Folgemarkierung gibt, unter der eine Transition mit den beiden Stellen im Vorbereich feuern kann, bzw. ob eine Transition mit einer Zweifachkante von einer Stelle feuern kann. Da es äquivalent und sprachlich einfacher ist, wird auf die Komplexität des Problems eingegangen, ob eine Transition jemals feuerbar ist oder nicht.

Leider ist das bekannte und schwierige Teil-Erreichbarkeitsproblem TEP polynomiell auf dieses Problem reduzierbar.

Definition 37 (Teil-Erreichbarkeitsproblem TEP)**.** Das TEP ist die Frage, ob von einem bestimmten Zustand ausgehend auf einer bestimmten Teilmenge der Stellen eine bestimmte Markierung erreicht werden kann, [PW02, Def. 5.1.1, S.120ff].

$$\begin{aligned} \mathsf{TEP} &= \{(N, M, M', S') \mid N = (S, T, F) \wedge M, M' \in \mathbb{N}^S \wedge S' \subseteq S \\ &\quad \wedge \exists \sigma \in T^* \exists \widehat{M}' : (M\ [\sigma\rangle_N\ \widehat{M}' \wedge \forall s \in S' : \widehat{M}'(s) = M'(s))\} \end{aligned}$$

Ist ein TEP (N, M, M', S') gegeben, so wird eine Transition t_{test} mit $\forall s \in S' : F(s, t_{test}) = M'(s) \wedge \forall s \in S \setminus S' : F(s, t_{test}) = 0 \wedge \forall s \in S : F(t_{test}, s) = 0$ eingefügt. Transition t_{test} kann also feuern, wenn $\exists \sigma \in T^* \exists \widehat{M}' : (M\ [\sigma\rangle_N\ \widehat{M}' \wedge \forall s \in S' : \widehat{M}'(s) \geq M'(s))$. Sei $S' = \{s_1, \ldots, s_n\}$. Weiter werden Transitionen $t_1, \ldots, t_n$ eingefügt mit $\forall 1 \leq i \leq n : (F(s_i, t_i) = M'(s_i) + 1 \wedge \forall s \in S \setminus S' : F(s, t_i) = 0)$. Transition t_i kann also feuern, wenn mehr als $M'(s_i)$ Marken auf Stelle s_i liegen. Zur Reduktion reicht dies noch nicht aus, da beim TEP allgemeine Netze eingegeben werden und die Semantik Netze mit bestimmten Eigenschaften erzeugt.

Dass die von der strukturellen Semantik konstruierten Netze eine spezielle Form haben, ist zu vernachlässigen, da endliche Petrinetze und strukturell stationäre Prozesse „gleichmächtig“ sind: Mit [Mey07b] kann zu jedem strukturell stationären Prozess ein endliches Petrinetz mit isomorphem Transitionssystem konstruiert werden. Mit [AM02] wird zu jedem endlichen Petrinetz ein Prozess mit isomorphem Transitionssystem konstruiert,

der ausschließlich freie Namen verwendet. Wie [Mey07b] in Theorem 2 anmerkt, ist ein Prozess mit ausschließlich freien Namen, strukturell stationär.

Ohne Einschränkung kann also von einem allgemeinen Petrinetz ausgegangen werden, für das die Frage nach einer erreichbaren, transitionsaktivierenden Markierung beantwortet werden muss.

Genau dann, wenn in N von M aus ein Zustand $\widehat{M'}$ erreicht werden kann, in welchem $\widehat{M'}\ [t_{test}\rangle \wedge \forall 1 \leq i \leq n : \neg(\widehat{M'}\ [t_i\rangle)$ gilt, ist das übersetzte TEP erfüllt.

Damit ist das hier zu entscheidende Problem mindestens so schwer wie das TEP. Dass es ebenso schwer ist, folgt daher, dass es selbst einfach in das TEP umgeformt werden kann. Jeder Algorithmus, der das hier gestellte Problem entscheidet ist also mindestens EXPSPACE kompliziert. Mittels eines Überdeckungsgraphen lässt es sich lösen.

7.1.2 Vorgehen

Da ein Überdeckungsgraph parallel zum Petrinetz aufgebaut wird, kann der bekannte Algorithmus aus Definition 3.2.26 in [PW02, S. 66ff] nicht wörtlich übernommen werden. Ziel ist es, eine Vorschrift zu finden, mit der ein Überdeckungsgraph von einem Petrinetz so erweitert werden kann, dass ein Überdeckungsgraph für ein neues Petrinetz entsteht, welches das vorherige um eine zusätzliche Transition und ihren Nachbereich erweitert.

7.1.3 Aufbau eines Überdeckungsgraphen

Sei $\mathbb{S}$ die Menge aller Stellen, die in Petrinetzen vorkommen können. Ist $S \subset \mathbb{S}$ eine endliche Menge von Stellen, so kann eine Markierung $M \in \mathbb{N}_\omega^S$ kanonisch auf $M_{ext} \in \mathbb{N}_\omega^{\mathbb{S}}$ fortgesetzt werden:

$$\forall s \in \mathbb{S} : M_{ext}(s) = \begin{cases} M(s), & \text{falls } s \in S \\ 0, & \text{sonst} \end{cases} .$$

Analog kann M_{ext} als M dargestellt werden, wenn $supp(M_{ext})$, also die Nichtnullstellenmenge von M_{ext}, endlich ist.

Seien nun $N = (S \subset \mathbb{S}, T, F, M_0 \in \mathbb{N}_\omega^{\mathbb{S}})$ und $N' = (S', T', F', M_0)$ mit $T' = T \cup \{t_{neu}\}$, $S' = S \cup t_{neu}^\bullet \subset \mathbb{S}$. Dabei sei t_{neu} eine neue Transition, deren Vorbereich in S liegt. Alle

neuen Stellen liegen im Nachbereich von t_{neu}. Sie sind unter der Anfangsmarkierung unmarkiert und besitzen neben einem leeren Nachbereich alle den gemeinsamen Vorbereich $\{t_{neu}\}$. In einer Formel ausgedrückt, die auch als Vorbedingung für den hier vorgestellten Algorithmus Einsatz findet:

$$
\begin{aligned}
PRE \quad = \quad & t_{neu} \notin T \\
\wedge \quad & {}^{\bullet}t_{neu} \subseteq S \\
\wedge \quad & t_{neu}{}^{\bullet} \supseteq S' \setminus S \\
\wedge \quad & \forall s \in S' \setminus S : M_0(s) = 0 \\
\wedge \quad & F'|_{S \times T \cup T \times S} = F \\
\wedge \quad & \forall t \in T \forall s \in S' \setminus S : F'(s,t) = F'(t,s) = 0
\end{aligned}
$$

Aus $Cov(N) = (V, E, T)$ mit $V \subseteq \mathbb{N}_\omega^{\mathbb{S}}, E \subseteq \mathbb{N}_\omega^{\mathbb{S}} \times T \times \mathbb{N}_\omega^{\mathbb{S}}$, dem Überdeckungsgraph von N, soll nun $Cov(N') = (V', E', T')$ mit $V' \subseteq \mathbb{N}_\omega^{\mathbb{S}}, E' \subseteq \mathbb{N}_\omega^{\mathbb{S}} \times T' \times \mathbb{N}_\omega^{\mathbb{S}}$, konstruiert werden. Der in Listing 7.1 beschriebene Algorithmus nimmt $Cov(N), N, F'$ und t_{neu} als Eingaben, und berechnet $Cov(N')$. Die ab Zeile 28 beschriebene Prozedur betrachtet einen Knoten des Überdeckungsgraphen in Hinsicht auf eine Transition. Diese Prozedur entspricht dem Hauptteil des in [PW02] vorgestellten Algorithmus zum Aufbau eines Überdeckungsgraphen: Wenn von einem Knoten M mit Transition t ein Knoten M' erreicht wird, so werden alle Pfade im bisher erzeugten Graphen von M_0 nach M betrachtet und entsprechend ω-Werte in M' eingefügt. Ist M' nicht im Graphen enthalten, so wird der Knoten eingefügt und auch in der Menge der zu betrachtenden Knoten Neu vermerkt.

Um einen Überdeckungsgraphen von N' aus $Cov(N)$ zu errechnen, werden in der Schleife ab Zeile 11 alle Knoten in V durchlaufen und wird unter Anwendung der `expand`-Prozedur geprüft, ob sie t_{neu} aktivieren.

Die Schleife ab Zeile 15 arbeitet die neu entstandenen Knoten ab und entspricht wieder dem in [PW02] vorgestellten Algorithmus.

Es zeigt sich, dass die oben eingeführte Schreibweise von Zuständen als Elemente aus $\mathbb{N}_\omega^{\mathbb{S}}$ den Vorteil hat, dass keine Umrechnung von Elementen aus $\mathbb{N}_\omega^{S}$ in Elemente aus $\mathbb{N}_\omega^{S'}$ nötig ist. Beim Ausführen von `addTransition` müssen die Knoten aus V nicht verändert werden, da die Stellen in $S' \setminus S$ unter M_0 unmarkiert sind und nicht von Transitionen in

T markiert werden können. Es ist klar, dass $E \subseteq E'$, da $T \subset T'$.

Listing 7.1: Erweiterung eines Überdeckungsgraphen

```
1  function addTransition(in (V,E,T), in (S,T,F,M_0),
2                         in F', in t_neu, inout C)
3      // C ist die Menge der neuen Kommunikationspartner.
4      // Sie wird im weiteren gefüllt.
5      (V',E',T') := (V,E,T ∪ {t_neu}) // alten Graphen übernehmen
6      S' := S ∪ t_neu• // neue Stellen übernehmen
7      assert (V,E,T) = Cov((S,T,F,M_0)) ∧ PRE
8
9      Neu := ∅ // einzufügende Knoten
10
11     for each M ∈ V do // neue Transition betrachten
12         expand((V',E',T'), (S',T',F',M_0), Neu, M, t_neu)
13     end for
14
15     while Neu ≠ ∅ do // gängiger Algorithmus
16         wähle M ∈ Neu
17         Neu := Neu \ {M}
18         C := C ∪ checkCommunications(M)
19         for each t ∈ T' do
20             expand((V',E',T'), (S',T',F',M_0), Neu, M, t)
21         end for
22     end while
23
24     ensure (V',E',T') = Cov((S',T',F',M_0))
25     return (V',E',T')
26 end function
27
28 procedure expand(inout (V',E',T'), in (S',T',F',M_0), inout Neu,
29                  in M, in t)
30     assert ∃ Pfad von M_0 nach M in E'
31
```

```
32      if M [t⟩ M' then // hier wird F' benutzt
33          for each M̃ ∈ V' auf einem Weg von M₀ nach M do
34              if M̃ < M' then // M̃ wird von M' überdeckt
35                  for each s ∈ S' do
36                      if M̃(s) < M'(s) then
37                          M'(s) := ω
38                      end if
39                  end for
40              end if
41          end for
42          if M' ∉ V' then
43              V' := V' ∪ {M'}
44              Neu := Neu ∪ {M'}
45          end if
46          E' := E' ∪ {(M, t, M')}
47      end if
48
49      ensure M [t⟩ M' ⇔ ∃M̃ ∈ V' : (M' ≤ M̃ ∧ (M, t, M̃) ∈ E'
50                                    ∧(M̃ neu in V' ⇒ M̃ ∈ Neu))
51  end procedure
52
53  function checkCommunications(in M)
54      C := ∅
55      for each s₁ ∈ supp(M) do
56          if M(s₁) ≥ 2 then
57              C := C ∪ {{s₁, s₁}}
58          end if
59          for each s₂ ∈ supp(M) do
60              if s₁ ≠ s₂ then
61                  C := C ∪ {{s₁, s₂}}
62              end if
63          end for
64      end for
65      ensure ∀(c₁, c₂) ∈ C∃v ∈ V : ((c₁ = c₂ ∧ v(c₁) ≥ 2)
```

```
66                                        ∨(c1 ≠ c2 ∧ v(c1) ≥ 1 ∧ v(c2) ≥ 1)
67        return C
68    end procedure
```

Für die durch die behandelten Semantiken erzeugten Petrinetze kann on-the-fly ein Überdeckungsgraph sukzessive aufgebaut werden. Der initiale Überdeckungsgraph ergibt sich zum initialen Netz $N_0 = (\varnothing, \varnothing, F_0, M)$ als $Cov(N_0) = (\{M\}, \varnothing, \varnothing)$, wobei F_0 die leere Abbildung ist und M der Anfangszustand des initialen Netzes ist. Siehe hierzu Abschnitt 3.1.

7.1.4 Korrektheit

Würde man die Zuweisung $Neu := \varnothing$ durch $Neu := \{M_0\}$ ersetzen, so würde der ganze Überdeckungsgraph neu berechnet, da die Schleife ab Zeile 11 den Graphen und die Neu-Menge höchstens erweitert und der Algorithmus ab Zeile 15 wieder der Vorlage entspricht. Mit der initialen Zuweisung $(V', E', T') = (V, E, T \cup \{t_{neu}\})$ wird der ursprüngliche Graph übernommen. Da die neuen Stellen $S' \setminus S$ ausschließlich von t_{neu} markiert werden können und es unter M_0 nicht sind, bliebe der Graph unverändert, wenn t_{neu} nicht feuern könnte - insbesondere wäre dann $Cov(N') = Cov(N)$. Es sind also alle bereits erzeugten Knoten auf Aktivierung von t_{neu} zu prüfen. Dies geschieht in der Schleife ab Zeile 11 mittels der bereits bekannten Prozedur `expand`. Neue Knoten werden als Rand in Neu vermerkt, dem Graphen hinzugefügt und der bekannte Algorithmus anschließend wie gewohnt fortgesetzt.

Damit errechnet die in Listing 7.1 beschriebene Vorschrift einen gewünschten Überdeckungsgraphen $Cov(N') = (V', E', T')$, für den die bekannten Eigenschaften gelten. Unter anderem entspricht jeder in M_0 beginnenden Transitionssequenz in E' eine in M_0 aktivierte Feuersequenz in N' und umgekehrt:

$$\forall t_1, \ldots, t_n \in T' \forall M_1, \ldots, M_n \in \mathbb{N}_\omega^{\mathbb{S}} \exists M'_1 \geq M_1, \ldots, M'_n \geq M_n \in V' :$$
$$M_0 \left[t_1\right\rangle M_1 \cdots M_{n-1} \left[t_n\right\rangle M_n \Leftrightarrow (M_0, t_1, M'_1), \ldots, (M'_{n-1}, t_n, M'_n) \in E'.$$

7.1.5 Ausgaben

Überdeckungsgraphen werden vom Compiler benötigt, um herauszufinden, welche Paare von Stellen jeweils gleichzeitig markiert werden können, so dass die Fragmente eine

Kommunikation im Sinne des π-Kalküls durchführen können. Auch gilt es für bestimmte Transitionen zu berechnen, wie oft sie höchstens feuern können. Der zweite Aspekt wird in Abschnitt 7.2 näher betrachtet.

Welche Stellenpaare markiert werden können, lässt sich prüfen, wenn in Zeile 43 ein neuer Knoten M in den Graphen eingefügt wird. Kommunizieren können alle Paare verschiedener Stellen, die unter M markiert sind und alle Paare derselben Stelle, die unter M mit mindestens zwei Marken belegt sind: $supp(M) \times supp(M) \setminus \{(s, s) \mid s \in supp(M), M(s) < 2\}$. Dieses Vorgehen setzt keinen vollständigen Überdeckungsgraphen voraus. Die so gefundenen Stellenpaare werden an den Compiler zurückgeliefert, der dann entsprechende Kommunikationen berechnet, die zu einer Erweiterung des Netzes führen.

7.1.6 Markieren überdeckter Zustände

Eine Bestrebung beim Aufbau eines Überdeckungsgraphen ist es, ihn klein aber aussagekräftig zu halten. Zustände des Überdeckungsgraphen (V, E, T), die von anderen Zuständen desselben Graphen überdeckt werden, die auf längeren Pfaden liegen, werden im Rahmen der Anforderungen nicht benötigt. Grundsätzlich gilt: $\forall M, M' \in V \forall t \in T : (M < M' \Rightarrow (M\,[t\rangle \Rightarrow M'\,[t\rangle))$ – jede Transition, die in M aktiviert ist, ist auch in M' aktiviert.

Es liegt nahe, die überdeckten Zustände nicht weiter zu expandieren. Dazu kann eine Menge $C \subseteq \mathbb{N}_\omega^{\mathbb{S}}$ in den Überdeckungsgraphen und die `expand`-Prozedur wie in Listing 7.2 beschrieben eingebunden werden. In Zeile 5 wird sichergestellt, dass nur an solchen Knoten ausgehende Kanten erzeugt werden, die nicht überdeckt sind. Mit Anweisung in Zeile 8 werden alle vom neuen Zustand überdeckten Zustände auf dem Weg von M_0 markiert.

Die Menge der überdeckten Zustände kann von der `addTransition`-Funktion aus dem vorangegangenen Überdeckungsgraphen übernommen und erweitert werden.

Das Aussehen und die Größe des Überdeckungsgraphen hängt von der Auswahlreihenfolge der Transitionen und der betrachteten Knoten ab. In jedem Fall aber bleiben diese Graphen endlich.

Listing 7.2: Markieren überdeckter Zustände

```
1  procedure expand(inout (V', E', T', C'), in (S', T', F', M_0), inout Neu,
2                   in M, in t)
3      assert ∃ Pfad von M_0 nach M in E'
4
5      if M ∉ C' ∧ M [t⟩ M' then // hier wird F' benutzt
6          for each M̃ ∈ V' auf einem Weg von M_0 nach M do
7              if M̃ < M' then // M̃ wird von M' überdeckt
8                  C' := C' ∪ M̃
9                  for each s ∈ S' do
10                     if M̃(s) < M'(s) then
11                         M'(s) := ω
12                     end if
13                 end for
14             end if
15         end for
16         if M' ∉ V' then
17             V' := V' ∪ {M'}
18             Neu := Neu ∪ {M'}
19         end if
20         E' := E' ∪ {(M, t, M')}
21     end if
22
23     ensure M [t⟩ M' ⇒ ∃M̃, M̃' ∈ V' : (M ≤ M̃ ∧ M' ≤ M̃' ∧ (M̃, t, M̃') ∈ E'
24                                      ∧(M̃' neu in V' ⇒ M̃' ∈ Neu))
25 end procedure
```

7.1.6.1 Korrektheit

Sei $M_0 \cdots \widetilde{M} \cdots M$ ein Pfad, bei dem $M > \widetilde{M}$ gilt. Da $M > \widetilde{M}$ gilt $\forall t \in T : (\widetilde{M}\,[t\rangle \Rightarrow M\,[t\rangle)$, so dass alle mit $M_0 \cdots \widetilde{M}$ beginnenden Pfade über M verlängert werden können. Das Maximum bezüglich der Anzahl der Feuerschritte einer Transition bleibt also unverändert.

Verletzt wird die in Unterabschnitt 7.1.4 genannte Eigenschaft, dass jeder Feuersequenz des Petrinetzes eine Transitionenfolge im Überdeckungsgraphen entspricht. Diese Eigenschaft gilt in einer abgeschwächten Version, die für den Compiler genügt: Zu jeder in M_0 aktivierten Feuersequenz gibt es eine Transitionenfolge im Überdeckungsgraphen, die alle gefeuerten Transitionen in derselben Reihenfolge enthält und zu jedem dabei im Petrinetz erreichten Zustand einen überdeckenden enthält. Diese Transitionenfolge kann eben durch die Verlängerung der ursprünglichen Folge um den Teil $\widetilde{M} \cdots M$ gefunden werden. Also:

$$\begin{gathered} \forall t_1, \ldots, t_n \in T' \forall M_1, \ldots, M_n \in \mathbb{N}_\omega^{\mathbb{S}} \exists t'_1, \ldots, t'_m \in T' \exists M'_1, \ldots, M'_m \in V' \\ \exists i_1 < i_2 < \cdots < i_n, j_1 < j_2 < \cdots < j_n \in \mathbb{N} : \\ M_0 \left[t_1\right\rangle M_1 \cdots M_{n-1} \left[t_n\right\rangle M_n \Rightarrow ((M_0, t_1, M'_1), \ldots, (M'_{m-1}, t_m, M'_m) \in E' \\ \wedge t'_{i_1} = t_1, \ldots, t'_{i_n} = t_n \wedge M'_{j_1} \geq M_1, \ldots, M'_{j_n} \geq M_n) \end{gathered}$$

Die Eigenschaft, dass es zu jeder in M_0 beginnenden Transitionenfolge im Überdeckungsgraphen eine Feuersequenz im Petrinetz gibt, bleibt offensichtlich erhalten.

7.1.6.2 Auswahlreihenfolge

Als Auswahlreihenfolge der Anweisung „wähle $M \in Neu$“ in Zeile 16 in Listing 7.1 bietet sich ein Tiefendurchlauf an. Im Gegensatz zu einem Breitendurchlauf traversiert ein Tiefendurchlauf zu beginn möglichst lange Pfade, so dass potentiell früher Knoten auf den Pfaden überdeckt und damit markiert werden. Ein Tiefendurchlauf wird erreicht, indem Neu als Kellerspeicher umgesetzt wird.

7.1.7 Nebenläufigkeit

Da zum Aufbau eines Überdeckungsgraphen lediglich das Petrinetz als Eingabe benötigt wird und als direkte Ausgaben während des Übersetzungsprozesses nur die in Unterabschnitt 7.1.5 beschriebenen Stellenpaare benötigt werden, wurde die Erzeugung des Überdeckungsgraphen von dem eigentlichen Übersetzungsprozess entkoppelt und in einen eigenen Thread ausgelagert. Wegen des erheblichen Aufwandes, den Überdeckungsgraphen zu berechnen, könnte so auf Mehrprozessorsystemen eine bessere Leistung erzielt werden.

Dazu würden ein Eingabe- und ein Ausgabepuffer eingebunden werden, die eine asynchrone Kommunikation zwischen dem Übersetzer und dem Aufbau des Überdeckungsgraphen ermöglichen.

7.1.8 Aufwandsabschätzung

In Satz 5.2.14 in [PW02, S. 142] wird erläutert, dass die Größe eines Überdeckungsgraphen nicht durch eine, von der Größe des Netzes abhängige, primitiv rekursive Funktion beschränkbar ist. Jeder Algorithmus zum Aufbau des solchen Graphen hat mindestens EXPSPACE-Aufwand. Der hier vorgestellte Algorithmus liegt, wie ebenfalls in [PW02] beschrieben, in EXPSPACE.

7.1.9 Sonderfall geschlossene Prozesse

Lemma 6 in [Mey07b] sagt aus, dass ein Netz $\mathcal{N}[\![P]\!]$ zu $P \in \mathcal{CP}$ kommunikationsfrei ist, d.h. $\forall t \in T : |{}^\bullet t| = 1$. Dieses Lemma lässt sich noch verschärfen durch die Eigenschaft, dass die Kantengewichte von Stellen zu Transitionen höchstens eins sind: $\forall t \in T \forall s \in {}^\bullet t : F(s,t) = 1$.

Eine Kommunikation zwischen zwei markierten Stellen, bzw. einer Stelle mit zwei Marken, verläuft immer über öffentliche Kanäle und damit freie Namen. Für diesen Fall wird die Berechnung des Überdeckungsgraphen für die strukturelle Semantik nicht benötigt, da keine echten Kommunikationen stattfinden können.

Im Rahmen der sequentiellen Semantik wird der Überdeckungsgraph benötigt, falls eine Transition erzeugt wird, die einen neuen freien Namen „erzeugt“. Ob ein Prozess erreicht wird, der das Erzeugen einer solchen Transition zur Folge hat, ist allerdings bei Programmstart nicht bekannt. Der Überdeckungsgraph muss in jedem Fall erzeugt werden.

7.1.10 Bemerkungen zur Implementierung

In Programmen wie PEP wird eine Markierung $M \in \mathbb{N}^S$ eines Petrinetzes $N = (S, T, F, M_0$ als Feld mit $|S|$ Einträgen von natürlichen Zahlen umgesetzt. Dieses naheliegende Vorgehen bringt aber einen erhöhten Aufwand beim Erweitern des Netzes mit sich, da durch

das Hinzufügen von Stellen die Felder der bereits als Knoten im Überdeckungsgraph gespeicherten Markierungen vergrößert werden müssen. Hier kann die Multimengenschreibweise von $M_{ext} \in \mathbb{N}^{\mathbb{S}}$ als Funktion genutzt werden. Da die jeweils neu hinzukommenden Stellen unter M_0 leer sind und frühestens durch die neu hinzugefügte Transition belegt werden können, sind keine weiteren Aktualisierungen vorzunehmen. Die Implementierung verwendet zur Umsetzung der Multimenge eine Hash-Tabelle, einen assoziativen Speicher, um den stets endlichen Träger von M_{ext} abzubilden.

7.2 Feuerbarkeit von Transitionen

Für die starke Faltung (siehe Unterabschnitt 3.1.2.1) ist es notwendig, zu berechnen, wie oft eine Transition höchstens feuern kann, beziehungsweise, ob sie beliebig oft feuern kann. Dies kann am Überdeckungsgraphen entschieden werden, ist allerdings erst nach dem Erzeugen des vollständigen Überdeckungsgraphen möglich.

7.2.1 Rekursive Berechnung

Die folgende Formel drückt aus, dass eine bestimmte Transition $t \in T$ von $M \in V$ so oft feuern kann, wie es $bnd_t(M)$ angibt. Ist $bnd_t(M) = k \in \mathbb{N}_\omega$, so kann t von M aus höchstens k-mal feuern und es gibt einen Pfad von M aus, auf dem t k-mal vorkommt. Ist $bnd_t(M) = \omega$, so kann t von M aus beliebig oft feuern.

$$\begin{aligned} & cor(bnd_t, (V, E, T), M) \\ = \quad & (bnd_t(M) = \omega \Rightarrow \forall i \in \mathbb{N} \exists n \in \mathbb{N} \exists \sigma \in T^* \exists M' \in V : (M \xrightarrow{\sigma}_E M' \wedge p(\sigma)(t) > i)) \\ \wedge \quad & (bnd_t(M) \neq \omega \Rightarrow \begin{pmatrix} & \forall \sigma \in T^* \exists M' \in V : (M \xrightarrow{\sigma}_E M' \wedge p(\sigma)(t) \leq bnd_t(M)) \\ \wedge & \exists \sigma \in T^* \exists M' \in V : (M \xrightarrow{\sigma}_E M' \wedge p(\sigma)(t) = bnd_t(M)) \end{pmatrix}) \end{aligned}$$

Listing 7.3: Berechnen von Feuerbarkeit

```
1 function firingBound(in (V,E,T), in M_0, in t)
2     let bnd_t : V → ℕ_ω
3
4     visited := ∅ ⊆ V
```

```
5       path := ε ∈ E*
6       dfsFiringBound((V,E,T), t, M_0, bnd_t, visited, path)
7
8       ensure cor(bnd_t, (V,E,T), M_0)
9       return bnd_t(M_0)
10  end function
11
12  procedure dfsFiringBound(in (V,E,T), in t, in M, inout bnd_t,
13                           inout visited, in path)
14      assert ∀M' ∈ visited : cor(bnd_t, (V,E,T), M')
15             ∧(path = (M_1,t_1,M_2)···(M_i,t_i,M_{i+1})(M_{i+1},t_{i+1},M_{i+2})···(M_n,t_n,M)
16               ∨path = ε)
17
18      if M ∉ visited then
19          bnd_t(M) := 0
20          for each (M,t',M') ∈ E do // Nachfolgeknoten betrachten
21              // ist M' auf dem bereits gegangenen Pfad?
22              looped := ∅
23              step := path(M,t',M')
24              do
25                  let step = γ(M̃, t̃, M̃')
26                  step := γ
27                  looped := looped ∪ {t̃}
28              while step ≠ ε ∧ M' ≠ M̃
29              if M' ≠ M̃ then // kein Kreis
30                  looped := ∅
31              end if
32              // looped enthält die Transitionen des Kreises oder
33              // ist leer, wenn noch kein Kreis gelaufen wurde
34
35              if looped ≠ ∅ then // ein Kreis, keine Rekursion
36                  if t ∈ looped then
37                      bnd_t(M) := ω // t kann beliebig oft feuern
38                      break // beende "for each (M,t',M') ..."
```

```
39                  end if
40              else // kein Kreis, bnd_t(M') berechnen
41                  dfsFiringBound((V,E,T), t, M', bnd_t, visited,
42                                 path(M,t',M'))
43                  if t' = t ∧ bnd_t(M') + 1 > bnd_t(M) then
44                      // t kann mindestens bnd_t(M') + 1-mal feuern
45                      bnd_t(M) := bnd_t(M') + 1
46                  else if bnd_t(M') > bnd_t(M)
47                      // t kann mindestens bnd_t(M')-mal feuern
48                      bnd_t(M) := bnd_t(M')
49                  end if
50                  if bnd_t(M) = ω then
51                      break // beende "for each (M,t',M') ..."
52                  end if
53              end if
54          end for
55          visited := visited ∪ {M}
56      end if
57
58      ensure M ∈ visited ∧ cor(bnd_t, (V,E,T), M)
59  end procedure
```

7.2.2 Korrektheit

Um die Korrektheit der `firingBound`-Funktion einzusehen, genügt es, die Richtigkeit der `dfsFiringBound`-Prozedur nachzuweisen. Aus der Nachbedingung von `dfsFiringBound` folgt bereits die Nachbedingung von `firingBound`. Beim Aufruf von `dfsFiringBound` ist die Vorbedingung offensichtlich erfüllt, da $visited = \varnothing \wedge path = \epsilon$.

Wenn $M \in visited$, so folgt die Nachbedingung $M \in visited \wedge cor(bnd_t, (V, E, T), M)$ bereits aus der Vorbedingung. Ist $M \notin visited$ und die Vorbedingung erfüllt, so werden alle von M ausgehende Kanten betrachtet. Für jede Kante (M, t', M') sind zwei Fälle zu unterscheiden:

Es wird ein Kreis gefunden M' liegt auf dem durch *path* beschriebenen Pfad. Dies wird

durch die Schleife ab Zeile 24 festgestellt, indem der durch *path* beschriebene Kantenzug rückwärts durchlaufen wird und dabei die zugehörigen Transitionen in der *looped*-Menge gesammelt werden. Wird keine Kante gefunden, die von M' ausging, so wird $looped = \varnothing$ gesetzt und damit angezeigt, dass (noch) kein Kreis vorliegt. Ist $t \in looped$, so kann t auf dem gefundenen Kreis beliebig oft feuern und es wird $bnd_t(M) = \omega$ gesetzt.

Offensichtlich gilt mit dem betrachteten Pfad bereits der Teil der Nachbedingung und die Schleife darf verlassen werden:

$$\begin{aligned} & bnd_t(M) = \omega \wedge \exists \sigma \in looped^* \subseteq T^*, p(\sigma)(t) \geq 1 : (\forall i \in \mathbb{N} : M \overset{\sigma^{i+1}}{\rightarrow} M) \\ \Rightarrow\ & bnd_t(M) = \omega \wedge \forall i \in \mathbb{N} \exists n \in \mathbb{N} \exists \sigma \in T^* \exists M' \in V : (M \overset{\sigma}{\rightarrow}_E M' \wedge p(\sigma)(t) > i) \\ \Rightarrow\ & cor(bnd_t, (V, E, T), M). \end{aligned}$$

Es erfolgt kein weiterer rekursiver Aufruf.

Bisher wurde kein Kreis gefunden Für den Fall, dass die *looped*-Menge leer ist, der aktuelle Pfad also keinen Kreis beschreibt, wird mit dem rekursiven Aufruf der Prozedur für den Nachfolgeknoten M' und den um (M, t', M') verlängert erweiterten Pfad *path* die Korrektheit von $bnd_t(M')$ im Sinne von $cor(bnd_t, (V, E, T), M)$ versichert. Bei diesem rekursiven Aufruf wird die Vorbedingung erfüllt, da die *visited*-Menge nicht verändert wurde und der übergebene Pfad entweder (M, t', M'), wenn $path = \epsilon$, oder $\gamma(M'', t'', M)(M, t', M')$, wenn $path = \gamma(M'', t'', M)$, ist. Nach dem Aufruf gilt also $M' \in visitedt \wedge cor(bnd_t, (V, E, T), M')$. Auf den von Knoten M' ausgehenden Pfaden, kommt t höchstens $bnd_t(M')$-mal vor, beziehungsweise beliebig oft, wenn $bnd_t(M') = \omega$ ist.

Da es eine Kante von M nach M' gibt, kommt t auf einem von M ausgehenden Pfad mindestens $bnd_t(M')$-mal vor. Ist $t' = t$, so kommt t sogar mindestens $bnd_t(M')+1$-mal vor. Wenn diese untere Schranke den bisher in $bnd_t(M)$ gespeicherten Wert übersteigt, so wird $bnd_t(M)$ durch die neue untere Schranke ersetzt. Wird dabei der Wert ω erreicht, so kann die Schleife verlassen werden, da keine weitere Erhöhung der unteren Schranke mehr möglich ist.

Mit der Schleife wird der korrekte Wert für $bnd_t(M)$ von 0 ausgehend maximiert, da alle von M ausgehenden Kanten betrachtet werden, solange eine weitere Erhöhung möglich ist. Abschließend wird M zu *visited* hinzugefügt und damit die Nachbedingung erfüllt.

Damit berechnet die Funktion `firingBound(`(V, E, T)`, `M_0`, `t`)` den gewünschten Wert.

7.2.3 Aufwandsabschätzung

Für jeden erreichbaren Knoten in V wird bnd_t höchstens einmal bestimmt. Bei der Berechnung wird für jede Ausgangskante des Knotens berechnet, ob ein Kreis vorliegt.

Es sei bemerkt, dass die innere Schleife ab Zeile 24 durch ein entsprechendes Hashverfahren oder ein übliches Markieren besuchter Knoten (vgl. [Sed04, Prog. 18.2, S.94ff]) ersetzt werden kann, mit dem festgestellt wird, ob M' auf dem übergebenen Pfad liegt. Ob ein Kreis vorliegt kann dann in $\mathcal{O}(1)$ berechnet werden. Zur Bestimmung der Transitionen des Kreises ist die Schleife allerdings wieder nötig, wobei höchstens $|V|$-viele Durchläufe durchgeführt werden können, bis $step = \epsilon \wedge M' = \widetilde{M}$ erfüllt wird, also der Pfad bis zum Anfang gelaufen, oder ein Kreis erkannt wurde.

Im schlechtesten Fall liegt der Algorithmus damit in der Komplexitätsklasse $\mathcal{O}(V + EV) \subseteq \mathcal{O}(EV)$: Jeder Knoten wird sondiert und für jede Kante wird geprüft, ob ein Kreis vorliegt.

7.2.4 Bemerkungen zur Implementierung

Da die `firingBound`-Funktion meist für mehrere Transitionen desselben Überdeckungsgraphen ausgeführt wird, kann statt einer einzelnen Transition t eine Transitionenmenge Tr übergeben werden. Die Funktion bnd_t wird zu $bnd : Tr \times V \to \mathbb{N}_\omega$ verändert und alle Vorkommen von $cor(bnd_t, (V, E, T), M)$ werden durch die quantifizierte Bedingung $\forall t \in Tr : cor(bnd(t), (V, E, T), M)$ ersetzt. In der rekursiven Prozedur `dfsFiringBound` werden die Zugriffe auf bnd_t sinnvoll korrigiert. Die Schleifenabbrüche mittels `break` werden entfernt – als weitere Möglichkeit werden statt des `break`s die entsprechende Transition aus der Tr-Menge entfernt, da sie bereits beliebig oft feuern kann. Wird die Tr-Menge auf diese Weise geleert, so ist es möglich, dass der Algorithmus frühzeitig terminiert.

Da sich die Aufwandsabschätzung nichts verändert, kann also für beliebige Mengen $Tr \subseteq T$ die maximale Anzahl der Feuerschritte der Transitionen aus Tr in $\mathcal{O}(EV)$ errechnet werden.

7.3 Deadlocks

Wird ein Überdeckungsgraph zu dem erzeugten Petrinetz aufgebaut, so kann er mit einem einfachen Algorithmus nach Deadlock-Zuständen des Petrinetzes durchsucht werden.

7.3.1 Eine hinreichende Bedingung für Deadlocks

Besitzt ein verallgemeinerter Zustand M im Überdeckungsgraphen zu $N = (S, T, F, M_0)$, $Cov(N) = (V, E, T, M_0)$, keinen Nachfolger, so stellt er einen, wegen möglicher ω-Einträge, verallgemeinerten Deadlock dar.

$$(\exists M \in V : E \cap \{(M, t, M') \mid t \in T, M' \in V\} = \varnothing) \Rightarrow M \text{ verallgem. Deadlock von } N$$

Ist M ein verallgemeinerter Deadlock, so gilt mit den Eigenschaften von $Cov(N)$:

$$\exists M' \in [M_0\rangle : M' \leq_\omega M \wedge M' \text{ Deadlock von } N$$

Einem verallgemeinerten Deadlock entsprechen also echte Deadlocks von N. Enthält ein verallgemeinerter Deadlock M eine ω-Komponente, so sind es sogar unendlich viele echte Deadlocks, da es unendlich viele solcher M' mit $M' \leq_\omega M$ gibt.

7.3.2 Eine hinreichende Bedingung für Deadlockfreiheit

Die Umkehrung der Bedingung für Deadlocks gilt nicht: daraus, dass alle Knoten im Überdeckungsgraphen Nachfolger besitzen, folgt nicht, dass das Netz deadlockfrei ist. Betrachtet man allerdings nur Ausgangskanten mit Transitionen, die Stellen im Vorbereich haben, die im Zustand kein ω tragen, so kann man dies folgern.

$$(\forall M \in V \exists (M, t, M') \in E \forall s \in {}^{\bullet}t : M(s) \in \mathbb{N}) \Rightarrow N \text{ deadlockfrei.}$$

Ein Netz ist also deadlockfrei, wenn jeder Knoten M eine ausgehende Kante (M, t, M') besitzt, bei der ${}^{\bullet}t$ eine Teilmenge der in M spezifizierten Stellen ist. Zum Nachweis dieser Eigenschaft sei $M \in V$ und sei $(M, t, M') \in E$ ohne Einschränkungen die einzige Ausgangskante von M, mit $\forall s \in {}^{\bullet}t : M(s) \in \mathbb{N}$. Offensichtlich gilt dann

$$\forall M'' \in [M_0\rangle : (M'' \leq_\omega M \Rightarrow M'' \ [t\rangle),$$

da der Vorbereich von t in M spezifiziert ist und alle M'' auf ${}^\bullet t$ dieselben Werte wie M tragen. Dass es ein solches M'' gibt, gilt wegen der Eigenschaften des Überdeckungsgraphen. Wird diese Eigenschaft für alle Knoten des Überdeckungsgraphen erfüllt, so ist das Netz deadlockfrei.

7.3.3 „Don't know“

Betrachtet man beide hinreichenden Bedingungen zusammen, so fällt auf, dass keine Aussage gemacht werden kann, wenn alle Knoten Nachfolger haben, aber nicht jeder Knoten eine Kante wie in der zweiten Formel gefordert besitzt.

$$
\begin{array}{ll}
 & (\forall M \in V \exists M' \in V : M \to_E M') \\
\wedge & (\exists M \in V \forall (M, t, M') \in E \exists s \in {}^\bullet t : M(s) = \omega) \\
\Rightarrow & \text{don't know.}
\end{array}
$$

Werden wie in Unterabschnitt 7.1.6 überdeckte Zustände markiert, um die Größe des Graphen geringer zu halten, so kann höchstens öfter auf „don't know“ entschieden werden, jedoch nie ein falscher Deadlock gefunden oder falsche Deadlockfreiheit festgestellt werden.

Durch das Markieren überdeckter Zustände wird lediglich die Eigenschaft verletzt, dass jede Feuersequenz des Netzes einer Kantenfolge im Überdeckungsgraphen entspricht. Es wurde festgestellt, dass es zu jeder Feuersequenz des Netzes eine verlängerte Kantenfolge im Überdeckungsgraphen gibt, welche die gewünschte Eigenschaft für erreichten Zustand und Knoten besitzt. Der über die verlängerte Kantenfolge erreichte Knoten ist weniger genau spezifiziert als der erreichte Zustand im Netz. Unverändert bleibt die Eigenschaft, dass jeder Kantenfolge im Überdeckungsgraphen eine Feuersequenz im Netz entspricht, bei welcher der im Netz erreichte Zustand spezifizierter ist als der im Graphen erreichte.

Mit diesen Beobachtungen folgt die Behauptung, dass durch das Ignorieren markierter Zustände lediglich mehr Fälle von „don't know“ auftreten können, die gefundenen Deadlocks, bzw. die Deadlockfreiheit, aber korrekt sind.

Die folgenden zwei Beispiele in Abbildung 7.1 sollen verdeutlichen, wie es zu einem „don't know“ kommt. Sie besitzen beide den gleichen Überdeckungsgraph und N besitzt einen Deadlock, während N' deadlockfrei ist. Jeder Knoten im Überdeckungsgraphen hat zwar

einen Nachfolger, aber es gibt Knoten $(\omega, \omega)^\mathsf{T}$, bei jede Transition auf einer Ausgangskante keinen vollständig spezifizierten Vorbereich hat. Da die Vorbereiche nicht vollständig spezifiziert sind, kann nicht erkannt werden, ob unter den erreichbaren Zuständen des Netzes, die von $(\omega, \omega)^\mathsf{T}$ überdeckt wird, ein Deadlock ist oder nicht.

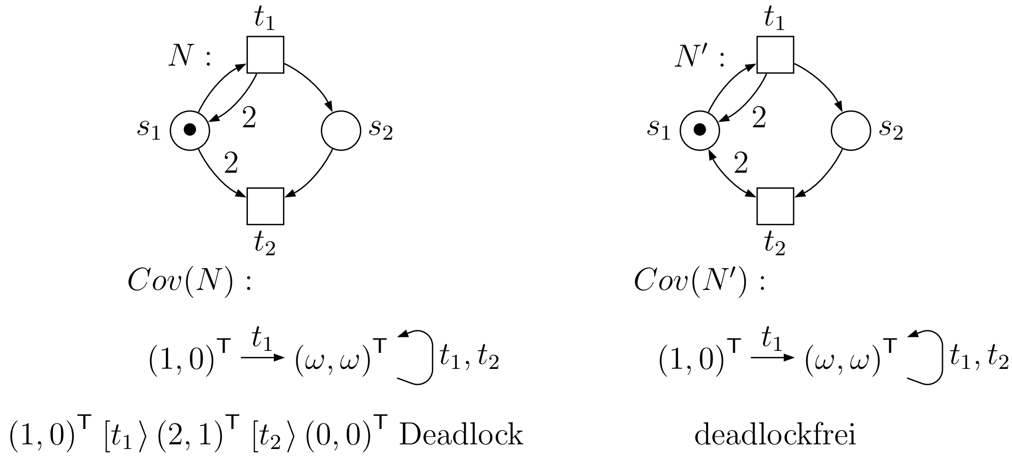

Abbildung 7.1: Fälle von „don't know“

7.4 Beschränktheit

Für alle Stellen, und damit für das gesamte Netz, kann die Beschränktheit am Überdeckungsgraphen abgelesen werden. Gibt es einen Knoten $M \in V$ mit $M(s) = \omega$, so ist die Stelle unbeschränkt, ansonsten ist das Maximum der $M(s)$ für alle $M \in V$ die (kleinste) obere Schranke für die Markenanzahl von Stelle s. Offensichtlich wird diese Eigenschaft nicht vom Ignorieren markierter Zustände (Unterabschnitt 7.1.6) verletzt.

8 Auffaltung zur Approximation der sequentiellen Semantik

Um die in Unterabschnitt 3.1.2 beschrieben Semantik zu approximieren, wird eine Auffaltung stark gefalteter Netze entwickelt. Idee dabei ist es Transitionen, mit denen eine Restriktion zu einem freien Namen aufgelöst wird, zu vervielfältigen. In der starken Faltung kann eine solche Transition höchstens k-mal feuern ($k \in \mathbb{N}_\omega$). Ist das $k \in \mathbb{N}$, so bleibt das Netz mit der k-fachen Auffaltung der Transition endlich. Dabei werden die von der Transition erzeugten, freien Namen entlang Ausgangskanten verfolgt, bis eine Stelle erreicht wird, die den freien Namen nicht kennt. Zu dem so bestimmte Netzteil werden $k-1$ Kopien angelegt, die dort Kanten in das ursprüngliche Netz haben, wo der freie Name „vergessen“ wird. Dieses intuitive Vorgehen funktioniert wegen der Eigenschaft der strukturellen Kongruenz, freie Namen zu erhalten: Das Umbenennen freier Namen ist nicht erlaubt. Zu beachten ist, dass, wenn bei der Verfolgung von Namen $\tilde{a}$ eine Transition vervielfacht wird, die Namen $\tilde{b}$ erzeugt, von dort ab beide, $\tilde{a}$ und $\tilde{b}$, verfolgt werden und entsprechende Netzteile vervielfältigt werden. Da die Transition, die $\tilde{b}$ erzeugt vervielfältigt wird, werden auch eindeutige Kopien der erzeugten Namen $\tilde{b}$ angelegt. Würde nun lediglich $\tilde{a}$ weiter verfolgt werden, so käme es zu Inkonsistenzen im Netz.

Im vorgestellten Algorithmus werden die Kopien der von einer bestimmten Transition ausgehenden Teilnetze simultan erzeugt. Kann eine Transition k-mal feuern, so wird der Netzteil erhalten und $k-1$ Kopien angelegt. Der unveränderte Netzteil, von dem die Kopien erzeugt werden, wird im Folgenden „Original“ genannt. Wird im Zuge des Algorithmus eine Transition vervielfältigt, die ihrerseits Namen erzeugt, so wird die zu vervielfältigende Transition wird entsprechend häufig kopiert und die übergebenen Abbildungen $firingBounds$ und T_ν aktualisiert. Ebenso wird die *todo*-Menge des Algorithmus aus Listing 3.2 um die Kopien erweitert.

Zu Beginn des Programms werden Speicher für die Kopien des Teilnetzes angelegt, die von dem jeweiligen Originalelement auf die Kopien abbilden. Wichtig ist, dass die Kopien eindeutig nummeriert sind, um sie voneinander getrennt zu halten. Ebenso wird die Starttransition vervielfältigt und zu jeder Kopie und jedem erzeugten Namen ein eindeutiger, neuer Name erzeugt. Eine Abbildung ordnet dem Originalnamen und der Nummer der Kopie den neuen, eindeutigen Namen zu, der in der Kopie an Stelle des Originalnamens benutzt wird.

Der Algorithmus verwendet im weiteren Verlauf eine Randmenge, die Originaltransitionen enthält, welche bereits vervielfältigt wurden und noch zu bearbeiten sind. Die Randmenge enthält initial nur die Starttransition.

In der Hauptschleife wird jeweils eine Originaltransition aus dem Rand genommen. Erzeugt sie Namen, so werden neue und eindeutige Kopien der Namen angelegt und die erzeugten Namen in die Menge der zu verfolgenden Namen aufgenommen. Zudem werden die Kopien der Transition für den Algorithmus in Listing 3.1 markiert, da die Kopien noch aufgefaltet werden müssen.

Von der aktuellen Transition aus wird der Fluss der verfolgten Namen entlang der Ausgangskanten nachempfunden und entsprechende Stellen vervielfacht. Bei solchen Stellen, die einen der verfolgten freien Namen benutzen, werden wiederum die Ausgangskanten betrachtet und die Transitionen im Nachbereich vervielfältigt. Die Originaltransition im Nachbereich der Originalstelle wird dann in den Rand aufgenommen. Wird eine Kante von oder zu einer Stelle gefunden, die keinen der verfolgten freien Namen benutzt, so wird sie in die Menge der Kanten, die potentiell ins Originalnetz führen aufgenommen. Diese Kantenmenge wird nach Terminierung der Hauptschleife behandelt, indem jeweils geprüft wird, ob die Kanten tatsächlich ins Originalnetz führen. Entsprechend werden Kanten erzeugt.

Der folgende Code stellt das Vorgehen genauer dar.

Listing 8.1: Auffaltung

```
1  function unfold(in (S,T,F,M_0), in t, inout T_ν, inout firingBounds,
2                  inout todo)
3      (S',T',F',M_0) := (S,T,F,M_0)
4      count := firingBounds(t) − 1
5      // Neue, eindeutige Namen für jede Kopie
```

```
6     let σ : 𝒩 × ℕ → 𝒩
7     // Kopien der Stellen
8     let pl : S × ℕ → S′
9     // Kopien der Transitionen
10    let tr : T × ℕ → T′
11
12    // Starttransition t vervielfältigen
13    for i = 1 ... count do
14        // Für jede Kopie neue Namen erzeugen
15        for each a ∈ T_ν(t) do
16            σ(a, i) := neuen, im Netz eindeutigen Namen wählen
17        end for
18        // Kopien von t erzeugen
19        tr(t, i) := neue Transition in T′ erzeugen
20        // Vermerken, dass die Kopien Namen erzeugen
21        T_ν(tr(t, i)) := supp(σ(a))
22    end for
23
24    // Die arcs Menge wird gewichtete Kanten enthalten,
25    // bei denen nicht sicher ist, ob sie in das Original−
26    // netz zurückführen, oder ob die Stellen doch ver−
27    // vielfältigt werden.
28    arcs := ∅ // ⊆ S′ × ℕ × T′ ∪ T′ × ℕ × S′
29
30    // Erzeuge initialen Rand
31    Rand := {t} // ⊆ T′
32    // Solange der Rand nicht leer ist
33    while Rand ≠ ∅ do
34        wähle t′ ∈ Rand
35        Rand := Rand \ {t′}
36
37        // Wenn t′ eine Transition ist, die Namen erzeugt...
38        if t′ ≠ t ∧ t′ ∈ supp(T_ν) then
39            // Neue, eindeutige Namen für Kopien von t′ erzeugen
```

```
40              let σ′ : 𝒩 × ℕ → 𝒩
41              for i = 1 . . . count do
42                  for each a ∈ T_ν(t′) do
43                      σ′(a, i) := neuen, im Netz eindeutigen Namen wählen
44                      // Die von t′ erzeugten Namen müssen
45                      // ebenfalls verfolgt werden.
46                      σ(a, i) := σ′(a, i)
47                  end for
48                  T_ν(tr(t′, i)) := supp(σ′(i))
49                  firingBounds(tr(t′, i)) := firingBounds(t′)
50                  // Die Kopie als „zu bearbeiten“ markieren
51                  // (für Listing 3.1)
52                  todo := todo ∪ supp(tr(t′))
53              end for
54          end if
55
56          // Der Vorbereich von t′ ist potentiell im Originalnetz
57          for p ∈ •t′ do
58              if p ∉ supp(pl(1)) then
59                  arcs := arcs ∪ {(p, F′(p, t′), t′)}
60              end if
61          end for
62
63          // Den Nachbereich von t′ betrachten
64          for p ∈ t′• do
65              if p ∈ supp(pl(1)) then
66                  // p ist potentiell im Originalnetz
67                  for i = 1 . . . count do
68                      F′(tr(t′, i), pl(p, i)) := F′(t′, p)
69                  end for
70
71              else if fn(p) ∩ supp(σ(1)) ≠ ∅ then
72                  // Ein erzeugter Name wurde propagiert
73                  // Die Stelle wird vervielfältigt
```

```
74          for i = 1...count do
75              p' := p // neue Stelle in S' erzeugen
76              // Die Namen der Kopie einsetzen
77              for each a ∈ supp(σ(i)) do
78                  p' := {σ(a,i)/a}p'
79              end for
80              pl(p,i) := p'
81              S' := S' ∪ {p'}
82              F'(p',tr(t',i)) := F'(p,t')
83          end for
84
85          // Von dieser Stelle kann der Name weiter
86          // propagiert werden. Der Nachbereich wird
87          // überprüft.
88          for each t'' ∈ p• do
89              if t'' ∈ supp(tr(1)) then
90                  // Diese Transition wurde bereits
91                  // vervielfältigt. Kanten einfügen.
92                  for i = 1...count do
93                      F'(pl(p,i),tr(t'',i)) := F'(p,t'')
94                  end for
95              else
96                  // Diese Transition wird verviel-
97                  // fältigt und dem Rand hinzugefügt.
98                  // Sie wurde vorher noch nicht
99                  // vervielfältigt.
100                 Rand := Rand ∪ {t''}
101                 for i = 1...count do
102                     tr(t'',i) := neue Transition in T' erzeugen
103                     // Eingangskanten kopieren
104                     F'(pl(p,i),tr(t'',i)) := F'(p,t'')
105                 end for
106             end if
107         end for
```

```
108
109             else
110                 // p wurde nicht vervielfältigt und
111                 // verwendet nicht die erzeugten Namen.
112                 // Damit ist sie potientiell im Originalnetz.
113                 arcs := arcs ∪ {(t, F'(t', p), p)}
114             end if
115         end for
116     end while
117
118     // Potentielle Kanten in das Originalnetz betrachten.
119     for each (p, w, t') ∈ arcs do
120         if p ∉ supp(pl(1)) then
121             // p wurde nicht vervielfältigt
122             for i = 1...count do
123                 F'(p, tr(t', i)) := w
124             end for
125         else
126             // p wurde vervielfältigt
127             for i = 1...count do
128                 F'(pl(p, i), tr(t', i)) := w
129             end for
130         end if
131     end for
132
133     for each (t, w, p) ∈ arcs do
134         if p ∉ supp(pl(1)) then
135             // p wurde nicht vervielfältigt
136             for i = 1...count do
137                 F'(tr(t', i), p) := w
138             end for
139         else
140             // p wurde vervielfältigt
141             for i = 1...count do
```

```
142                         F'(tr(t', i), pl(p, i)) := w
143                   end for
144              end if
145         end for
146
147         return (S', T', F', M_0)
148    end function
```

8.1 Aufwandsabschätzung

Da jede Transition des Originalnetzes höchstens einmal im Rand aufgenommen werden kann und keine geschachtelten Vervielfältigungen, sondern nur eine vorgenommen wird, läuft der Algorithmus in zur Netzgröße linearer Zeit. Allerhöchstens könnte das gesamte Netz vervielfacht werden.

9 Implementierung

Unter Verwendung generischer Typen wurden die Algorithmen in Java 5 umgesetzt. Die Implementierung weicht in sofern von den Algorithmen ab, dass viele in Java-Bibliotheken vorhandenen Datenstrukturen, wie zum Beispiel `IdentityHashMap`s und `Queue`s verwendet werden. Besonderer Wert wurde auf eine zeit- und speichereffiziente Ausführung gelegt, wobei die Größe des Überdeckungsgraphen für den Speicherverbrauch des Programms meist maßgebend ist.

Auch auf Benutzerfreundlichkeit, mit aussagekräftigen Fehlermeldungen und Informationen, und flexible, aber bequeme Einstellmöglichkeiten wurde geachtet. Ein einfacher Layout-Algorithmus breitet die entstehenden Netze bereits aus, um ein gewisses Maß an Übersichtlichkeit als Ausgangspunkt für das manuelle Layout zu geben.

9.1 Paket- und Klassenbeschreibung

Das *Petruchio*-Programm ist in mehrere Pakete aufgeteilt:

petruchio.compiler Das Compiler-Paket enthält die in dieser Arbeit besprochenen Algorithmen.

petruchio.pi Die Datenstrukturen für π-Kalkül-Prozesse sind in diesem Paket zusammengefasst.

petruchio.pn Ein sehr einfaches Datenmodell für Petrinetze und ihre Ausgabe in das PEP-Format ist in diesem Paket implementiert.

petruchio.piparser Die Grammatik und der daraus generierte Parser für π-Kalkül-Prozesse sind in diesem Paket enthalten.

Alle Klassen implementieren die jeweiligen Interfaces, die im `petruchio.interfaces`-Paket zur Verfügung stehen. In Abbildung 9.1 und in Abbildung 9.2 wird eine Übersicht

der Schnittstellen, bzw. der Klassen dargestellt.

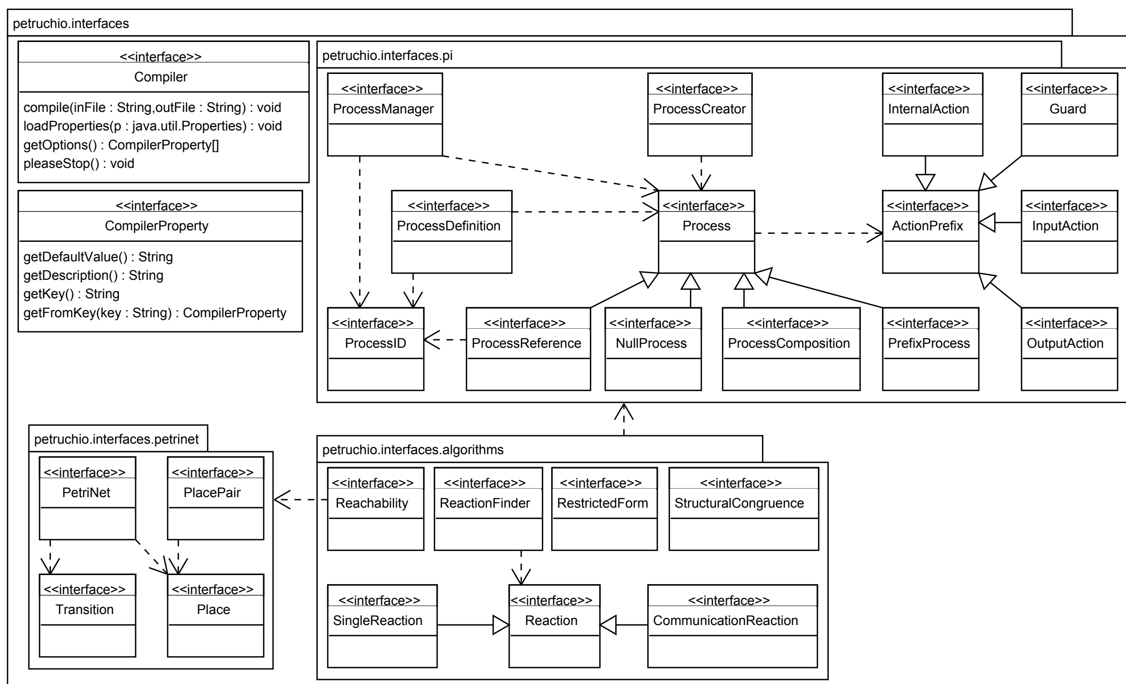

Abbildung 9.1: Schnittstellen

9.2 Erweiterbarkeit

Da die Klassen im Compiler, sogar der Compiler selbst, ausschließlich ihre Schnittstellen angesprochen werden und nicht statisch im Programmcode stehen, lassen sie sich leicht durch andere Implementierungen auswechseln. Die in Abschnitt 9.4 unter Implementierung aufgeführten Einstellungen erlauben die Auswahl zu treffen, welche Klassen tatsächlich geladen werden. Wichtigste Ansatzpunkte sind dabei die zu ladenden Klassen

- Compiler, der das `Compiler`-Interface implementieren muss,
- Prozess-Manager zur Verwaltung der Prozesse im Speicher, der das `ProcessManager`-Interface implementieren muss,
- Prozess-Erzeuger, der das Interface `ProcessCreator` implementieren muss und maßgeblich dafür ist, welche Prozess-Implementierung benutzt wird,

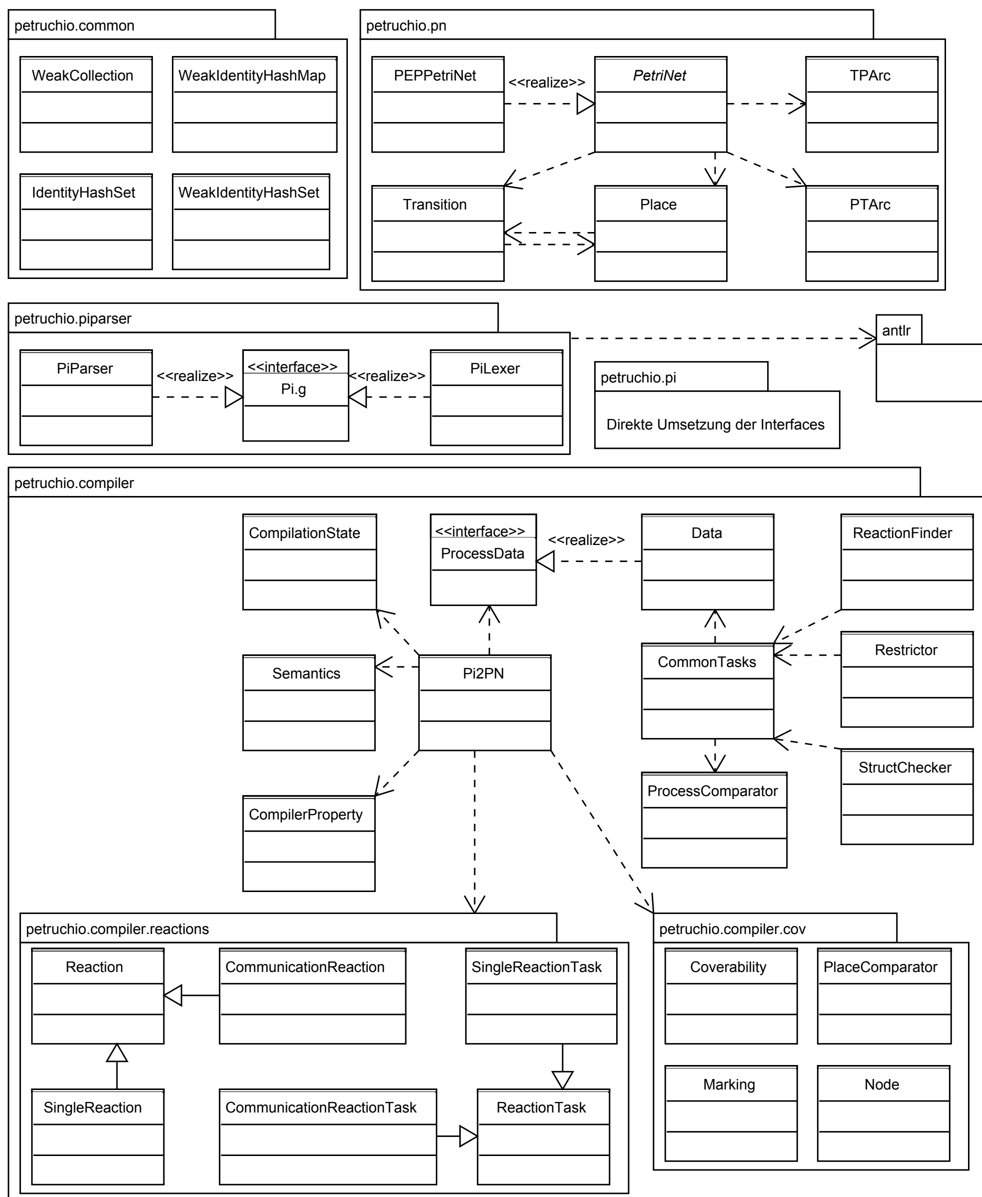

Abbildung 9.2: Klassen

- Petrinetz, welches das Ausgabeformat bestimmt und das `PetriNet`-Interface implementieren muss,
- Algorithmen zum Entscheiden struktureller Kongruenz, Herstellung der strengen Restriktionsform, Finden von Reaktionen und inkrementeller Aufbau des Überdeckungsgraphen, die ihre jeweiligen Interfaces `StructuralCongruence`, `RestrictedForm`, `ReactionFinder` und `Reachability` zu implementieren haben.

9.3 π-Grammatik

Der Compiler akzeptiert Textdateien als Eingabe, die eine beliebige Anzahl von Prozessdefinitionen enthalten. Dabei soll höchstens ein Prozess mit dem `main`-Schlüsselwort als Systemprozess deklariert werden, oder aber ein Prozess den für Systemprozesse gewählten Namen tragen. Siehe dazu auch Abschnitt 9.4.

Zusätzlich zur hier angegebenen Grammatik in EBNF werden Kommentare im Java-Stil unterstützt. Mit `//` werden Zeilenkommentare eingeleitet und Blockkommentare werden in `/*` und `*/` eingefasst. Leer- und Tabulatorzeichen, sowie Zeilenumbrüche werden ignoriert. Es muss genau eine Prozessdefinition mit dem `main`-Schlüsselwort als Systemprozess deklariert werden.

Listing 9.1: Verwendete π-Grammatik

```
PiFile = { ProcessDefinition } ;

ProcessDefinition =
    [ 'main' ] ProcessID Parameters ':=' Process ';' ;

Parameters = [ '(' Names ')' ] ;

ProcessID = Name ;

Name = Character { Character } | '"' (Character | ' ')* '"';

Names = [ Name { ',' Name } ] ;

Process =
    { Restrictions } { Guard '.' }
    ( '0'
    | ProcessReference
    | ProcessComposition
    | PrefixProcess
    ) ;

```

```
22 Restrictions =
23       [ 'new' | 'nu' ] Name { ( '.' | ',' ) Name } ( '.' | ',' ) ;
24
25 Guard = '[' Name ( '=' | '==' ) Name ']' ;
26
27 ProcessReference = ProcessID Parameters ;
28
29 ProcessComposition =
30         '(' Process { '|' Process } ')'
31       | '(' PrefixProcess { '+' PrefixProcess } ')'
32       | '(' Process { ';' Process } ')' ;
33
34 PrefixProcess = Prefix '.' { Prefix '.' } Process ;
35
36 Prefix =
37         'in'  Name [ '(' Names ')' ]
38       | 'out' Name [ '(' Names ')' | '<' Names '>' ]
39       |       Name ( '(' Names ')' | '<' Names '>' )
40       | 'tau' ;
41
42 Character =
43         'a' | ... | 'z' | 'A' | ... | 'Z' | '0' | ... | '9'
44       | '_' | '-' | '~' | '*' | '#' | ''' | '!' | '?' | '&'
45       | '^' | '$' | '@' | '%' ;
```

Diese Grammatik ist offensichtlich mehrdeutig, da sich zum Beispiel die Prozessdefinition `A := (0);` mittels einer parallelen, einer sequentiellen, wie auch einer Choice-Komposition herleiten lässt.

Die tatsächlich verwendete, komplexere ANTLR-Grammatik beschreibt die gleiche Menge an Prozessen, ist aber eindeutig und enthält Programmfragmente, die während des Parsens einer Datei entsprechende Syntaxbäume für die Prozesse im gewählten Datenmodell erzeugen (siehe dazu auch [ANT]).

9.4 Compilereinstellungen

Zur Steuerung des Programms ist eine Konfigurationsdatei vorgesehen, welche die Standardwerte für die einzustellenden Eigenschaften festlegt. Diese Standardwerte können dann für eine Ausführung des Programms zeitweise überschrieben werden (siehe Abschnitt 9.5).

Folgende Eigenschaften können in der Konfigurationsdatei über `Eigenschaft=Wert`-Einträge in einzelnen Zeilen definiert werden.

Semantik

- `Semantics`
  Eine kommaseparierte Aufzählung der zur Übersetzung zu verwendenden Semantiken. Beispielsweise `STRUCTURAL,SEQUENTIAL,STR_SEQ,SEQ_STR`.
- `CheckForDeadlocks`
  Ein boolescher Schalter (`true` oder `false`), der bestimmt, ob die in Abschnitt 7.3 beschriebene Heuristik angewendet werden soll, um das entstehende Netz auf Deadlocks zu untersuchen. Diese Eigenschaft setzt `ProcessReachableCommunicationsOnly=true` voraus.
- `CheckForInfiniteBreadth`
  Ein boolescher Schalter (`true` oder `false`), der bestimmt, ob die eine Heuristik angewendet werden soll, um Fragmente, die unbeschränkt in der Breite wachsen können zu identifizieren.
- `ComputeBounds`
  Ein boolescher Schalter (`true` oder `false`), der bestimmt, ob die Beschränktheit der Stellen und des Netzes in der Netzdatei vermerkt werden soll. Diese Eigenschaft setzt `ProcessReachableCommunicationsOnly=true` voraus.
- `KeepRestrictionsOnPublicNames`
  Ein boolescher Schalter (`true` oder `false`), der bestimmt, ob die Restriktionen des Systemprozesses, die allen Unterprozessen bekannt sind in freie Namen umgewandelt werden sollen. Beispielsweise würde `SYSTEM := new a, b.(X(a) | new c.(Y(b);D));` als `SYSTEM := (X(a) | new c.(Y(b);D));` aufgefasst werden. Erfahrungsgemäß werden die Netze kleiner, wenn diese Eigenschaft auf `true` gesetzt wird.

- `MaximalReactionCount`
  Legt fest, wieviele Reaktionsschritte, bzw. erzeugte Transitionen, höchstens bearbeitet werden. Eine nicht-positive Zahl steht dabei für „beliebig viele“. Hiermit lässt sich auch bei Prozessen die nicht strukturell stationär sind ein Halten des Programms erzwingen.
- `ProcessReachableCommunicationsOnly`
  Ein boolescher Schalter (`true` oder `false`), der bestimmt, ob Transitionen, die für Kommunikationen von Prozessen stehen, ausschließlich dann erzeugt werden, wenn es einen vom Startzustand aus erreichbaren Zustand gibt, der die entsprechende Transition aktivieren würde. Wird der Schalter auf `false` gesetzt, so wird ein generalisiertes Netz erzeugt, dass dem zu übersetzenden Prozess entspricht, bei dem angenommen wird, dass jedes erreichbare Fragment „in beliebiger Vielzahl“ vorkommt – also jede Stelle im Netz mit jeder kommunizieren kann. Intuitiv ist das Netz dann endlich, wenn auch unter der Annahme der beliebigen Kommunikation nur endlich viele Fragmente erreichbar sind. Da die so „zusätzlich“ (im Vergleich zu Schalter `false`) erzeugten Transitionen tot sind, besitzt ein so erzeugtes Netz, wenn es endlich ist, den gleichen Erreichbarkeitsgraphen wie ein mit Schalter `true` erzeugtes Netz. Ist der Prozess geschlossen, so können Fragmente nicht miteinander kommunizieren und dieser Schalter kann ohne Informationsverlust auf `false` gesetzt werden, um das Testen von Erreichbarkeit einzusparen. Siehe dazu Abschnitt 7.1.

Logging

- `LogHandler`
  Die zum Logging zu verwendende Handler-Klasse. Siehe dazu `http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/Handler.html`. Üblicherweise `java.util.logging.ConsoleHandler`.
- `LogLevel`
  Die Granularität des Loggings. Ein Wert aus { `SEVERE`, `WARNING`, `INFO`, `FINE`, `FINER`, `FINEST`, `ALL` }. Siehe dazu `http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/Level.html`
- `UseParentLoggers`
  Ein boolescher Schalter (`true` oder `false`), der bestimmt, ob die Log-Mit-

teilungen an die sogenannten ParentHandler weitergegeben werden sollen. Siehe dazu auch `http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/Logger.html#setUseParentHandlers(boolean)`

Multithreading

- `UseThreads`
  Ein boolescher Schalter (`true` oder `false`), der bestimmt, ob die Berechnung des Überdeckungsgraphen parallel zur Übersetzung ausgeführt werden soll. Siehe dazu Unterabschnitt 7.1.7.
- `TerminationPatience`
  Legt eine Dauer in Millisekunden fest, die nach Beenden der Übersetzung auf das Halten der Berechnung des Überdeckungsgraphen gewartet wird, bevor diese erzwungen abgebrochen wird.
- `MainPriority`
  Legt die Priorität (`1` bis `10`) des Prozesses zur Übersetzung fest. Siehe auch die Dokumentation im Netz: `http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html#setPriority(int)`
- `ReachabilityPriority`
  Legt die Priorität (`1` bis `10`) des Prozesses zur Berechnung des Überdeckungsgraphen fest. Siehe dazu auch `http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html#setPriority(int)`.

Implementierung

- `Compiler`
  Die zu verwendende Klasse zur Umsetzung der Semantiken. Beispielsweise `petruchio.compiler.Pi2PN`.
- `ProcessCreator`
  Die zu verwendende Klasse zum Erzeugen von Prozessen. Beispielsweise `petruchio.pi.ProcessCreator`.
- `ProcessManager`
  Die zu verwendende Klasse zum Verwalten von Prozessen. Beispielsweise `petruchio.pi.ProcessManager`.
- `RestrictedForm`
  Die zu verwendende Klasse zum Herstellen der Restriktionsform. Beispiels-

weise `petruchio.compiler.Restrictor`.

- `Reachability`
  Die zu verwendende Klasse zum Prüfen der Erreichbarkeit im Sinne von Abschnitt 7.1. Beispielsweise `petruchio.compiler.cov.Coverability`.
- `PetriNet`
  Die zu verwendende Klasse zum Aufbau und zur Ausgabe des erzeugten Petrinetzes. Beispielsweise `petruchio.pn.PEPPetriNet`.
- `PiLexer`
  Die zu verwendende Klasse zum Vorbereiten eingegebener Prozesse für den Parser. Beispielsweise `petruchio.piparser.PiLexer`.
- `PiParser`
  Die zu verwendende Klasse zum Aufbau der Syntaxbäume eingegebener Prozesse. Beispielsweise `petruchio.piparser.PiParser`.

Sonstiges

- `MainProcess`
  Wird in der zu übersetzenden `.pi`-Datei keine Prozessdefinition mit dem `main`-Schlüsselwort zum Systemprozess deklariert, so wird der Prozess mit dem hier angegebenen Namen als Systemprozess gewählt. Beispielsweise `SYSTEM`.
- `PetriNetOffsetX`
  Eine natürliche Zahl, die den gewünschten Abstand zwischen zwei Petrinetz-Elementen in X-Richtung angibt. Beispielsweise `60`.
- `PetriNetOffsetY`
  Eine natürliche Zahl, die den gewünschten Abstand zwischen zwei Petrinetz-Elementen in Y-Richtung angibt. Beispielsweise `60`.

9.5 Programmaufruf

Das Programm kann mittels

```
java -cp antlr-runtime-3.0.jar petruchio.Petruchio [options]
```

aufgerufen werden, wobei `[options]` für eine beliebige Anzahl an Aufrufargumenten steht. Aufrufargumente sind die folgenden:

- `-h`
  Lässt einen kurzen Informationstext zur Verwendung des Programms ausgeben.
- `-c Dateiname`
  Lädt die angegebene Datei als Konfiguration. Der Standardwert ist `cfg/petruchio.cfg`.
- `-p Eigenschaft=Wert`
  Übernimmt den gegebenen Wert für die Konfigurationseigenschaft temporär für den aktuellen Programmlauf.
- `-i Dateiname`
  Gibt die `.pi`-Datei an, die übersetzt werden soll. Diesem Argument muss ein `-o`-Argument folgen.
- `-o Dateiname`
  Gibt die Datei an, in die das übersetzte Petrinetz geschrieben werden soll. Dieses Argument muss einem `-i`-Argument folgen. Es darf auch ein `-`, als Abkürzung für „den gleichen Namen mit `.net`-Endung", an Stelle des Dateinamens übergeben werden.

Es stehen zwei Startskripte zur einfacheren Ausführung zur Verfügung, die in Listing 9.2 und Listing 9.3 aufgeführt sind. Ihnen werden beim Aufruf lediglich die Aufrufargumente übergeben.

Listing 9.2: Windows-Batch-Skript

```
1  @ECHO OFF
2  SET DIR=%~dp0%
3  SET LIB=%DIR%\lib
4  SET CFG=%DIR%\cfg\petruchio.cfg
5  SET ANTLR=%LIB%\antlr-runtime-3.0.jar
6  SET ANTLRCOMPILE=%LIB%\antlr-3.0.jar;%LIB%\antlr-2.7.7.jar;
7                   %LIB%\stringtemplate-3.0.jar
8  SET JUNIT=%LIB%\junit-4.1.jar
9  SET PETRUCHIO=%LIB%\petruchio.jar
10 SET MAINCLASS=petruchio.Petruchio
```

```
11
12 SET ARGUMENTS=
13 :CMDLOOP
14 IF "%1" == "" GOTO END
15    SET ARGUMENTS=%ARGUMENTS% %1
16    SHIFT
17    GOTO CMDLOOP
18 :END
19
20 java -Xmx512 -cp %ANTLR%;%PETRUCHIO% %MAINCLASS%
21       -c %CFG% %ARGUMENTS%
```

Listing 9.3: Unix-Bash-Skript

```
1 #!/bin/bash
2 DIR=$(dirname $0)
3 LIB=$DIR/lib
4 CFG=$DIR/cfg/petruchio.cfg
5 ANTLR=$LIB/antlr-runtime-3.0.jar
6 ANTLRCOMPILE=$LIB/antlr-3.0.jar:$LIB/antlr-2.7.7.jar:
7                $LIB/stringtemplate-3.0.jar
8 JUNIT=$LIB/junit-4.1.jar
9 PETRUCHIO=$LIB/petruchio.jar
10 MAINCLASS=petruchio.Petruchio
11
12 java -Xmx512 -cp $ANTLR:$PETRUCHIO $MAINCLASS -c $CFG $@
```

10 Zusammenfassung und Ausblick

Automatische Verifikation von dynamisch rekonfigurierbaren, nebenläufigen Systemen erleichtert das Prüfen, ob ein System seine Spezifikation erfüllt.

In der vorliegenden Arbeit wurden Algorithmen entwickelt und implementiert, welche die Berechnung von Petrinetz-Semantiken für π-Kalkül-Prozesse automatisieren.

Die in [Mey07b] eingeführte und in Abschnitt 2.3 erklärte Semantik für eine Variante des π-Kalküls wurde kanonisch auf eine polyadische Variante des Kalküls und auf den Sequenzoperator fortgesetzt, der im Rahmen dieser Arbeit elegant definiert wurde. Ihre prozedurale Berechnung wurde in Unterabschnitt 3.1.1 erläutert.

In besonderem Maße wurde auf die Effizienz der entwickelten Algorithmen geachtet, da sich in Unterabschnitt 7.1.1 herausstellte, dass die Berechnung von Meyers strukturellen Semantik für Prozesse, die nicht geschlossen sind, in EXPSPACE liegt. Dazu wurde in Unterabschnitt 7.1.6 eine Vereinfachung des Algorithmus zur Berechnung von Überdeckungsgraphen vorgestellt und herausgefunden, dass diese Berechnung in einen eigenen Thread ausgelagert werden kann.

Für den in Kapitel 6 besprochenen Entscheidungsalgorithmus zur strukturellen Kongruenz wurden notwendige Bedingungen gefunden und bewiesen. Sie wurden in Abschnitt 6.2 zu einer Ordnung auf Prozessen zusammengefasst, die den Programmlauf erheblich abkürzen kann.

Des Weiteren wurde in Unterabschnitt 3.1.2 eine Alternative zu Meyers sequentieller Semantik, die in Abschnitt 2.4 definiert ist, vorgestellt, welche in vielen Fällen einfachere Petrinetze liefert. Die dort eingeführte Idee der starken Faltungen wird in Unterabschnitt 3.1.2.1 und Kapitel 8 genauer betrachtet. Einige Netze der sequentiellen Semantik und der Approximation werden in Unterabschnitt 3.1.2.2 verglichen.

Meyer benutzt verschiedene Restriktionen, um seine strukturelle und seine sequentielle

Semantik zu kombinieren. Der in Abschnitt 2.5 dieser Arbeit vorgestellte Ansatz ist benutzerfreundlicher, da die Erfüllung hinreichender Bedingungen für die Unendlichkeit der Semantiken automatisch erkannt wird. Mit den erfüllten Bedingungen wird automatisch entschieden, welche Restriktionsform zu benutzen ist.

Mit der in Kapitel 9 besprochenen Implementierung in Java wurde ein flexibles, benutzerfreundliches und plattformunabhängiges Werkzeug geschaffen, welches die berechneten Petrinetze im gängigen PEP-Format ausgibt.

Diese Arbeit bietet einige Ansatzpunkte zu weiterführenden Untersuchungen:

- Formale Beweise der entwickelten Algorithmen.
- Genaue Betrachtung des eingeführten Sequenzoperators.
- Untersuchung der stark gefalteten Netze und ihrer Auffaltungen.
- Entwicklung hinreichender Bedingungen für die Unendlichkeit der strukturellen und der sequentiellen Semantik.
- Weitere Vereinfachungen der erzeugten Überdeckungsgraphen.
- Berechnung von Meyers sequentieller Semantik, wie sie definiert ist.
- Erstellen einer integrierten Entwicklungsumgebung für das bequeme Erstellen, Übersetzen und Model-Checken von π-Kalkül-Prozessen.

Anhang

Beispiele

Um ein klareres Bild vom Eingabeformat des Compilers zu geben, folgen vier Beispiele, die im Rahmen der Arbeit als Testfälle benutzt wurden. Die berechneten Petrinetz-Semantiken sind allerdings zu groß, um sie hier anschaulich darzustellen.

Listing A.1: Beispiel: Mobile Phones

```
1  /*
2   * Project:   Petruchio
3   * Filename:  phones.pi
4   * Created:   22.02.2007
5   * Author:    Tim Strazny
6   * Remarks:   This example is taken from Robin Milner:
7   *            Communicating and Mobile Systems: the
8   *            Pi-Calculus, 1999, Cambridge University
9   *            Press, pages 80 and following.
10  */
11
12 main
13 SYSTEM :=
14     new talk_1, switch_1, gain_1, lose_1, talk_2, switch_2,
15         gain_2, lose_2.
16     ( CAR(talk_1, switch_1)
17     | TRANS_1(talk_1, switch_1, gain_1, lose_1)
18     | IDTRANS_2(gain_2, lose_2)
19     | CONTROL_1(talk_1, switch_1, gain_1, lose_1, talk_2,
20                 switch_2, gain_2, lose_2)
21     );
22
```

```
23 TRANS_1(talk, switch, gain, lose) :=
24     ( in talk.TRANS_1(talk, switch, gain, lose)
25     + in lose(t, s).out switch(t, s).IDTRANS_1(gain, lose)
26     );
27
28 IDTRANS_1(gain, lose) :=
29     in gain(t, s).TRANS_1(t, s, gain, lose);
30
31 TRANS_2(talk, switch, gain, lose) :=
32     ( in talk.TRANS_2(talk, switch, gain, lose)
33     + in lose(t, s).out switch(t, s).IDTRANS_2(gain, lose)
34     );
35
36 IDTRANS_2(gain, lose) :=
37     in gain(t, s).TRANS_2(t, s, gain, lose);
38
39
40 CONTROL_1(talk_1, switch_1, gain_1, lose_1, talk_2, switch_2,
41           gain_2, lose_2) :=
42     out lose_1(talk_2, switch_2).out gain_2(talk_2, switch_2).
43           CONTROL_2(talk_1, switch_1, gain_1, lose_1, talk_2,
44                     switch_2, gain_2, lose_2);
45
46 CONTROL_2(talk_1, switch_1, gain_1, lose_1, talk_2, switch_2,
47           gain_2, lose_2) :=
48     out lose_2(talk_1, switch_1).out gain_1(talk_1, switch_1).
49           CONTROL_1(talk_1, switch_1, gain_1, lose_1, talk_2,
50                     switch_2, gain_2, lose_2);
51
52 CAR(talk, switch) :=
53     ( out talk.CAR(talk, switch)
54     + in switch(t, s).CAR(t, s)
55     );
```

Listing A.2: Beispiel: Car Platoon

```
1  /*
2   * Project:   Petruchio
3   * Filename:  platoon.pi
4   * Created:   04.12.2006
5   * Author:    Roland Meyer
6   * Remarks:   This example is taken from: Pi-Calculus
7   *            Dimensions: Structure, 2006, Department
8   *            of Computing Science, University of
9   *            Oldenburg, Oldenburg, Germany.
10  */
11
12 main
13 SYSTEM :=
14     (CREATE | MERGE);
15
16 CREATE :=
17     (new id, ca, rq.FA(id, ca, rq) | CREATE);
18
19 MERGE :=
20     in CFA(cax, rqx).in CFA(cay, rqy).out cax(rqy).MERGE;
21
22 FA(id, ca, rq) :=
23     out CFA(ca, rq).
24     ( in ca(rqnl).REQ(id, rqnl)
25     + in rq(nf).out nf(id).LD(id, nf)
26     );
27
28 REQ(id, rqnl) :=
29     out rqnl(id).in id(nl).FL(id, nl);
```

Listing A.3: Beispiel: GSM Protocol

```
1  /*
2   * Project:   Petruchio
3   * Filename: gsm.pi
4   * Created:   16.05.2007
5   * Author:    Tim Strazny
6   * Remarks:   This translation of the GSM handover protocol
7   *            into the pi calculus is taken from Fredrik
8   *            Orava and Joachim Parrow: "An Algebraic
9   *            Verification of a Mobile Network", p. 11, Fig 2.
10  *            A formal specification of the handover procedure.
11  *            (1 Mobile Station, 2 Base Stations, 1 Mobile
12  *            Switching Center, 1 channel per Base Station)
13  */
14 
15 CC(fa, fp, l) :=
16       ( cin(v).fa<data>.fa<v>.CC(fa, fp, l)
17       + l(m_new).fa<ho_cmd>.fa<m_new>.
18            ( fp(x).[x=ho_com].fa<ch_rel>.fa(m_old).l<m_old>.
19                    CC(fp, fa, l)
20            + fa(y).[y=ho_fail].l<m_new>.CC(fa, fp, l)
21            )
22       );
23 
24 HC(l, m) :=
25       l<m>.l(m_new).HC(l, m_new);
26 
27 MSC(fa, fp, m) :=
28       new l.(HC(l, m) | CC(fa, fp, l));
29 
30 BSp(f, m) :=
31       m(x).[x=ho_acc].f<ho_com>.BSa(f, m);
32 
```

```
33 BSa(f, m) :=
34       f(x).
35       ( [x=data].f(v).m<data>.m<v>.BSa(f, m)
36       + [x=ho_cmd].f(v).m<ho_cmd>.m<v>.
37             ( f(y).[y=ch_rel].f<m>.BSp(f, m)
38             + m(z).[z=ho_fail].f<ho_fail>.BSa(f, m)
39             )
40       );
41
42 MS(m) :=
43       m(x).
44       ( [x=data].m(v).cout<v>.MS(m)
45       + [x=ho_cmd].m(m_new).
46             ( m_new<ho_acc>.MS(m_new)
47             + m<ho_fail>.MS(m)
48             )
49       );
50
51 P(fa, fp) :=
52       new m.(MSC(fa, fp, m) | BSp(fp, m));
53
54 Q(fa) :=
55       new m.(BSa(fa, m) | MS(m));
56
57 main
58 SYSTEM :=
59       new fa, fp.(P(fa, fp) | Q(fa));
60
61 SENDER :=
62       new d.cin<d>.SENDER;
63
64 RECEIVER :=
65       cout(d).RECEIVER;
```

Listing A.4: Beispiel: Handover Protocol

```
1  /*
2   * Project:   Petruchio
3   * Filename:  handover.pi
4   * Created:   18.06.2007
5   * Author:    Roland Meyer
6   * Remarks:   Modelling the handover protocol of the car
7   *            platooning case study, A. Hsu, F. Eskafi,
8   *            S. Sachs, and P. Varaiya: Design of Platoon
9   *            Maneuver Protocols for IVHS
10  */
11
12 main
13 SYSTEM :=
14     new app, appf.(EL(app, appf) | ENV(app, appf));
15
16 ENV(app, appf) :=
17     new id.app<id>.appf(x).new id.app<id>.appf(x).
18     new id.app<id>.appf(x).new id.app<id>.appf(x).
19     new id.app<id>.appf(x).0;
20
21 EL(app, appf) := // empty list
22     in app(nc).out appf(nc).LE(app, appf, nc);
23
24 LE(app, appf, car) := // list end
25     in app(nc).out appf(nc).new appn, appnf.
26     ( LI(app, appf, car, appn, appnf)
27     | LE(appn, appnf, nc)
28     );
29
30 LI(app, appf, car, appn, appnf) := // list item
31     in app(nc).out appn(nc).in appnf(x).out appf(x).
32         LI(app, appf, car, appn, appnf);
```

Abbildungsverzeichnis

2.1 Ein Petrinetz 10

3.1 Beispiel einer Auffaltung 35
3.2 Ein Problemfall für die starke Faltung 36
3.3 Vergleich 1: Auffaltung und sequentielle Semantik 38
3.4 Vergleich 2: Auffaltung und sequentielle Semantik 39

7.1 Fälle von „don't know“ 97

9.1 Schnittstellen 108
9.2 Klassen 109

Listings

3.1 Berechnung der strukturellen Semantik 25
3.2 Approximation der sequentiellen Semantik 29
3.3 Berechnung von Fragmenten 40
4.1 Überführen in Restriktionsform 42
4.2 Berechnung der Restriktionsform: Flache Form 48
5.1 Berechnung von Reaktionen eines Prozesses 52
5.2 Berechnung von Reaktionen: Kommunikationswege 54
5.3 Berechnung von Kommunikationen zweier Prozesse 56
6.1 Strukturelle Kongruenz 68
6.2 Strukturelle Kongruenz: Substitutionen 71
6.3 Strukturelle Kongruenz: Präfixe 73
6.4 Strukturelle Kongruenz: Kommutativität 75
7.1 Erweiterung eines Überdeckungsgraphen 83
7.2 Markieren überdeckter Zustände 86
7.3 Berechnen von Feuerbarkeit 90
8.1 Auffaltung 100
9.1 Verwendete π-Grammatik 110
9.2 Windows-Batch-Skript 116
9.3 Unix-Bash-Skript 117
A.1 Beispiel: Mobile Phones III
A.2 Beispiel: Car Platoon V
A.3 Beispiel: GSM Protocol VI
A.4 Beispiel: Handover Protocol VIII

Literaturverzeichnis

[AM02] R. M. Amadio und C. Meyssonnier. On Decidability of the Control Reachability Problem in the Asynchronous π-Calculus. *Nordic Journal of Computing*, 9(1):70–101, 2002.

[ANT] Homepage des LL(*)-Parsergenerators ANTLR (Another Tool for Language Recognition). `http://www.antlr.org/`, letzter Zugriff 20.07.2006.

[BG95] N. Busi und R. Gorrieri. A Petri Net Semantics for π-Calculus. In: *Proceedings of the 6th International Conference on Concurrency Theory, CONCUR 1995*, Band 962 von *LNCS*, S. 145–159. Springer-Verlag, 1995.

[Dam93] M. Dam. Model Checking Mobile Processes. In: *Proceedings of the 4th International Conference on Concurrency Theory, CONCUR 1993*, Band 715 von *LNCS*, S. 22–36. Springer-Verlag, 1993.

[DM06] G. Delzanno und R. Montagna. On the Reachability Problem for Fragments of Mobile Ambients with Name Restriction. In: *Infinity 2006*, 2006.

[EG04] Joost Engelfriet und Tjalling Gelsema. A new natural structural congruence in the pi-calculus with replication. *Acta Inf.*, 40(6):385–430, 2004.

[FGMP03] G.-L. Ferrari, S. Gnesi, U. Montanari und M. Pistore. A Model-Checking Verification Environment for Mobile Processes. *ACM Transactions on Software Engineering and Methodology*, 12(4):440–473, 2003.

[Mey07a] Roland Meyer. Structure and Concurrency in the π-Calculus. July 2007.

[Mey07b] Roland Meyer. A Theory of Structural Stationarity in the π-Calculus. June 2007.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999. ISBN: 0521658691.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. Technische Hochschule Darmstadt, 1962.

[PW02] Lutz Priese und Harro Wimmel. *Theoretische Informatik: Petri-Netze*. Springer-Verlag, 2002. ISBN: 3-540-44289-8.

[Sed03] Robert Sedgewick. *Algorithms in Java, Third Edition, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley, 2003. ISBN: 0-201-36120-5.

[Sed04] Robert Sedgewick. *Algorithms in Java, Third Edition, Part 5: Graph Algorithms*. Addison-Wesley, 2004. ISBN: 0-201-36120-3.

Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich diese Arbeit selbständig angefertigt habe und alle Teile, die wörtlich oder inhaltlich anderen Quellen entstammen, als solche kenntlich gemacht und in das Literaturverzeichnis aufgenommen habe. Diese Arbeit wurde weder in dieser noch einer ähnlichen Form einer anderen Prüfungsbehörde vorgelegt.

Tim Strazny

Oldenburg, Juli 2007